Banyan : A Language for Scalable Parallel Programming on Loosely Coupled Distributed Systems

D.Janaki Ram, M.V.Sudhakiran, T.N.Srikanta
Distributed and Object Systems Group
Department of Computer Science and Engineering
Indian Institute of Technology, Madras, INDIA
djram@iitm.ernet.in

Abstract

Parallel programming on loosely coupled distributed systems is becoming a viable approach with the rapid increase in network speeds and availability of large amounts of unused CPU capacity on individual workstations. Parallel programs are often written for a specific configuration of the distributed system such as the number of nodes, their relative speeds and their network connections. These programs perform poorly when there is a change in the configuration or when they are taken to a system other than what they are intended for. We propose a new paradigm using which one can express the scale of a program as a part of the program itself. The scale of the program is specified in an abstract manner for an arbitrary number of nodes, their relative speeds and their network connections. The runtime system uses this information to decide the actual scale of program on a given distributed system.

1 Introduction and Motivation

Current distributed systems, specifically, a loosely coupled network of workstations have tremendous unused computational power which makes them attractive for parallel programming [1] [2]. Reduced communication to computation ratios due to the recent developments in communication networks have resulted in growing interest in these interconnected workstations [3].

Distributed systems are inherently open ended. Their capacities and architectures constantly change with time. The number of nodes may range anywhere from 10 for a laboratory wide system and to 1000 for a university wide system. Writing programs for such systems can be very tedious as the programmer, while writing his programs, cannot predict the number of idle workstations available at runtime. Conventionally, parallel programs on loosely coupled distributed systems are written as a set of fixed processes or tasks that are resident on different nodes in a distributed system. When such programs are taken to a different configuration or run at a later time, they either overload or under utilize the system resources. Programs should typically be able to scale to the changing configuration or to the runtime environment so as to utilize available system resources. Scalability of parallel programs has been discussed in [4] and is done in the following ways.

- Constant problem size scaling: In this model, the size of the problem remains constant irrespective of the number of processors or nodes available for computation. Optimal number of nodes needs to be chosen.
- Memory constrained scaling: In this model, the memory requirements grow linearly with the number of processors or nodes available for computation.
- Time constrained scaling: In this model, the problem is scaled to a new configuration such that its solution always takes the same amount of time.(i.e. the user wants to solve the largest possible problem in a given time interval.)

We propose Banyan, a new paradigm for parallel programming to address the constant problem size category.

Several languages such a Parset [5], Linda [6], Orca [7], OBS [8] address the issue of parallel programming for distributed systems. Though Parset provides a transparent way of creating processes and addresses the issue of scale at runtime to some extent, the grain size is fixed by the programmer while writing his programs. Also, these languages do not specify how the program should scale when more resources are available. They do not address the adaptive nature of the programs when more computing resources are available. The key idea behind writing programs for open ended distributed systems is that programs written must be able to adapt themselves to the changing runtime environment. The program should be able to divide the problem into grains of appropriate size that can be executed efficiently on the other nodes on the network when they are available. A number of factors, like the size of the grain, communication overhead involved , the costs of splitting and aggregating, affect the performance of the program running on a network of workstations.

2 Banyan - A new approach

Banyan makes it unnecessary for the programmer to make assumptions about the underlying architecture. Here the number of processes that form the executing program depends on the availability of free nodes. The greater the number of free nodes, the greater is the number of processes that execute concurrently. The degree of parallelism is determined only at runtime.

The programmer first defines a *virtual machine* for himself which can scale with the underlying architecture. Programs are written for this *virtual machine.* By programming for the virtual machine, no assumption is made about the physical machine and the scale of the program.

This information, as to how the program should scale with the environment cannot be obtained by the system by itself and has to be specified by the programmer. However, the system will have information about the size and speed of the underlying nodes and its network connections, none of which need to be known by the programmer. Hence, this achieves a *clear separation of concerns* between the programmer and the system.

The system is allowed complete freedom in deciding nodes to be loaded and the number of processes to be created based on the current load.

3 Description of the language

We describe Banyan and its constructs with the example of an integral calculation problem using the quadrature method which is given below.

The integral of a function is to be calculated between two points. This is done by dividing the interval, say into three parts, calculating the integrals over the three sub-intervals separately and then adding up the results. The calculation of the integral over the sub-intervals can in turn be done in the same manner. Here, more the number of intervals more is the parallelism. We aim at solving this problem with the greatest possible parallelism that the underlying distributed environment can permit.

3.1 An Overview of a Banyan program

A Banyan program consists of three parts.

- First part : The definition of a *virtual machine*
- Second part : The actual code for the different parallely executing components.
- Third part : The mapping of processes to vertices in the *virtual machine.*

The key concept here is the *virtual machine.* The *virtual machine* is specified by the programmer himself. It includes the notion of *scale*, which dictates how the program is to scale with the underlying system. A programmer may construct different *virtual machines* for solving different problems. The constructs provided allow the programmer to build a variety of *virtual machines.*

We describe the *machine* construct which defines a *virtual machine*, and then discuss scaling of the machine with the underlying architecture Then we explain programming on the *virtual machine.*

3.2 The Machine Description.

The definition of a *virtual machine* is shown in figure 3a.

Node and Link Types

The components of a *machine* are vertices and connections between vertices, which are instances of different NodeTypes and LinkTypes respectively.

A *NodeType* defines a type of virtual processor. The *Power* attribute of a node indicates the amount of computational power the programmer expects it to possess. In the example given, two types of nodes are defined, one with Power 5 and the other with Power 10. These powers are relative and describe the weights of the process that will run on these nodes. The weight is in terms of the work done by a process.

Similarly, a *LinkType* defines a type of virtual connection between two nodes. The *bandwidth* attribute indicates the amount of bandwidth that can be expected from connections of that particular type.

The Machine Construct

A *machine definition* consists of three parts: Vertices section, Connections section, Fractals section.

The first part, preceded by the keyword *Vertices* defines the different virtual processors in the *machine* and their types. The above *machine Tree* has four vertices denoted by 0,1,2,3. Each vertex is an instance of a NodeType or a Machine.

The *Connections* section allows the programmer to express which nodes will communicate with which other nodes in the program. The types of the corresponding links define the bandwidth that the programmer expects out of those links. By specifying the nodes and connections in a *machine* in the above manner, the programmer expresses

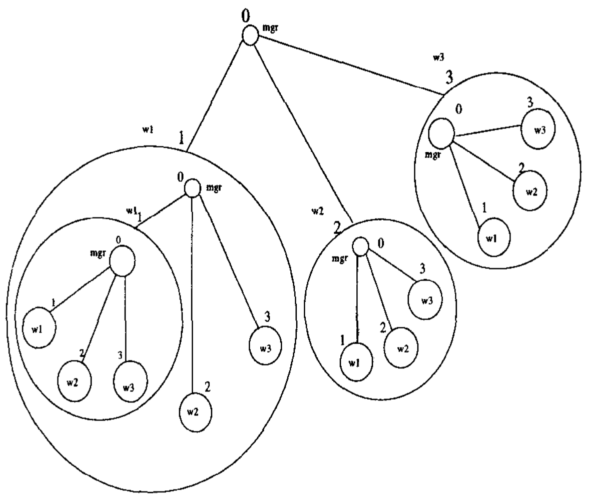

Figure 1. An example virtual machine

his requirements (of computational power and bandwidth) to the system.

The statement *Tree 1,2,3;* in the *Fractals* section, means that if the *machine* needs to be expanded, it can be done by having another *machine Tree* instead of either nodes 1,2 and 3. This is the mechanism with which the programmer tells the system how to expand the *virtual machine* if more computational resources are available. The *Fractals* section is optional in a *virtual machine.*

The above definition of a *virtual machine* is very similar to an inductive definition. The *Vertices* and *Connections* section together define a minimum *virtual machine* and hence can be thought of as the *basis* clause. The *Fractals* section tells how to extend machines to get bigger machines. Hence, it can be thought of as the *inductive clause.* Any *machine* which can be generated by this inductive definition is called a valid *machine* and the set of all such machines is called the *valid set.* By defining the machine as above, we are defining a set of *machines,* which is the *valid set* and agree that any of this configuration is acceptable. The system, depending on the state at runtime, will construct one of the valid configurations for the programmer. The *virtual machine* adapts itself to the available resources at run-time.

A *virtual machine* which obeys the inductive definition of *Tree* is shown in Figure 1. At the highest level, there are 4 vertices (0,1,2,3). As can be seen from the *Fractals* part of the machine definition, vertices 1,2 and 3 can themselves be *machines* of type *Tree.* In this case, each of them is a *Tree* by itself. So, each of them will contain vertices and links within themselves. In any machine belonging to the *valid set,* this nesting of *machines* inside *machines* can be to any depth.

Since the behavior of a part of the whole is similar to that of whole, and this property is exhibited by mathematical structures called fractals, the construct has been named fractals.

The final *machine* constructed is the one with the name *MainMc.* So, every program defines a *MainMc.* Note that the type of a vertex can also be a *machine* defined earlier. This allows us to define *machines* which are aggregates of earlier defined machines. Hence, we have the *MainMc* defined as shown in figure 3a.

Finally, it is the system's responsibility to construct the *virtual machine* named *MainMc* for the programmer on the available hardware. The size of the *virtual machine* is not specified at the time of writing the program. Hence, the system can look at the number of available resources at runtime and decide how big the *virtual machine* should be for that execution of the program.

By giving only minimal *virtual machine* and the way in which it can expand, the programmer is leaving the decision of the size of the actual *machine* to the run-time environment. This eliminates the need for the programmer to make assumptions about the hardware while writing the program.

3.3 The Program for the Virtual Machine

We now describe how the programs for the *virtual machine* are written. It consists of several different *programs,* each of which will be run as a separate process.

All programs are written in C. The syntax for a typical program is : *Program <prog-name>* { *<any-c-program>* } where <any-c-program> can be any program written in C. It should be possible to compile the program and run it as any other C program. An instance of such a program is shown in figure 3b.

The program shown in figure 3b contains primitives *send* and *receive* which are explained in the next section. But, essentially what the program does is that it receives some work from *main.* If the work is big enough, it divides it into three parts and sends it to the three workers. It then receives the results from the three workers, adds them and sends the result back to *main.* Similarly, we also have programs *worker and main.*

Each node in the virtual machine executes a *Process.* A process is an executing instance of a program. The definition of which process executes which program is given in the section ProcessDefinition. An example of such a definition is shown in figure 3c.

The processes *w1,w2, w3* are executing instances of the program *worker, ma* is an instance of the program *main* and so on.

The processes defined thus are mapped onto the nodes of the virtual machine using the *Map* construct. The mapping should map every vertex in the machine to a process. Every such mapping of the vertices of a machine to a set of processes is given a name by the map construct. For example, Map Tree(0,mgr),(1,w1),(2,w2),(3,w3) is TreeSet;

This name *TreeSet* is later used to refer to the set of processes on the machine *Tree*.

3.4 Communication Primitives

There are two basic communication primitives, *send* and *receive*.

Send is non-blocking and has the syntax *send(<process-name>, <msgtype>, <type-data>)* where <process-name> is the name of the receiving process and <msgtype> is an integer. <type-data> is a list of data items preceded by their types. For example, we can have *send (w1, 1, int, i, float, f)* where i and f are variables of type int and float respectively.

Receive is a blocking primitive i.e it does not return control until the corresponding send has been matched with it. Its syntax is similar to that of send, *receive(<process-name>, <msgtype>, <type-data>)*

Here, instead of variables, the addresses of the variables should be passed as parameters. The <process-name> can be the name of some process or can be ANY. If the name is ANY, it allows the receiver to wait for any message of the appropriate type, irrespective of the sender.

The message type allows receiver to distinguish between messages. A receive of a certain type can be matched only by a send of the same type (with one exception stated below). For example, a receive having a type field of 1 can be matched only by a send with type field 1. But, type 0 is a special case in that a receive of type 0 can be matched by a send of any type. For example, a receive might look like *receive(ANY, 0, int, &i, float, &j)*

The IN and OUT message queues

The <process-name> in *send* can only be a process which is executing on the same *virtual machine*. The reason is that there may be several identical *machines* with processes of the same names in them. So, if communication between processes on different *machines* were allowed, a process name could match with many processes, all on different *machines*.

For every *machine*, there are two special message-queues, IN and OUT. These can be passed to send and receive just as process names are sent. The purpose of these message queues are discussed below.

Referring to Figure 1, a process such as *mgr* at the highest level has three entities with which it can communicate, viz. w1,w2 and w3. In case the vertex 1 in that *machine* were a single node, then w1 would have been a worker process. Here, 1 is not a single node, but is a *machine Tree* by itself. Let us, for the purpose of discussion call this particular *machine* M.

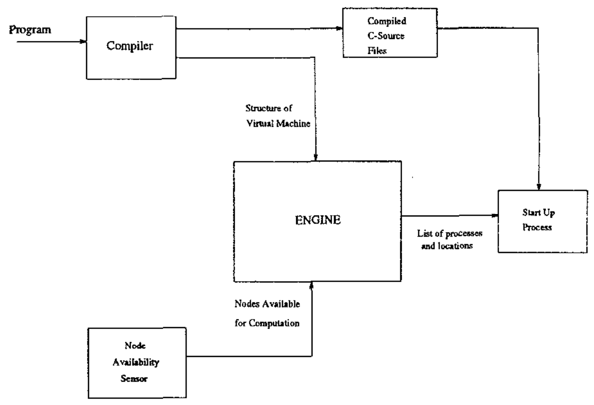
Program
Compiler
Compiled C-Source Files
Structure of Virtual Machine
ENGINE
List of processes and locations
Start Up Process
Nodes Available for Computation
Node Availability Sensor

Figure 2. Overview of the Implementation

So, w1 refers to the collection of processes in vertex 1 and M refers to the *machine* that is present on vertex 1. However, whether vertex 1 is a single node or M does not make any difference to the way mgr treats them. It stills sends a message to w1 just as if it were a single process.

When a message reaches w1 (which is not a single process), it is sent to the message-queue IN for M. Therefore, IN acts as a receiver for all the messages to the entity w1. All the processes inside M (those in the set represented by w1) will receive from IN just as if they were receiving from any other process internal to the *machine*.

Similarly, when mgr receives from the entity w1, it receives the contents of the other message-queue OUT. Any message that is to be sent out of M from a process inside it has to be sent to OUT. So, if mgr executes the statement *receive(w1, 3, float, value1)* then, it will get the contents of the OUT of M.

The processes inside M have to decide as to which message is sent to OUT to be taken by processes external to it. They can control the receiving process from OUT by using message tags.

The use of IN and OUT has made the interface between w1 and the entities outside it the same whether w1 is a single process or is a set of processes. The process mgr sends to w1 and receives from w1 in both cases.

Figure 3 contains a program written in Banyan which solves the problem of calculating an integral using the quadrature method.

4 Implementation

The language constructs have been implemented on a network of Sun 3/50s running Sun OS and supporting NFS, the network file system. The run-time environment is built on the PVM [9] platform.

(a) The Machnine Definition

```
NodeType N1type{
        Power 10;
}
NodeType N1type{
        Power 5;
}
LinkType L1type{
        Bandwidth 10;
}
Machine Tree{
       Vertices{
            N1type 0;
            N2Type 1,2,3;
        }
        Fractals{
            Tree 1,2,3;
        }
        Connections{
            L1Type (0,1);
            L2Type (0,2);
            L2Type (0,3);
        }
}
Machine MainMc{
        Vertices{
           N1type 0;
           Tree 1;
        }
        Connections{
          L1Type (0,1);
        }
}
```

(b) The Program Part.

```
Program Main{
    float left,right,value;
    /* Get input from user*/
    printf("Enter left and right");
    scanf("%d %d",&left,&right);
    send(TreeSet,1,float,left,float,right);
    receive(TreeSet,1,float,value);
    Printf("%d\n",value);
}
Program Manager{
    float left,right,m1,m2;
    float value1,value2,value3,value3,epsi;
    receive(IN,1,float,left,float,right);
    if (abs(left-right) >epsi)
    {
        m1=(2*left+right)/3;
        m2=(left+2*right)/3;
        send(w1,2,float,left,float,m1);
        send(w2,2,float,m1,float,m2);
        send(w3,2,float,m2,float,right);
        receive(w1,3,float,value1);
        receive(w2,3,float,value2);
        receive(w3,3,float,value3);
        result=value1+value2+value3;
        send(OUT,1,float,result);
    } else {
        result=Compute_Integral(left,right);
        send(OUT,1,float,result);
    }
```

```
Program Worker{
    float left,right,result;
    receive(mgr,2,float,left,float,right);
    result=Compute_Integral(left,right);
    send(mgr,3,float,result);
}
```

(c) The Process Node Mapping

```
ProcessDefinition{
    Main    :ma;
    Manager:mgr;
    Worker  :w1,w2,w3;
 }
Map MainMc{ (0,ma), (1,TreeSet)};
Map Tree{ (0,mgr),(1,w1),(2,w2),
              (3,w3) } is TreeSet;
```

Figure 3. A Complete Banyan Program

4.1 Compiler

The compiler separates out the C programs in the *program* constructs into separate files. It replaces the *send* and *receive* primitives by the PVM *snd* and *rcv* primitives. These are then compiled into separate PVM components. In addition, it extracts information about the virtual machine structure (both the minimal machine and how it can expand) and the processes that should run on them, and stores it in a file. These files are then used by the run-time system.

4.2 Run-time system

4.2.1 Node Availability Sensor

The first step in the execution of a program is to sense the load on each machine. This is done with the help of a daemon running on every machine. This also decides, based on the load information and information about the power of the nodes (which is obtained statically), whether to create more processes on that node or not. In the current implementation, a simple heuristic is followed. Machines with heavy loads are ignored and those with light loads are selected, irrespective of their power.

4.2.2 Engine to decide process distribution

An engine takes the following input :

1. The structure of the virtual machine and the name of the process running on each node. (the compiler output)
2. The set of *active* nodes and the load that they can still take (the output of the node-availability sensor).

It then decides the actual number of processes to create (the actual *process tree*) and on which node each process executes. The processes that communicate more often will be mapped on to a single node so that the communication overhead involved will be low. In the current implementation as we have NFS underlying, the runtime system need not worry about the file availability in order to run the processes on different nodes on the network. In the absence of such a system the runtime system must also take care of making the resources such as the files available for the program to run on the various nodes on the network. Thus, the runtime system must be aware of the underlying environment.

The engine creates a start-up program with all the information needed such as the processes on the various nodes on the network. The start-up program then places the processes on the appropriate nodes on the network. So, the size of the virtual machine constructed depends on the resources available at run-time.

4.3 Case Study

A template matching problem from graphics based on cross correlation was programmed in *Banyan*. The problem is computationally intensive and well suited to be programmed on a distributed system. A 64 * 64 template was matched on a 256 * 256 image. The time taken for the sequential code on a single machine was 524.01 secs. The parallel program when executed on a single machine used one manager process and three worker processes since the minimum machine has been defined as 1 manager node with 3 worker nodes and hence it took 885.4 secs. The program scales in units of 1 manager + 3 workers as specified by the programmer.

The results are tabulated below :

#Nodes	Time taken in Secs	Speedup
10	87.2	6.01
9	88.2	5.94
8	117.5	4.45
7	119.1	4.35
6	138.5	3.70
5	288.3	1.81
4	289.9	1.80
3	293.9	1.78
2	544.4	0.96
1	885.4	0.59

4.3.1 Discussion

It can be observed that the program scales well on the whole, the speedup obtained for 10 nodes being 6.01. The speedup for a single node was 0.59 , the overhead being due to PVM and due to the number of processes running on that node. However, the speedup does not increase uniformly. The reason for this is that when the machine scales, the unit of scale is not a single process, but a set of processes. If the number of nodes is increased gradually, the size of the virtual machine will not increase for the addition of every node, but will increase for the addition of every 3-4 nodes. So, one can see a jump in the speedup as we move in the sequence.

Another reason for variations in the speedup is the fact that the machine can be in an *unbalanced* state. For example, in figure 1, the machine is not balanced, in one of the machines, only vertex 1 has been expanded into a tree and not vertex 2 and 3. Though the result from processes on 1 might come faster, the manager on 0 cannot return the result until both 2 and 3 have replied. One solution to this problem is to either expand all the vertices 1,2 and 3 into *Tree*s or do not expand any of them. This is equivalent to reducing the set of valid machines. Such heuristics can always be used by the system in constructing the virtual machine. This benefit is derived from the fact that the construction of the virtual machine is completely left to the system.

5 Conclusion

We have introduced Banyan, a language in which programs can be written without making assumptions about the run-time environment. Programs written using Banyan are truly scalable with their run-time environment.

The actual algorithm for solving the problem is coded in C. The language could easily be modified so that the programs can be written in some other popular language. This has the advantage that programmers do not have to learn a new language for writing their algorithms.

Banyan ensures that there is a clear separation of system's responsibilities and programmer's responsibilities.

Typically computationally intensive problems such as Computation Fluid Dynamics (CFD) can be programmed to run effectively on a network of workstation environment.

References

[1] Thomas Anderson, David Culler, D.Patterson, *Case for NOW (Network of Workstations)* Technical report, University of California at Berkeley, 1994.

[2] Jiubin Ju, Gaochao Xu, Jie Tao, *Parallel Computing Using Idle Workstations*, SIGOPS : 12th ACM Symposium on OS principles, 1988, pp. 87-96.

[3] M. Ferhan Pekergin *Parallel Computing Optimization in the Apollo Domain Network*, IEEE Transactions on Software Engineering, Vol 18, No 4, April 1992, pp. 296-303.

[4] Singh Jaswinder Pal , John L. Hanessey and Anoop Gupta, *Scaling Parallel Programs for Multiprocessors: Methodology and examples*, IEEE Computer July 1993 pp. 42-50.

[5] Rushikesh.K.Joshi, D.Janaki Ram *Parset : A language construct for system independent parallel programming on distributed systems*, Microprocessors and Microprogramming, 1995, Vol 41, pp. 245-259.

[6] Sudhir Ahuja, Nicholas,C., and D.Gerelenter, *Linda and friends*, IEEE Computer, August 1986, pp. 26-34.

[7] Henri E.Bal, M.Frans Kaashoek and Andrew S.Tanenbaum, *Orca : A language for Parallel programming of Distributed Systems* IEEE Transactions on Software Engineering, March 1993, Vol 18 No. 3, pp. 190-205.

[8] Joshi Rushikesh.K, D.Janaki Ram, *Object Based Subcontracting for parallel programming on loosely coupled distributed systems*, Journal of programming languages, June 1996.

[9] V.S.Sunderam, *PVM : A framework for parallel distributed computing*, Concurrency : Practice and Experience, December 1990, pp. 315-338.