ELSEVIER

Microprocessing and Microprogramming 41 (1995) 245–259

Microprocessing and Microprogramming

Parset: A language construct for system independent parallel programming on distributed systems

Rushikesh K. Joshi, D. Janaki Ram *

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

Received 31 May 1994; revised 18 November 1994; accepted 20 January 1995

Abstract

Parallel programming on loosely coupled distributed systems involves many system dependent tasks such as sensing node availability, creating remote processes, programming inter-process communication and synchronization, etc. Very often these system-dependent tasks are handled at the programmer level. This has complicated the process of parallel programming on distributed systems. The portability of these programs is also severely affected. The programmer may also start his remote processes on heavily loaded nodes, thereby degrading the overall performance of the system. To overcome these difficulties, we introduce a language construct called parset at the programming level. Parset captures various kinds of coarse grain parallelism occurring in distributed systems. It also provides scalability to distributed programs. We show that this construct greatly simplifies writing programs on distributed systems providing transparency to various system dependent tasks.

Keywords: Distributed parallel program; Language construct; Loosely coupled distributed system; Parset; System independence

1. Motivation and introduction

In the last decade, there has been a significant amount of interest shown in the development of parallel programs on loosely coupled distributed systems. An example of such a system is a set of powerful multiprogrammed workstations connected through a local area network. Several mechanisms for performing inter-process communication, synchronization, mutual exclusion and remote accession have been proposed to live with the distributed nature of these systems. Some examples of these mechanisms are client-server communication, message passing multiple programs, and remote procedure calls (RPC). An excellent review of such programming paradigms for distributed systems appeared in [1]. Writing parallel programs on loosely coupled systems demands effective use of these paradigms in the program. For example, in [2], Bal et al. demon-

* Corresponding author.

strate how RPCs [6] can be used for writing parallel programs. In addition to making an effective use of these paradigms, several tasks such as creating remote processes on various nodes and writing external data representation (XDR) routines have to be performed by the programmer. Distributed programs written with these mechanisms are generally difficult to understand and to debug. A programmer can start his processes on heavily loaded nodes, thereby causing severe load imbalances resulting in under-utilization of the network. This can also adversely affect the performance of other programs running in the network. A key property of distributed systems is that they are open-ended. Various system parameters like node configuration and node availability keep changing over a period of time. In such cases, the programs need to adapt themselves dynamically to the changing system configurations.

Programming tools and languages such as ConcurrentC [5], P4 [3] and PVM [8] exist for developing such parallel programs. These systems have greatly unified the concept of coarse grain parallel programming on several architectures by providing the architecture independent interface to the programming paradigms in distributed systems. But the programmer has not been relieved from the burden of distributed programming and he needs to write the program as a collection of processes which communicate and synchronize explicitly.

Thus, there is no clear separation between the programmer's concerns and the system's concerns in the present approaches to distributed programming. It has become necessary to provide high level language constructs which can do this task. These constructs should be provided with adequate low level runtime support which can achieve the separation between the system's concerns and the programmer's concerns. These language constructs can be provided as extensions to existing programming languages. The programmer can specify his coarse parallel blocks within his program by making use of these high level language constructs. These language constructs are suitably translated and handled by the low level system mechanisms. The various advantages of using such constructs are listed below.

Advantages from the system's point of view

If the selection of nodes for performing computation is made at the programming level, the programmer can write programs which can generate heavy load imbalances in the system. For example, P4 gives the choice of node selection to the user. At the time of node selection, the user may not be in a position to predict the future load on the selected nodes.

Also, the programs may not make use of dynamically changing loads on the machines due to its rigid process configuration. For example, a program in PVM may create a fixed number of processes. In such a case, the program will not be able to utilize the additional capacity in the system if some nodes become lightly loaded when the program actually starts executing.

With appropriate high level language constructs, the programmer can only express the willingness of parallel execution. With this, the programs need not be modelled as a preconfigured collection of processes. This provides maximum flexibility to the system to make an effective use of the available resources.

Advantages from the programmer's point of view

(i) The user can be alleviated from the burden of creating processes and performing explicit communication and synchronization between them.

(ii) The number of available nodes and their interconnection pattern varies from one distributed system to the other, and also from time to time in a single distributed system. The load on the machines frequently keeps changing. In such a case, if a program is written as a collection of a fixed number of processes, it cannot make use of the dynamically changing loads in the system. By using the high level language constructs, the programs can be written in a system independent fashion, thereby making them scalable and portable.

(iii) Several programming errors, which occur while programming inter-process communication, synchronization, termination etc. can be avoided by programming with such language constructs.

In this paper, we present a language construct called *parset* and the low level runtime support for implementing it. Parsets can be used for expressing coarse grain parallelism on distributed systems. The parset construct consists of a data structure and a set of functions which operate on this data structure. The construct has been specially designed for capturing several kinds of coarse grain parallelism occurring in distributed systems. The use of parsets relieves the programmer from the burden of handling the remote processes, inter-process communication, remote procedure calls, etc. A low level distributed parset kernel creates subtasks, locates suitable remote nodes and gets the code executed on the remote nodes. This makes the programs written using parsets scalable over varying system parameters. Thus, parsets draw a clear distinction between the system's concerns and the programmer's concerns.

We develop the parset constructs in Section 2. Parsets can capture both SPMD (Single Program, Multiple Data) and MPMD (Multiple Program, Multiple Data) kinds of parallelism. This is explained in Section 3. We implemented the parsets on a network of workstations. This implementation with a case study of an application program is discussed in Section 4.

2. Semantics of the parset construct

In this section, we first describe the parset data structure with the basic operations which manipulate the data structure. When functions receive parsets as their arguments, they derive special meanings. The function semantics on parsets is explained subsequently. Finally in this section, a special case of parset, called *indexparset* is described.

2.1. The parset data structure

Parset is a set type of data structure. The elements of a parset can be:

- basic data types,
- untyped,
- functions.

When the elements of a parset belong to a basic data type, it is called a *simple parset*. When the elements are untyped, it is an *untyped parset*. A *function parset* has functions as its elements. Simple parsets can be used in expressing SPMD kind of parallelism. This is discussed in detail in Section 3.1. Untyped parsets find applications in expressing MPMD kind of parallelism using polymorphic functions [4]. Function parsets are the most general ones, and using them, it is possible to express the MPMD parallelism in a general way. The MPMD parallelism using parsets is discussed in detail in Section 3.2.

A parset is kept logically ordered based on the entry of its elements on a first-come-first-served basis. Cardinality is an attribute associated with a parset. The cardinality of an empty parset is zero. A typed parset declares the type of the elements held by the parset. For example,

parset P of int;

declares a parset P of elements of type integer.

The operations that can be performed on parsets are **insert**(), **flush**(), **get**(), **delete**(), **getcard**() and **setcard**(). The functions that can be executed on parsets have their arguments tagged as **RO** (read-only), **RW** (read-write) or **WO** (write-only). This scheme is very similar to the Ada language approach which places the reserved keywords *IN*, *OUT*, and *INOUT* before the arguments. The semantics of these tags are as follows:

WO: The argument is only modified inside the function but not read.

RO: The argument is passed to the function only for reading.

RW: The argument is read as well as written to inside the function.

When functions taking parsets as arguments are executed, the arguments are locked in a proper mode. For example, if an argument is tagged as **RO**, it is locked in read mode. This tagging scheme allows the exploitation of parallelism in control flow. When two functions are sequenced one after the other in a program, they can be run in parallel if the earlier function does not lock the arguments needed for execution of the second function. For example, if two functions take the same argument tagged **RO**, they can be executed in parallel. On the other hand, if the argument is tagged as **RW**, unless one function releases the locks on the argument, the other function cannot start execution. The argument tagging performs an important role in the case of parsets and has implications in implementation. This is discussed in detail in Section 4.4.2. The operations which manipulate the parsets are as follows:

insert (WO P, **RO** i, **WO** order)
Inserts an element *i* in the parset *P* as its last element. The cardinality of *P* increases by 1 after the insertion operation. The argument *order* returns the order of the inserted element. P is tagged as **WO** here as the function writes an element into the parset but does not read any element from it.

flush (WO P)
Flushes the parset *P*. After flushing, the cardinality of *P* becomes zero.

delete (WO P, **RO** n)
Deletes the *n*th element from the parset *P*. The delete operation leaves the order of the parset intact.

get (RO P, **RO** n, **WO** element)
The argument *element* returns the *n*th element of the parset *P*.

getcard (RO P, **WO** c)
The argument *c* returns the cardinality of the parset *P*.

The above operations provide means for manipulating the elements of a parset. When functions are called on parsets, they acquire special meanings. This is explained below.

2.2. Function semantics on parsets

When a parset is passed as an argument to a function, three possibilities exist for the execution of the function. The function can execute in parallel on each element of the parset. This is the first type of execution. The function can also execute sequentially on each element of the parset one after the other. This is the second type of execution. In the third type of execution, the function takes the parset as a simple argument for further processing within the function. To differentiate between these three function types, the two keywords **par** and **seq** are used. We now illustrate the three types of function calls with simple examples.

par function call:
Example: **par** process (**RO** P);
The function process() is applied to each element of *P* in parallel. Each activation proceeds asynchronously. If a function has more than one parset as its arguments, then the cardinalities of all of them must be the same. This will enable a particular function activation to pick up the corresponding element from each parset. The *par* function call exploits the data parallelism expressed in a parset.

seq function call:
Example: **seq** print (**RO** P);
The function print() is applied to each element of *P* sequentially in the order of the elements.

Ordinary function call:
Example: myprint (**RO** P);
Here no keyword is prefixed to the function call. Hence this is treated as an ordinary function call, and the parset *P* is passed just as a plain argument to it.

As an example, the following can be the description of myprint():

```
Function myprint (RO P) {
        seq print (P);
}
```

A special function called myid() is provided to know on which element of the parset the present function activation is operating. This function returns the order of the element of the parset on which a *par* or a *seq* function call is operating.

2.2.1. Defining the functions which execute on a parset

When a *par* or a *seq* function is called with a parset as its argument, each activation of that function receives one element of the parset. Hence, the function is defined for one element of the parset. On the contrary, a function, which takes a parset as a plain argument as in the case of an ordinary function call, declares its argument type the same as the parset type itself. The following example illustrates the difference. In this example, add() concatenates strings and howMany() tells the cardinality of a collection of strings.

```
Function MassConcat() {
      ...
      parset P of string = "Work", "Think",
      "Speak";
      parset Q of string = "hard", "deep",
      "truth";
      parset R of string;
      int n;
            ...
            ...
            ...
            par add (P, Q, R);
            howMany (R, n);
            ...
}
Function add (string RO x, string RO y, string WO
z){
            concat (x, y, z);
}
Function howMany (parset RO strset of string, int
WO n){
            getcard (strset, n);
}
```

Each activation of function add() binds an element of parset P to argument x, and the corresponding element of parset Q to y. The returned argument z is bound to the corresponding element of R. The cardinality of the parset P and Q must be the same in this case.

2.2.2. Concurrent execution of multiple function calls on parsets

There can be situations in which a parset, which is an output parameter of one function call, becomes an input parameter in a subsequent parset function call. This offers additional possibilities of concurrent execution. This is explained in the following example.

```
Function Encourage () {
          ...
          parset E of employee, A of assessment, R
          of reward;
                  par assess (RO E, WO A);
                  par encourage (RO A, WO R);
                  ...
}
```

In the function Encourage() as given above, the employee records are assessed so as to encourage the employees with suitable rewards. The output of assess(), which is parset A, is an input to encourage(). As soon as the function assess() finishes with any one of its multiple activations, the **WO** lock on the corresponding element of the parset will be released. After the release of the lock, the next function encourage() acquires the **RO** lock for that particular element of the parset. Once the lock is acquired, the function can start its execution. Thus, multiple functions can execute concurrently providing additional parallelism in control flow.

2.3. Indexparset

The indexparset is a special case of parset. It holds no elements but carries an index. The index can be seen as cardinality of the indexparset. A function call on an indexparset is activated index number times. Hence the indexparsets can be used when multiple activations of the same function are required. An indexparset can be declared as:
indexparset I;
Only three operations namely **setcard()**, **getcard()**, and **flush()** are performed on an indexparset. The last two are the same as described in Section 2.1 on parsets. Operation **setcard()** sets the cardinality of the indexparset *I* to a given value *c* and is defined as:
setcard (WO I, **RO** c);
When an indexparset becomes an argument to a *par* or a *seq* function call, each function activation receives an integer which represents the order of that particular activation. The following example demonstrates the use of an indexparset. It collects the status of distributed resources in a parset called StatusSet.

```
Function CollectStatus {
         indexparset I;
         parset StatusSet of int;

                setcard (I, 10);
                par myread (I, StatusSet);
                           /* The activations of
                           myread collect
                           the status of 10 re-
                           sources */
                ...
}
```

```
Function myread (int RO ResourceId, int WO s)
{
                /* ResourceId = current function
                activation number */
                ReadStatus (ResourceId, status);
                s = status;
}
```

3. Expressing parallelism through parsets

Parsets can be employed for expressing both SPMD and MPMD kinds of parallelism. This is discussed in the subsequent sections.

3.1. Expressing the SPMD parallelism through simple parsets

The SPMD parallelism can be expressed by using simple parsets with *par* function calls. When a *par* function is called on a simple parset, the function executes in parallel on different elements of the parset. The simple parsets can be created in two ways. An empty parset can be declared initially and the elements can be inserted into the parset explicitly by insert calls. The other method of creating simple parsets is to convert an array of elements into a parset with a grain control mechanism. These two methods of creating simple parsets are explained below:

3.1.1. Creation of simple parsets using explicit insert() calls

First a simple parset of the desired data type is declared. Multiple data belonging to the same data type can now be added to the parset using the **insert()** function. Then a function can be executed on this parset by a *par* function call.

To express the SPMD parallelism in processing of arrays, one may create a parset and explicitly insert the array elements into the parset with this method. An easy way to convert an array into a parset is through the grain control mechanism, which is specially designed for this purpose.

3.1.2. Conversion of arrays into simple parsets by the grain control mechanism

Through this mechanism, one can indicate the granularity that is desired to build such a parset out of an array. The mechanism consists of two constructs, namely *granularity*, which is a metatype, and a function *CrackArray()*. The granularity works as a metatype in the sense that its value is a data type.

As an example, we may specify a granularity of *int[100]* to convert an array of type *int[1000]* into a parset. The parset will have 10 elements, each of type *int[100]* as specified by *granularity*. It is possible to covert a multidimensional array into a parset by cracking the array in any dimension. For example, we may specify a granularity of *int[25][100]* to convert a row major array of type *int[100][100]* into a parset. In this case, the parset will have 4 elements each of type *int[25][100]*.

We take the following image transformation example to demonstrate the use of this mechanism. In this example, the array named A is converted into a parset. The array A consists of 1000 elements. Each element of the parset is constructed by combining 100 elements of the array. Thus the parset will have 10 elements in it.

```
Function  ProcessArray () {

          int A [1000];
          granularity g = int [100];
          parset  P, Q of g;

                  CrackArray (A, P, g);
                  par transform (P, Q);
                  par plot (Q);
                  flush (P);
}
```

Converting a data structure like an array into a group of several grains of specific granularity becomes possible with the metatype *granularity*. The target parset is declared as a simple parset of the same data type as that of the granularity. The array is converted into a parset using the function *CrackArray()*. The function takes three arguments: the source array, the target parset handle and the granularity. When the function returns, the parset handle corresponds to the new parset that is built out of this array.

After the conversion, there are two ways to access the array. One is by manipulating the new parset handle, and the other is by directly using the array name. The array name refers to the copy of the array which is local. The parset handle refers to the copy of the array which may be scattered in the network. Now if both the handles are allowed to manipulate the array, it will create inconsistencies between these two copies. Hence till the parset handle is active, the array must be accessed only through the parset handle. However, a **flush**() call on the parset handle has a special meaning of deactivating the handle and storing back the new values of the array into its local copy. After the flush, the array can be referenced in the normal way.

By setting the granularity, the user can set an upper bound on the number of processors that can be utilized by the underlying parset kernel for an execution. For example, for a single dimensional array of size 1000, the granularity may be set to 100 or 250, thereby creating parsets of size 10 or 4. Thus a varying degree of parallel execution on a parset can be obtained using granularity.

The grain control mechanism thus achieves two goals. Firstly, it allows an array to be treated directly as a parset without the need of multiple **insert**() calls. Secondly, it can control the degree of parallel execution of a function on an array.

Dynamic specification of granularity

When a granularity variable is declared as an array, the dimensions of the array can be specified during runtime. The following example demonstrates this:

```
Function ProcessArray () {

         int A [1000];
         granularity g = int [];
         parset P, Q of g;
         int dim;
                dim = 100;
                CrackArray (A, P, g [dim]);
                par transform (P, Q);
                par plot (Q);
                flush (P);
}
```

It can be noted that the value of the granularity variable has been declared as an integer array of single dimension without mentioning its dimension value. The dimension of 100 has been specified during runtime in this example.

3.2. Expressing MPMD parallelism through parsets

The MPMD parallelism can be expressed with parsets using two mechanisms. The first is to use polymorphic functions with untyped parsets. It offers a limited way of expressing MPMD parallelism, and can be readily implemented in a language which provides function polymorphism. The other mechanism is a general mechanism and uses function parsets. These mechanisms are described in the following subsections.

3.2.1. MPMD parallelism through polymorphic functions

Untyped parsets can be used for expressing MPMD parallelism. In a typed parset, only the elements of the specified type can be inserted, whereas in untyped parsets, elements of any type can be inserted. An untyped parset declaration does not mention the type of its elements. As an example, an untyped parset Q can be declared as:
parset Q;
In this case, a parset is created as a collection of data belonging to different types. A polymorphic function identifies the type of each element and executes the required code on it. For example, with a function call such as **par** process (Q), various activations of the function process() may receive arguments of different types. In this way, we can obtain the concurrent execution of different codes on different data.

But this mechanism can not fully capture the MPMD parallelism for the following reason. Always the same piece of code is executed on two different elements of a parset if they are of the same type. So we cannot execute different codes on the elements of the same type by this mechanism. By using the function parsets, this limitation can be overcome.

3.2.2. MPMD parallelism with function parsets

A function parset can hold a collection of functions. Another parset is used to hold the corresponding argument set. Cardinalities of these two parsets must be the same. The function parset can now be invoked on the corresponding argument parset to achieve the general MPMD parallelism. Following example shows the structure of such a program.

```
Function MPMD-Prog-Structure () {

         int d11, d12, d3;
         char d2;
         parset F of func = {f1 (int, int), f2 (char),
         f3 (int)};
         parset D1 = {d11, d2, d3};
         parset D2 = {d12, NIL, NIL};

                  par (F) (D1, D2);
}
Function f1 (int RO d11, int RO d12) {
                  ...
}
```

```
Function f2 (char RO d2) {
        ...
}
Function f3 (int RO d3) {
        ...
}
```

In the above example, function f1() will take its first argument from the first element of parset *D1*, and the second argument from the first element of parset *D2*. Similarly, the functions f2() and f3() pick up their arguments from *D1* and *D2*.

4. Implementing parsets on a loosely coupled distributed system

We discuss an implementation of the parset constructs on a network of workstations consisting of Sun 3/50 and Sun 3/60s, running SunOS version 4.0.3. The environment supports NFS, the network file system. TCP/IP has been used for inter-process communication. The parset constructs are provided as extensions to the C language. A parset preprocessor translates the parset constructs into C code. An overview of the implementation is now given followed by the description of various components. We studied the performance of an application using the implementation.

4.1. Overview of the implementation

The heart of the implementation consists in a distributed parset kernel. The kernel is divided into resident and volatile parts. Each parset has an associated process called *P-Process* to maintain and manipulate the elements of the parset. A *par* function call on a parset is executed with the help of separate processes called *E-Processes*. *P-Processes* and *E-Processes* form the volatile kernel. *P-Processes* reside on the same node where the user program resides. *E-Processes* reside on different nodes to exploit the parallelism.

The Resident kernel consist of daemon processes which are started during the boot-up time on the machines which are willing to participate in the execution of programs that use parsets. The copy of the resident kernel is obtained by nodes from a designated node using the NFS. In this way both diskless as well as diskful machines can obtain the resident kernel. The resident kernel manages the *P-Processes* and the *E-Processes*. It provides an interface to user programs to create the parset processes and to execute various functions on it.

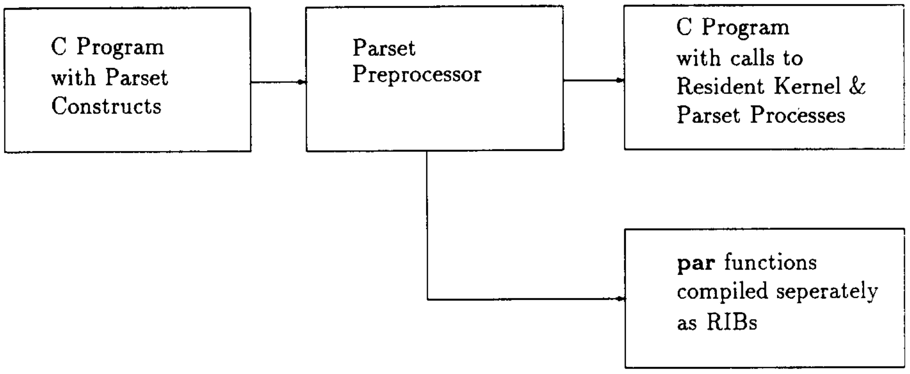

C Program with Parset Constructs
Parset Preprocessor
C Program with calls to Resident Kernel & Parset Processes
par functions compiled seperately as RIBs

Fig. 1. The parset preprocessor.

4.2. The parset preprocessor

Fig. 1 shows the functionality of the parset preprocessor. The high level user program is provided with clean parset constructs, as described in the earlier sections, which hide the underlying distributed implementation. The parset preprocessor then translates the user program into a low level C code. The low level code makes calls to the resident kernel and the parset processes.

The parset preprocessor identifies the *par* function calls separately as Remote Instruction Blocks (RIBs). An RIB contains code which can be migrated at run time to a remote node. Migration of RIB code is the migration of the passive RIBs and not active processes or tasks. The RIB is given control for actual execution at the remote node. RIBs are named and compiled separately by the preprocessor.

4.3. Remote Instruction Blocks (RIB)

The RIB facility was developed during the course of this implementation. The facility is similar to remote evaluation (REV) [7] developed by Stamos and Gifford. With the RIB primitives, as opposed to the RPC primitives [6], we can migrate a code to be executed to a remote site at runtime and get it executed there.

The parset preprocessor names and compiles the RIBs separately. Whenever a block has to be executed on a remote node, its name and address is made known to the remote node. The remote node can access the RIBs using the NFS, and can execute it as and when required.

4.4. The distributed parset kernel

The parset kernel is distributed over the network. It consists of a resident and a volatile part. The

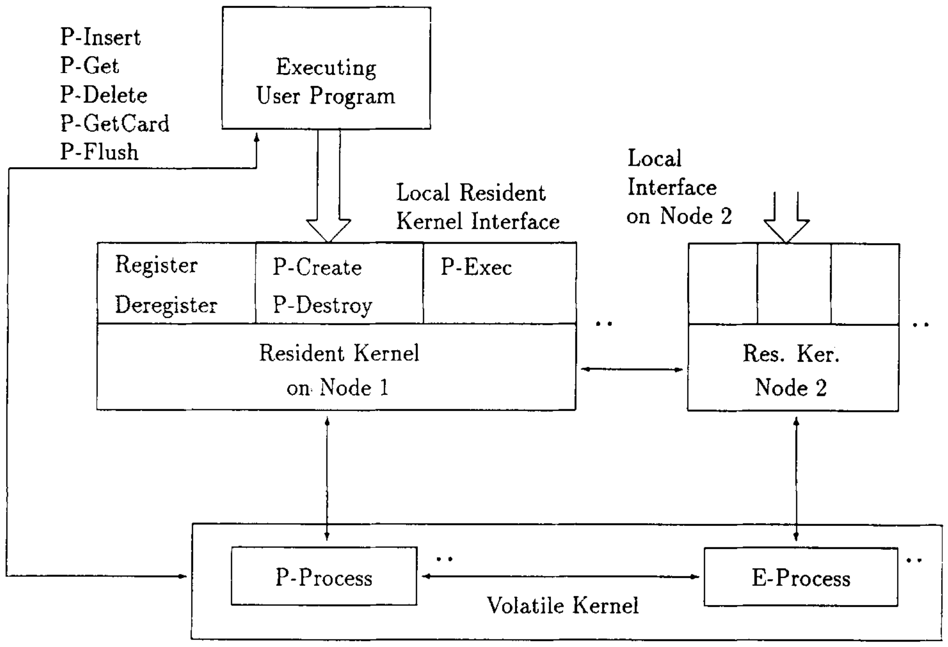


Fig. 2. The parset system components.

resident part of the kernel is always present on all the nodes that participate in distributed execution. The volatile part is dynamically created on selective nodes depending upon the requirements. Fig. 2 shows the organization of various components in the implementation. The functionalities of each are described below.

4.4.1. The resident kernel

The resident kernel performs the low level system dependent tasks which deal with the primitives for RIBs, creation and termination of *P-Processes* and *E-Processes*, etc. It is responsible for selecting the lightly loaded nodes for execution and managing the distributed execution. Each local resident kernel contacts other resident kernels and finds out the lightly loaded nodes. Achieving the program scalability is also a function of the kernel.

The resident kernel provides an interface to low level user programs. The interface consists of the calls for registration and deregistration of a user process, creation and destruction of parset processes, and notifications of function execution on parsets. When an executing user program declares a parset variable, the resident kernel spawns a parset process called *P-Process*. This *P-Process* manages the parset on the node where the user program runs. Any further manipulations performed by the user program on this parset with **insert**(), **get**(), **delete**(), **getcard**() and **flush**() calls, are directed to the corresponding *P-Process* bypassing the resident kernel.

During a remote execution of a function in a *par* call, the resident kernel creates new *E-Processes*, or locates free executor processes depending on their availability on the nodes suitable for the execution. For a particular function execution, the resident kernel helps *P-Processes* and the corresponding *E-Processes* to synchronize. After the synchronization, *P-Processes* send the required parset elements to *E-Processes*.

To improve the execution efficiency, the kernel can combine several elements in a parset together to form a super-grain. It can be noted that the super-grain formation is different from the grain control mechanism. The later is a mechanism meant for the user whereas the former is used by the kernel, making it transparent to the user. The resident kernel guides the volatile kernel for the super-grain formation.

4.4.2. The volatile kernel

The volatile part of the kernel consists of *P-Processes* and *E-Processes* as referred earlier. For a function execution, the *P-Processes* which correspond to parsets involved in the execution, perform the required inter-process communication with the allotted *E-Processes*. In this fashion, all *P-Processes* and *E-Processes* proceed concurrently. An *E-Process* can be allocated for other incoming execution requests after the current request is completely processed. Similarly, a *P-Process* is derailed by the resident kernel when it receives a **destroy**() call.

As discussed earlier, a *P-Process* manages one single parset variable. A parset variable is a collection of multiple grains (elements). When a *par* or a *seq* function takes a parset variable as its argument, it specifies the parset variable as **RO**, **WO**, or **RW** variable depending on its usage inside the function. At the time of actual execution, the elements of the parset have to be locked according to these specifications in order to exploit the concurrency as discussed in Section 2. In case of multiple parallel activations of a function, each activation operates on a different grain. Hence each activation can proceed independently as long as it can obtain the required locks.

At the first sight, it appears that in the place of **WO** locks, one may use **RW** locks, thereby eliminating the need of additional **WO** locks. This is based on the fact that along with a write permission, there is no harm in granting a read permission also. But it is not so in reality. Because, when an argument gets a read lock, it has to be moved to the remote site where the function is executing. But a **WO** argument need not be moved to the site of the remote function, since the function does not read the value of this argument. Thus these three distinct locks help in minimizing the communication overhead.

With an execution request of a *par* or a *seq* function, a parset process first establishes contact with the remote *E-Process* through the resident kernel. Then it continues to set the locks on the grains. As and when a lock is set, the following action is taken corresponding to each lock:

RO: (1) send a copy of the grain to the remote *E-Process*
(2) release the lock on the grain

WO: (1) Receive the grain value from the remote *E-Process*
(2) Update the grain value in the parset
(3) Release the lock

RW: (1) send a copy of the grain to the remote *E-Process*
(2) Receive the grain value from the remote *E-Process*
(3) Update the grain value in the parset
(4) Release the lock

4.5. A case study

In this section, we discuss a simple case study of the image transformation problem encountered in computer graphics. Two examples of such transformations are rotation and dragging. Fig. 3 shows the test program for image transformation using parsets. The program was run on a network of Sun workstations. The transformation problem being an SPMD type of problem, we used the techniques presented in Section 3.1.2.

The following observations can be made with respect to the case study program:

- The program only expresses the parallelism present in solving the graphics rotation problem without any explicit use of system dependent primitives.

```
#define TotalPoints 20000
#define GrainSize 400


typedef struct { int x, y ; } point ;
typedef point grain [GrainSize] ;


Main:
        point image [TotalPoints] ;
        granularity G = grain ;
        parset P of G ;


                read-image (image) ;
                CrackArray (image, P, G) ;
                par transform (P) ;
                flush (P) ;
EndMain



Function: transform ( grain RW image-grain) {
        int i;
                for (i = 0; i < GrainSize; i++)
                image-grain = rotate (image-grain) ;
}
```

Fig. 3. The image transformation problem.

- The user can control the size of the grain by specifying the grain-size in the program.
- The function transform() will be compiled separately as an RIB so that it can be migrated to a remote node for execution. The system will make appropriate choice of nodes and the mechanism is completely transparent to the user program.
- The program is easy to understand and to debug.
- The program can be executed on a distributed system supporting the parset construct and hence can be ported easily to other systems.

Table 1
Performance of the program

Problem Size	Tseq (sec.)	Grain Size	#nodes N	Tpar (sec.)	Speedup	Utilization
2000	17.85	1000	2	10.71	1.66	83
		500	4	7.50	2.38	59
4000	36.02	2000	2	20.36	1.76	88
		1000	4	12.84	2.80	70
10000	90.04	5000	2	48.00	1.87	93
		2500	4	26.48	3.40	85
		1250	8	16.93	5.31	66

- In the case of node failures, the kernel can reassign the subtasks which remain unevaluated to other nodes without the involvement of the user.

The above program has been executed on the current implementation of parsets on Sun 3/50s and 3/60s, and the performance has been measured.

The program performance

Table 1 summarizes the performance of the program with various problem sizes on a varying number of nodes. As the problem size increases, it is possible to use more number of nodes. The upper bound on the number of nodes that the system can choose for a given problem size is dictated by the grain size as discussed in Section 3.1.2. From the table, it can be observed that as the problem size increases, it is possible to employ more nodes thereby achieving better speedup and utilization.

Table 2
A comparison with PVM

Problem size	Grain size	#nodes N	Timings for parsets (sec.)	Timings for PVM (sec.)
2000	1000	2	10.71	10.04
	500	4	7.50	5.78
4000	2000	2	20.36	19.02
	1000	4	12.84	10.52
10000	5000	2	48.00	46.28
	2500	4	26.48	24.26
	1250	8	16.93	14.28

Table 3
Scalability of the program Problem size: 20000 points, sequential time: 179.99 sec

#nodes	Parallel time (sec.)	Speedup
2	93.94	1.91
4	54.50	3.30
10	25.44	7.07

A comparison with PVM

The same application program has been implemented using PVM. Table 2 shows a comparison between parsets and PVM. The execution speeds using parsets compare favorably with PVM. A negligible average overhead of less than 2 seconds was observed in the implementation of parsets.

The scalability test

The image array was converted into a parset with a grain size of 400 points. As the load on different nodes varied, the system automatically chose the nodes for executing the program. During different runs of the program, the system used nodes from 4 to 10 depending on their availability and load. The results are given in Table 3. From the table it can be seen that the parset constructs provide a high degree of scalability to the program without significant overheads. The scalability is completely transparent to the user. The system is able to handle the node availability dynamically, making use of the nodes as and when they become available on the network.

5. Discussion and future work

The parsets provide a clear separation between the system dependent tasks and the expression of parallelism. The programmer need not know about the underlying distributed system model. The user program has to only express the willingness for parallel execution through the parset constructs. The decision of the degree of parallelism that is actually exploited is delayed until run time. This decision is taken by the distributed parset kernel. For example, in the case of high activity on the network, the same program still runs on a fewer number of nodes, even on a single node. The kernel takes care of all the system dependent tasks such as node selection, process creation, remote execution and synchronization.

Each element of a parset is a potential source of concurrent execution. A *par* function call captures the data parallelism that exists among the various elements of a parset. On a single element of a parset, two separate functions can execute concurrently if they obtain the necessary lock on that element. In this way, multiple function calls on parsets capture the parallelism present in the control flow.

Fault tolerance is an important issue in distributed systems. The parsets can be made to tolerate node failures in the midst of function execution. This can be ensured at kernel level with the help of the locking mechanism without the involvement of the high level program. Whenever a node failure is detected, a new activation for the function executing on that node is spawned on a healthy node. The locks are not released till the activation returns successfully.

The RIBs used in this implementation were designed for a homogeneous system. We are planning to extend them for heterogeneous systems. This will enable us to run the parset programs over a wide area network and exploit the computation power that is available over heterogeneous systems.

6. Conclusions

We presented the parset language construct for writing coarse grain parallel programs on distributed systems. Parsets achieve a clear separation between the system's concerns and the programmer's concerns. Programmers can write neat and simple parallel programs on distributed systems using the parset constructs. A parallel program written with parsets becomes scalable and easily portable to other distributed systems supporting parsets. Language constructs like parsets can make parallel programming on distributed systems an easy task. The parset approach can prove to be an extremely promising direction for parallel programming on distributed systems.

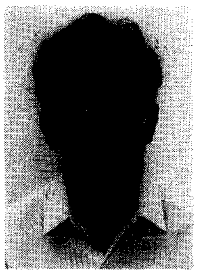
Acknowledgements

We thank the anonymous reviewers for their valuable comments which greatly improved the quality of this document.

References

[1] G.R. Andrews, Paradigms for process interaction in distributed programs, *ACM Comput. Surveys* 23 (1) (Mar. 1991) 49–90.
[2] H.E. Bal, R.V. Renesse and A.S. Tanenbaum, Implementing distributed algorithms using RPC, *AFIPS Conf. Proc.* Vol. 56, *Nat. Computing Conf.* (1987) 494–505.
[3] R. Butler and E. Lusk, User's guide to the P4 parallel programming system, Technical Report ANL-92/17, Argonne National Laboratory, Oct. 1992.
[4] L. Cardelli and P. Wegner, On understanding types, data abstractions and polymorphism, *ACM Comput. Surveys* 17 (4) (Dec. 1985) 471–522.
[5] R.F. Cmelik, N.H. Gehani and W.D. Roome, Experience with multiple processor versions of concurrent C, *IEEE Trans. Soft. Eng.* 15 (3) (Mar. 1989) 335–344.
[6] B.J. Nelson, Remote procedure calls, *ACM Trans. Comput. Sys.* (Feb. 1984) 39–59.
[7] J.W. Stamos and D.K. Gifford, Remote evaluation, *ACM Trans. Prog. Lang. Syst.* 12 (4) (Oct. 1990) 537–565.
[8] V.S. Sunderam, PVM: A framework for parallel distributed computing, *Concurrency: Practice and Experience* (Dec. 1990) 315–339.

R.K. Joshi was born in 1968 in Maharashtra, India. He obtained the B.E. degree in Instrumentation Engineering from College of Engineering, Pune in 1989, and M. Tech in Computer Sci. & Engg. from S.J. College of Engineering, Mysore in 1992. Currently, he is a doctoral student of the Dept. of Computer Sci. & Engg. at Indian Institute of Technology, Madras. His current interests include distributed and concurrent systems, programming languages, and object-oriented systems.



D. Janaki Ram was born in 1962 in Andhra Pradesh, India. He got his B.Tech in Mechanical Engineering from College of Engineering, Kakinada in 1983, his M.Tech in Computer Science in 1985 from I.I.T, Delhi, and his Ph.D degree in 1989, also from I.I.T, Delhi.

He worked in National Informatics Center at New Delhi and subsequently moved to I.I.T., Madras in 1989. He is currently working as Asst. Professor at Department of Computer Science and Engineering, I.I.T., Madras. His research interests include distributed and concurrent computing, distributed databases, object-based systems and CAD/CAM. He has published several international papers in these areas. He regularly teaches courses in the above areas at post-graduate level and has guided several B.Tech, M.Tech and M.S projects in the above areas. He has conducted several national level workshops on object oriented programming, object oriented design and unix programming environment. He has introduced the course on Object Oriented Software Development at I.I.T, Madras.