Configurable Shared Virtual Memory for Parallel Computing

Myungchul Yoon   Miroslaw Malek

Department of Electrical and Computer Engineering
The University of Texas at Austin
yoon@pine.ece.utexas.edu

July 15 1994

Abstract

Most distributed-shared memory (DSM) systems use a global address space which is shared by all processing nodes. Although the global address space is convenient in providing a single address space for programming models, it is applicable to DSM systems with relatively small number of processing nodes. As the system size grows, there is a problem with scalability. To avoid this drawback, a new shared virtual memory management scheme, "CSVM (Configurable Shared Virtual Memory)," is proposed. CSVM creates a new single address space for a given program which is shared only by the nodes cooperating for the parallel execution of the program. The new scheme, therefore, can be applied to any size of DSM system and even to heterogeneous DSM systems. We have developed two algorithms for the distribution and the management of a shared address space. These algorithms cover both the virtual memory of sequential computer and the shared virtual memory of the traditional DSM system, which means CSVM is reduced to the virtual memory management of sequential computer in executing a sequential program and to the traditional shared virtual memory in executing a parallel program. According to the analytical result, CSVM reduces the message traffic of DSM system with global address very effectively.

Keywords : *Shared virtual memory, Distributed shared memory, Memory management, Parallel computer architecture.*

1 Introduction

Building a teraflop supercomputer is ongoing challenge [1]. With technological advances pushed to physical limits, architecture and parallelism receive the most promising alternatives to pursue. Large part of most programs consists of subprocesses which either do the same work over different data or are mutually independent, so that concurrent execution of these subprocesses by multiple processors can greatly reduce the overall completion time of the programs. Employing parallelism in general problems, however, remains to be a challenge, though SIMD (single instruction-stream multiple data-stream) type architecture has succeeded in solving special type of problems.

Message passing and distributed shared memory (DSM) are two major paradigms in building general purpose parallel computers. Message-passing system is easily scalable because no inter-dependent element exists in its model. In the message-passing programming model, cooperating processes exchange their information by messages through appropriate inter-process communication protocols. But the absence of global data object in this model is inconsistent with the existing programming styles. Shared-memory programming model provides transparent data communication which meets with the conventional programming style and which is familiar to users. Although shared-memory multiprocessor system is suitable to this model, it is not scalable due to the access contentions of the shared memory. DSM systems release this problem of shared-memory systems by distributing the main memory to local processors and logically sharing these local memories. They use shared memory programming model on top of message passing system to achieve both programmability and scalability. Since the speed of internode communication is the key for maintaining the memory consistency and reducing the remote memory access delay, most of DSM systems use system-specific interconnection network such as hypercube, mesh or torus to increase the performance of the systems.

Recently, some DSM systems have been built through the general purpose network such as Ethernet. [9, 10, 11] The network-based parallel computer (NPC) has many merits in comparison to the traditional DSM systems. By using the general network, it can scale up easily, support the heterogeneous architecture, exploit directly the development of the network technology and of the commercial computer technology, and so on [9]. Other works on parallel computing in heterogeneous environments [12, 13] show the promise of such an architecture.

If we reflect on the evolution of computer networking technology of sequential computers, we can easily imagine the future of parallel computing. Like today's computer network, all computers in the future would be connected by high speed interconnection networks with ultrafast interface and we could use our PC or workstation as a processing node to execute our programs in parallel with other processing nodes in remote locations via the networks. Such a large scale, networked multicomputer system would consist of commercial sequential computers and would work as a general purpose parallel computer. Supporting a single address space for such a large system is one of the difficult problems to be solved.

Most DSM systems [4, 5, 6, 7, 10] use a global address space which is shared by all processing nodes to support the single address space for programs. But according to our research, such a global address space is inadequate for the scalable, general purpose parallel computers, especially for NPC. In this paper, we suggest a virtual memory management scheme that eliminates the disadvantages of global address space. Our method extends the virtual memory technique of conventional computers to shared virtual memory [3] of parallel computers so that each computing node can run both sequential and parallel programs with the same virtual memory technique. Each single address space for a program is created and destroyed at runtime and shared only by the processors cooperating for the same job. Two algorithms, the static and the dynamic algorithm, are also presented for the distributed management of the shared virtual address space.

Based on the object-oriented approach, Forin and his colleagues proposed a user-level shared memory server [13] working under Mach operating system. In their approach, the user can create shared memory objects and can map the objects to certain ranges of task's address space. These objects are managed by user-defined processes, called external pagers, and can be accessed by the processes via an asynchronous message interface between the pager and the kernel. [13] Our approach is similar to theirs in that the shared memory spaces are dynamically created at runtime but is different in that all address spaces of tasks (jobs) in our method are shared memory spaces and the creation and the management of the shared spaces, of which many parts of the functions can be implemented by hardware, are totally transparent to the users.

In the next section, we point out the disadvantages of a global address space. The new shared virtual memory scheme is presented in Section 3, and two algorithms for shared address space assignment are proposed in Section 4. In Section 5 we show the merit of our method in reducing the message traffic in the system.

2 Drawbacks of the Global Address Space

Most DSM systems use a global address space of which the management is distributed to all processing nodes. Application programs running at the same time in such systems use disjoint subspaces of the global address space so that every program runs in a single address space without interfering with each other. With regard to the systems with small number of processing nodes, the global address space might be convenient for the implementation and the management of the distributed memories, but it imposes many disadvantages on the system with a large number of processing nodes.

In this section, we describe the possible drawbacks of global address space in building

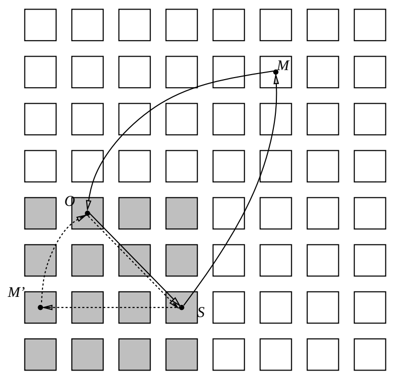


Figure 1: Detour of remote memory access in a DSM system. An application program is running on 16 nodes out of 64 nodes. A page fault of *S* is forwarded to the owner (*O*) via respective default managers: (a) default manager in global address space (*M*); (b) default manager in configurable shared virtual space (*M'*).

a scalable parallel computer. Hereafter, we use the term *global address space* as a predefined single address space shared by all PE's in the system, and *single address space* as a single address space given to a program which is shared by all or some PE's in the system. In addition, in the rest of this paper, NPC is used to refer to the large scale network-based multicomputer system which is the connection of a huge number of (heterogeneous) computers through the general purpose networks.

2.1 Increment of Message Traffic and Remote Memory Access Delay

The management of global address space could be centralized or distributed to several nodes and the manager(s) might be fixed or changed dynamically. Among the schemes proposed by K. Li [3], the "distributed fixed manager algorithm" is a widely used method for the management of global address space due to its simplicity and low runtime overhead. In this algorithm every node takes charge of fixed parts of the global address space. To access a page in remote locations, a node first gets the information about the owner of that page from the node (default manager) which is responsible for the management of the page.

This widely used algorithm has the redundancy of memory-sharing if it is used in global address space. General purpose parallel computer has to deal with a variety of applications. Due to the diversity of application programs, some programs use most of processing nodes of the system while others use only small parts of them. For example, let us consider a DSM system composed of 64 (8×8) PE's which are connected by a high speed interconnection network (Fig. 1). Each box in the Fig. 1 represents a processing node. In the DSM system with a global address space (GDSM), each PE in the system is in charge of a part of the global address (real or virtual) space whether or not it is involved in the parallel execution of a program. Note that not all programs use all the PE's for their execution because of the limit of their parallelism. Lots of application programs use smaller number of PE's than those supported by the system. If a program requires 16 PE's for its execution, it runs only on some parts of the system (for example, only the shaded processors in the Fig. 1), but its address space is shared by all the 64 PE's rather than these 16 PE's. Owing to this redundancy, memory references are not always directed to one of the cooperating PE's, which, on the average, takes longer path.

Fig. 1 shows the possible scenario of a detoured remote memory access. If a page fault is generated by a PE, *S*, *S* sends a message to the default manager of the page (*M*) to request a copy of that page. If *M* is the owner of the page, it sends a copy of the page to *S*. If the page is owned by another PE, *M* forwards the message to the owner (*O*) and *O* sends the copy to *S*. If the address space of the program is shared only by the PE group which consists of the cooperating PE's only, all memory references generated during the execution are confined to the group. In this case, there is other default manager (*M'*) in the group and the resulting memory access path may be shorter than that of GDSM. These longer message paths in GDSM result in higher message traffic density and higher delays in remote memory access.

This effect is more serious if the number of cooperating PE's is much fewer than the total number of PE's of the system. The analysis on this effect on the several network models is presented in Section 5.

2.2 Scalability Issue

As we mentioned before, the general purpose parallel computer has to run efficiently for both the highly parallelized program and the nearly sequential program. The highly parallelized programs favor the system having large number of PE's with distributed memories, whereas the programs with low parallelism prefer small number of high performance processors with large local memory. The ideal system, therefore, would be one that can support sufficient number of processors to maximize the parallelism and that has plenty of fast local memories to maximize the performance of each processor. But the global address space is an obstacle in achieving this goal.

To illustrate this point, let us try to scale up the DSM system from Fig. 1. If we define N as the number of PE's of the system, and m_i, V_i as the main memory size and the size of address space of *i-th* PE, respectively. The size of global address space, V_g, is determined by

$$V_g = \min_{1 \le i \le N} \{V_i\} \tag{1}$$

since any location in global address space has to be addressable by every node of the system.

The total memory size of the system is restricted such that

$$V_g \geq \sum_{i=1}^{N} m_i \tag{2}$$

or if the system is homogeneous,

$$V_g \geq Nm \tag{3}$$

In scaling up the system, we encounter the point where the total memory size is bigger than the size of global address space ($Nm > V_g$). The system is easily scalable only up to $N_c (= \lfloor V_g/m \rfloor)$ PE's. If we want to scale up the system over N_c PE's, we have to make a decision whether to use other processors with larger address space or to decrease the size of local memory of each node. Changing processors requires the total redesigning of the system which takes a lot of time and results in large overhead in scaling up. Cutting the local memory size results in performance degradation of each processor. It has been well known that the more powerful a processor is, the more memory it requires to maximize its capability. The performance degradation is more serious when each processor works in multiprogramming environment. Consequently, GDSM has the limited scalability up to N_c without much overhead.

It may be argued that using the recently developed 64 bit processors solves the problem, since the address space size of 2^{64} is virtually unlimited, which may be true if it is a homogeneous system. On the contrary, if it is a heterogeneous system the situation is not so simple and much more complex in NPC. The next section describes the rising problems when the global address space is used in NPC.

2.3 Application Problems for Network-based Parallel Computers

The ultimate goal of parallel computer system might be the networked multicomputer system where computers are distributed over wide areas and of which the high speed networks take the role of today's system-specific interconnection network. Because such a system consists of the mixture of heterogeneous computers, applying global address to it may cause several problems. Considering the situations that could take place in such a large heterogeneous system, we conclude that the global address space is inadequate to NPC for several reasons.

First, the maximum size of the global address space is determined by the processor of the smallest number of address bits (Eq. 1). If it is greater than virtual address size of a processor, the processor is excluded in memory-sharing or can access the limited parts of shared memory, which means that the address space is no more global. As the system size grows, it runs bigger programs or more programs simultaneously. But the global address space size of NPC does not increase as large as the system size by this reason. Thus, the global address space is saturated easily due to the demand of huge number of programs to be run simultaneously in NPC.

Second, the total memory size of the system may easily exceed the size of global address space due to the huge number of PE's and their local memories. Furthermore, the local memory sizes are widely different, so that neither the distribution of address space nor the translation of virtual addresses is simple.

Third, the system parameters, such as the number of processing nodes and the size of local memories, vary often. As we experience in today's computer network, the total number of computers connected to the network keeps changing, and so do the local memories. Even worse is that the effect of turning on/off of a computer is equivalent to attaching/detaching it to the network. A special mechanism has to be devised to handle such situations.

Fourth, the system with global address space is very susceptible to faults. This is true of both specially-designed DSM and NPC. As the system size grows, the chance of faults also increases. In the global address space, any fault of a node may cause the malfunction of the system. For example, if the PE, M, in Fig. 1 is faulty, the program could not be run properly even though all cooperating PE's are working correctly. Besides real faults, a lot of pseudo-faults (transient faults) are possible in NPC such as turning off, rebooting, etc.

Finally, the number of cooperating processors for the execution of a program is usually much smaller than the total number of processors in NPC. The detoured memory accesses increase the message traffic density of NPC more significantly than the small GDSM.

All the drawbacks discussed so far come from the globality of address space. To build the scalable general purpose parallel computer, we need to propose a solution which eliminates this globality.

3 Configurable Shared Virtual Memory (CSVM)

The single address space for a program is essential for the ease of programming. By the reasons presented in the previous section, application of global address space is effective to relatively small number of processing nodes. We propose a new method for the management of shared virtual addresses which can be applied to any kind of distributed memory system including NPC.

For the convenience' sake, we define some terminology used in this paper first. We use the *task* as the basic unit of parallel processing, therefore, a parallel program is divided into a set of tasks. We define a *job* as the execution of a program and the *requester* as the processor (node) which initiates a job and distributes the tasks of the job over the system. The processor scheduled to run at least one of the tasks of a program is defined as a *member* of the program. All members form the *group* of the program.

Our method provides a single address space to a program which is shared only by the members of the program. No predefined shared memory is required. Each shared virtual memory

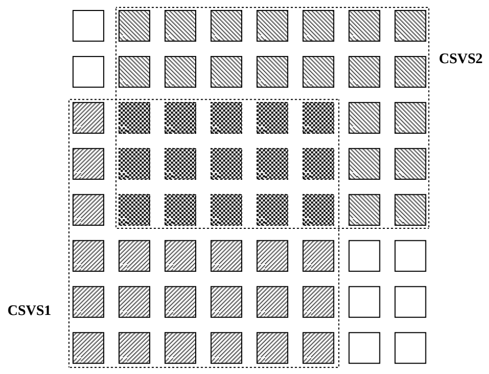
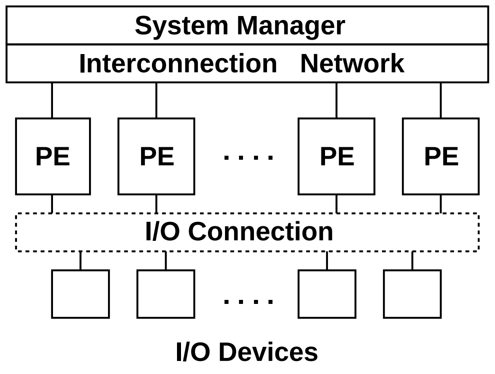


Figure 2: Example of CSVM. At the instant, two programs are running in parallel on the system and two configurable shared virtual address spaces (CSVS) distributed only to the member nodes are created to support the single address space for each program.

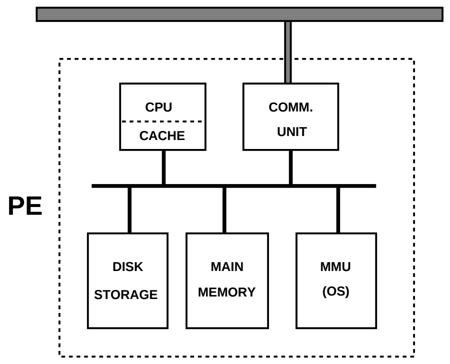
is created for a program at runtime and destroyed at its completion. All these operations and the management of the shared space are transparent to users. This makes it possible for each node to be designed and used as an independent system just like a PC or a workstation connected to a network. Each node serves as a processing node for parallel execution only when it is a member of the group of a given program. Of course, a node can be a member of several groups if it is working in multiprogramming environment. Fig. 2 shows an example of the system. Unlike GDSM systems, each differently colored (shaded) shared virtual memory is shared only by the member nodes.

3.1 The Model and Operation

The support system for our model consists of three components, *processing nodes, network and system manager,* and *I/O devices* (Fig. 3-(a)). Each processing node is virtually a conventional general purpose computer which is composed of a processor, main memory, disk storage, OS and memory management unit (MMU), and communication unit (CU) (Fig. 3-(b)). A processor and a main memory are essential for a node, but other facilities could be shared with other nodes. Each node of a DSM system can be a node in this system. In this case, the interconnection network of the DSM system forms a substructure of the network in the system. The whole DSM system can be a node as well, since it satisfies the requirement for a node. The memory hierarchy of each node is the same as that of the sequential computer except that the remote memories are at the same level of hierarchy with its local disk storage as in Fig. 4. All page faults, of which the virtual page addresses lie in its subspace, are handled between the main memory and its disk storage, while those in other subspaces are resolved between the main memory and the memories of the other members.



(a)



(b)

Figure 3: A support system for CSVM. (a) System architecture (b) Basic components of each node.

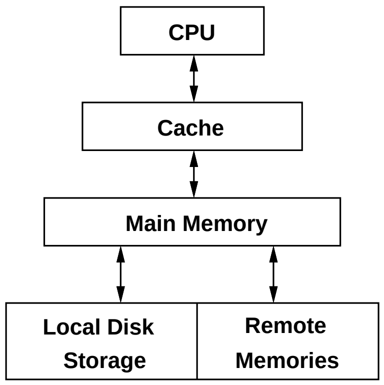


Figure 4: Memory hierarchy of each node in a CSVM system.

The network supports the point-to-point communication for any pair of nodes using message passing protocol, and the system manager is responsible for reliable and deadlock-free communication, system diagnosis, and system administration. I/O devices are distributed over the system and some of them may belong to a specific node or be shared by several nodes, in which cases other nodes can access the devices through the intermediation of the owner node.

In addition to this model, we assume the following for the operation of CSVM system.

- Each processing node has the full capability of a conventional computer. In the rest of this paper, we assume, without the loss of generality, that it is a general purpose uniprocessor system (Fig. 3-(b)).
- Each processor (processing node) has an identification number which is unique in the

network.

- There is a communication protocol which is agreed by all the nodes, so that any pair of node can exchange messages correctly.
- There is a mechanism which distributes the tasks to all or parts of the processing nodes. This could be centralized to the system manager or distributed to each node depending on the system size and operating system.
- A program can be divided by a set of tasks at compile time.
- As an implicit assumption, the speed of network is tolerable for executing programs in parallel.

In the absence of the execution of parallel program, CSVM system is just like the today's computer network or multicomputer system. Each node of the system does its own jobs with its own facilities, and its OS or MMU manages its memory and disk storage in its own way. There is no preassigned shared memory. The shared memory space is created in need of a program, rather than predefined. This space exists only during the execution and is destroyed with the end of the program. All of the operation is transparent to the user.

The preparation for the creation of a configurable shared virtual address space (CSVS) begins when the requester receives a job. As soon as the requester receives the job, it generates a job-identification number (*job_id*) for the program which has to be unique over the entire system. Next, the requester prepares the information table in its memory under the *job_id* (Fig. 5).

```
JOB_INFORMATION {
        status;
        number_of_tasks ;
        number_of_scheduled_tasks ;
        number_of_completed_tasks ;
        *member_list ;                 /* pointer to the first member */
        number_of_members ;
        IO_server ;
}
```

Figure 5: Structure of *JOB_INFORMATION* table.

The *status* has one of three values, *after, proceeding,* and *before*, which indicates that the CSVS has already been created, is being created, or has not been created, respectively.

The entries, *number_of_scheduled_tasks, number_of_completed_tasks* and *number_of_members* are initialized by 0, and the *number_of_tasks* by the value given at compile time. The entries *number_of_members* and *member_list* will be completed through the task distribution (see Section 4). The *IO_server* is initialized by the requester's processor identification number (*requester_id*) since it is the only processor which knows and must know all the I/O information. Some of the entries may not be used depending on the address space assignment algorithm (Section 4).

After the preparation of the table, all the tasks of the program are tagged by the job_id and the requester_id and then distributed over the system. According to the distribution, the members are determined and so is the CSVS. The requester may or may not be a member of the program depending on the distribution. Even if the distribution is completed, the corresponding CSVS is not created until the execution of the first task of the job. Under our model, every node runs two auxiliary procedures before and after each task, the *Head()* procedure and the *Tail()* procedure. The major role of the *Head()* procedure is checking the existence of the virtual address space under the job_id. If it has been already created, it executes the task immediately; otherwise, it creates the new CSVS under the job_id according to an algorithm (see Section 4). The role of the *Tail()* procedure is checking whether the task is the last of the program. If it is the last task, the node informs other members of the termination of the CSVS.

The creation of the CSVS is initiated by the member who runs the first task of the program. Before the execution of the first task, the member requests the creation of new CSVS to all other members (or via the requester). As soon as a member receives the request, it creates the assigned subspace under the job_id. The creation and management of the subspace are entirely up to each member so that it is possible to use the conventional virtual memory technique with a little modification. In addition to conventional virtual memory method, OS or MMU of the each member keeps a copy of *JOB_INFORMATION* table of the CSVS. This is used in finding the default manager of a page when page faults occur. We use the "distributed fixed manager algorithm" [3] for the management of CSVS such that each member undertakes the management of fixed parts of the CSVS.

If a page fault occurs, MMU locates the default manager of this page. If the page is in its subspace, MMU handles the fault with the conventional technique – a victim is selected from main memory and swapped with the page in its local disk storage. Only the pages in its subspace can be stored in its disk storage. If the selected page belongs to others, it is discarded or returned to its manager depending on its status. On the other hand, if the virtual page address of the fault is out of its subspace, the virtual address, job_id, and the default manager_id are sent to CU to generate the message requesting a copy of that page. All remote memory accesses in CSVM are carried on by sending the messages including both the job_id and the virtual address. The job_id acts as the password to access the memory locations, so that no other processor can access the remote memory locations without the job_id. With the job_id, the virtual address is translated into physical address at the final destination node.

3.2 Advantages of the CSVM

Though DSM system intends to combine the scalability of message-passing system and the programmability of shared memory system, GDSM partially succeeds in exploiting the scalability of message-passing system. CSVM is the system that can fully use the merits of both systems. It has the following advantages compared with the GDSM.

Node Independence Each node in CSVM could be designed and used independently. This property makes it easy to build and maintain the system. A node can be removed, added and upgraded easily. The improvement of network technology and sequential computer technology will be directly employed in the system.

Fault Tolerance It is a fault tolerant architecture for large DSM systems. As we explained in Section 2.3, one fault in GDSM systems affects the operation of the entire system. The chance of faults increases in proportion to the system size. To build a scalable parallel computer, we must take the fault tolerance of the system into consideration for the reliable operation. CSVM needs a diagnosis mechanism for this goal, while GDSM needs not only a diagnosis mechanism but also a reconfiguration algorithm. If CSVM detects a faulty node, it excludes the node in task distribution until the fault is fixed. This action is sufficient for the proper operation of the system, because the faulty node is automatically excluded in the share of any CSVS in the system. Any number of faulty nodes can be masked by this method as long as the faults do not break the communication network.

Reduction of Message Traffic Our model provides a degree of freedom in reducing the message traffic and remote memory access delay of the system. All messages for the program executions are confined in the group in CSVM, so that scheduling (distributing) the tasks to relatively closed nodes reduces the overall message traffic in the system (Section 5). But the situation is not so simple in GDSM, since the messages can be sent to any node in the system, we cannot be assured that such distribution of tasks will reduce the message traffic of GDSM.

Size Scalability The size of CSVM system is virtually unlimited. CSVM can support any number of nodes as well as heterogeneous nodes. The scale up of CSVM is very easy and cost effective. Unless the problem size does not exceed the addressability of the system, we need not change individual nodes to scale up the system.

Problem Scalability The largest problem size which CSVM can process is determined by the maximum of addressabilities of its nodes. If the system is homogeneous, the addressability of a node determines the maximum problem size. If it is a heterogeneous system, a node in CSVM can receive jobs of which the size is larger than its addressability, since the requester node need not be a member. If there are nodes whose addressability is bigger than the job size, the requester distributes the job to these nodes only. This option meets the goal of parallel computing in the future – a PC can ask the execution of large program to NPC built on WAN (wide area network). The PC supports the I/O files for the tasks and receives the final result. The speed of interconnection network is the determinant factor for its efficiency.

Unified Model The membership strategy in sharing the virtual address space covers from the virtual memory of sequential computer to that of DSM. If a sequential program is executed in CSVM or all the tasks of a program are assigned to a node, the node never shares the CSVS with any other node and, its virtual memory management is, therefore, the same as the virtual memory technique of sequential computers. If CSVM runs highly parallelized program so that the tasks are distributed to all nodes, its virtual memory management is that of GDSM. Thus, this system is a unified model for the virtual memory management of the sequential computer and the parallel computer.

In addition, the memory hierarchy and the virtual memory management are the extension of those of sequential computer for parallel computing. Without parallel computing, they are merely working the same as the conventional computer. Combining all of the merits, CSVM is a generalization of conventional virtual memory technique for both the sequential computer and the scalable parallel computer.

4 Address Space Assignment

CSVS is divided into a number of subspaces, which are distributed to the members. The partitioning method and the distribution method are very important because they affect the communication pattern during the execution of the job. In this section, we present two assignment (partitioning and distribution) algorithms. Their applicability depends on the availability of the complete member list before the execution of the job. Designing of the optimal algorithm for reducing the message traffic or for reducing the overhead of the system requires knowledge of runtime behavior of all application programs which is not possible in general. Our algorithms, therefore, are not optimal but are designed with a focus on the simplicity of hashing mechanism, the balanced distribution of the address space, and overhead minimization.

4.1 Static Assignment Algorithm

This algorithm can be used only if the complete information of two entries of the *JOB_INFORMATION* table, the *number_of_members* and the *member_list*, are available before the beginning of the first task, that is, before the creation of the CSVS. In this algorithm, all members create and destroy their subspaces at the same time. Since we know all the information by the creation of the CSVS, we can partition the CSVS and distribute the subspaces to M members in an arbitrary manner. The simplest method may be to assign the space equally to all members. The hashing of a page address into a subspace is done by simple function

$$H = (p/c) \bmod M \tag{4}$$

where p is a virtual page address and c is a natural number [3]. The processor_id of the default manager is obtained from *member_list[H]*. The order of members in the *member_list*, therefore, determines the assignment of the subspaces. It can be rearranged for optimization before the creation of corresponding CSVS.

Fig. 6 is the flowchart of the static assignment algorithm. The Head() procedure searches CSVS TABLE in MMU to match the *job_id* before the execution of each task. If the CSVS with the *job_id* exists, it checks the *status* of the CSVS. The task starts immediately if the *status* is *after*; otherwise the task is held until the *status* will be set to *after*. If the Head() procedure fails to match the *job_id* in CSVS TABLE, it sends the message, *request_new_CSVS*, to the requester. Upon receiving the message, the requester sets the *status* of *JOB_INFORMATION* to *proceeding*, and sends the copy of *JOB_INFORMATION* table with the message requesting the creation of the assigned subspace (*make_subspace*) to all nodes in *member_list*(If the *status* is *proceeding* already, the requester discards the received message). The processors receiving the request message from the requester save copy of the *JOB_INFORMATION* in CSVS TABLE and create their subspace in the same way as the usual virtual address technique. After finishing the creation of subspace, each member informs the requester of its completion (*subspace_created*). When the requester has received the *subspace_created* messages from all the members, it sends message, *creation_completed* to all members to set their copy of *status* to *after*.

After the completion of each task, each member sends the *task_completed* message to the requester. Whenever requester receives the message, it decreases the value of *number_of_tasks* and then checks the resulting value. If it is 0, the requester informs all members of the termination of the CSVS (*terminate_CSVS*) and removes the *JOB_INFORMATION* table, while the members destroy their subspaces.

The simplicity and ease of implementation is the advantage of this algorithm. It adds a little overhead to the virtual memory management of sequential computer at the creation of CSVS. After the creation, the overhead for checking the existence of the CSVS is negligible. The

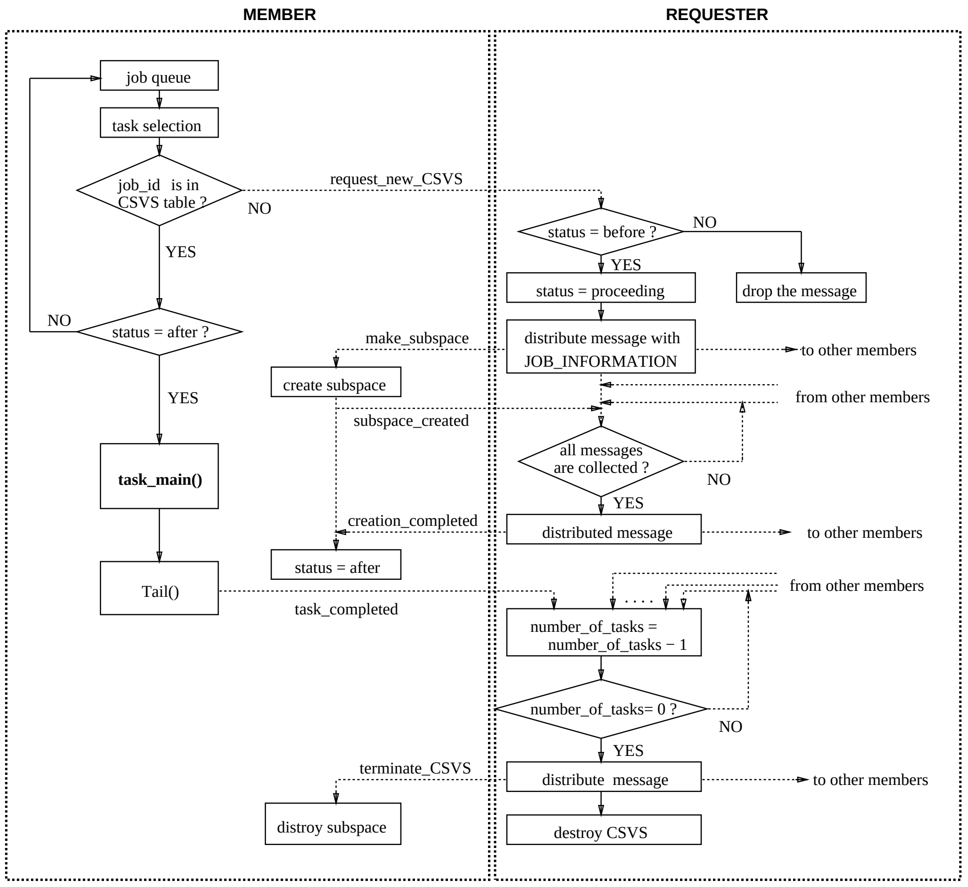


Figure 6: Flowchart of the static assignment algorithm. Solid arrows stand for the immediate flows and the dotted arrows represent the messages.

system with the centralized scheduler may be suitable for this algorithm, since the information on *member_list* can be easily provided by the scheduler. The major disadvantage of this algorithm is the requirement of prior-knowledge of all the members which may prevent it from using in some applications.

4.2 Dynamic Assignment Algorithm

The application of the static assignment algorithm is restricted by the availability of the member list at the beginning of a given program. The dynamic assignment algorithm presented here gets rid of this requirement at the expense of runtime overhead.

Note that, even though all members execute at least one task of the program, they do not start their execution at the same time. For example, let us consider a program composed of eight tasks of which the dependence graph is presented in Fig. 7-(a). At the beginning of T1,

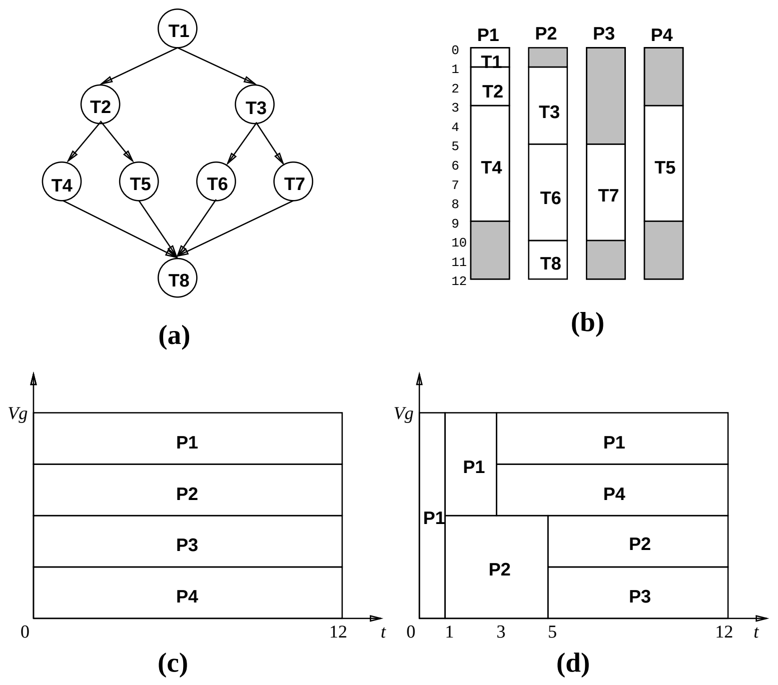


Figure 7: Comparison of two address assignment algorithm. (a) Dependency graph of a program. (b) Scheduling of the tasks. (c) Address space assignment of static assignment algorithm. (d) Address space assignment of dynamic assignment algorithm. A subspace is bipartitioned and one of them is assigned to new member.

only P1 is the known member and the other tasks may not have been scheduled yet. This situation could occur to enhance the dynamic scheduling, load balancing, the task migration, etc. Assuming that no other task is scheduled at the beginning of T1, P1 has to take charge of the whole CSVS, because it is the only known member at that instant without knowing the prospective members. P2 joins the group at the beginning of T3 (Fig. 7-(b)), and manages parts of the CSVS (T3 may be scheduled earlier than the end of T1, but it need not share the CSVS before T3 starts. Actually T1 may run more efficiently when P1 manages all the space than when it shares the CSVS with P2). Similarly, P3 and P4 join the group at the beginnings of T7 and T5.

In the dynamic algorithm, each member shares the CSVS from the beginning of its first task. Whenever a new member joins the group, the CSVS is rearranged and the information of the result has to be distributed to all members. This algorithm has the cons and pros–the large runtime overhead and the efficiency in transient time. Pages have to be moved from the disk of old default manager to that of new default manager whenever a new member joins the group, which introduces large overhead. On the other hand, some tasks may run more efficiently with

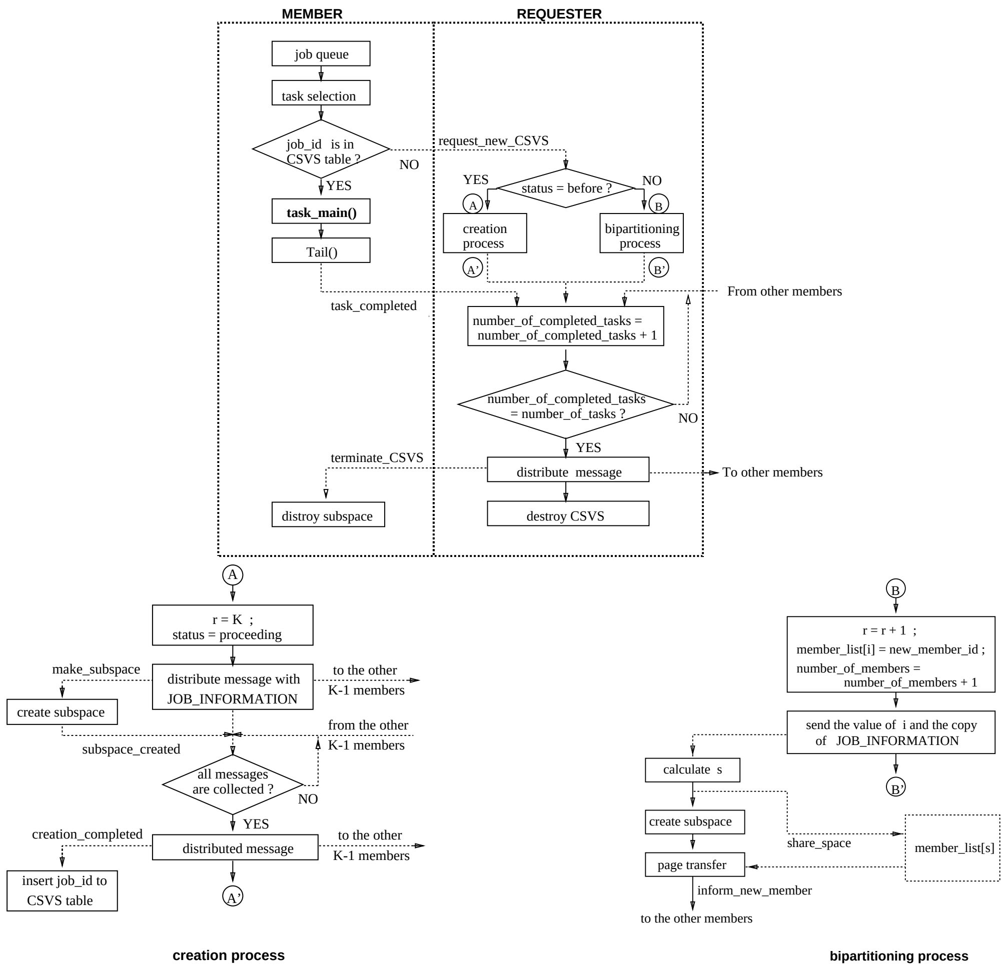


Figure 8: Flowchart of dynamic assignment algorithm. Solid arrows stand for the immediate flows and the dotted arrows represent the messages.

this strategy, that is the T1, T2, and T3 run in the CSVS which is shared by smaller number of members.

Reducing the runtime overhead is the key for the efficient algorithm. Intuitively, equi-assignment algorithm with the hashing function like Eq. (4) is not feasible due to the large overhead for rearrangement. We designed the “address space bipartition algorithm” to reduce the overhead. In our algorithm, only two members, one existing member and the new member, are involved in the rearrangement process.

Fig. 8 is the flowchart of our algorithm. As in the static assignment algorithm, Head() procedure looks at the CSVS TABLE, and sends *request_new_CSVS* message to the *requester* if desired *job_id* is not found. If the corresponding CSVS has not been created (*status=before*), it is created with the partially filled *member_list*. If only K out of M members are known at that time, this algorithm creates the CSVS with these K members by the same way in the static assignment algorithm. On the contrary, if the CSVS exists (*status=proceeding*), the requester adds the new member's *processor_id* in *member_list*, increases the *number_of_members*, and sends the value of index r with a copy of *JOB_INFORMATION* table to the new member. Then, the new member selects one existing member from the *member_list* (*member_list[s]*) and takes half of its subspace. The index, s is calculated by the following equation.

$$s = r - K\ 2^{\lfloor \log_2 \frac{r-1}{K} \rfloor} \tag{5}$$

The selected member divides its subspace into two sub-subspaces of equal size and transfers all the required information about one of them to the new member. After finishing the rearrangement, the new member informs all other members of its affiliation to make them modify their copy of *JOB_INFORMATION* table and the hashing function. The hashing function for locating the default manager of a virtual page address is given as

$$H = \begin{cases} (p/c) \bmod L & \text{if} \quad (p/c) \bmod L \le T \\ (p/c) \bmod L - S & \text{if} \quad (p/c) \bmod L > T \end{cases} \tag{6}$$

with

$$L = K\ 2^{\lceil \log_2 \frac{r}{K} \rceil} \tag{7}$$

$$S = K\ 2^{\lfloor \log_2 \frac{r}{K} \rfloor} \tag{8}$$

$$T = r \bmod L \tag{9}$$

The values L, S and T are calculated and stored in MMU whenever the new member joins. The member chosen for the rearrangement is responsible for the forwarding of transient messages to the transferred space which have been sent to it before the notification but which arrive after the notification.

Fig. 7-(c), (d) shows an example of the address space assignment of two algorithms. While it is fixed in static assignment algorithms, there is a transient period in dynamic algorithm till all members join the group. Note that, if all members are known at the beginning of the program ($K = M$), this algorithm would be the same as the static assignment algorithm in the previous section. The dynamic algorithm is general and has advantages of dynamic scheduling, and task migration, but the increased runtime overhead is a major disadvantage of this algorithm.

5 Message Traffic Analysis

In this section, we analyze the possible savings in message traffic density of traditional DSM system by using CSVM instead of global address space. The message traffic density (ρ) generated during the execution of a job (program) is defined as

$$\rho \equiv \frac{\sum_{k=1}^{m} s_k d_k}{l} \tag{10}$$

where m is the number of messages, s the message size, d the traveling distance of the message in terms of the number of communication links, and l the number of communication links in the system.

To compare the effect of CSVM on message traffic density, let us run a program on a CSVM system and a GDSM system. Except the distribution of address space, every condition comprising the distribution of task, architecture, network structure etc. is the same in both systems.

We clarify the messages into two categories, the direct and indirect message. The direct message is the one that its path does not depend on the address space distribution, while that of indirect message does. For example, like a control message or a message for cache coherence in the directory-based cache coherence architecture [7, 6], the direct message goes to a definite processor so that its path is the same both in CSVM and in GDSM. On the other hand, the indirect message such as message for remote memory reference, goes to different processor according to address space management scheme. Although the final destination of the message might be the same, two systems route the message through different paths by message detouring as discussed in Section 2.1.

By separating the message types, the Eq. (10) becomes

$$\begin{aligned} \rho &= \frac{\sum_{k=1}^{m_d} s_k d_k + \sum_{k=1}^{m_i} s_k d_k}{l} && (11) \\ &= \frac{\bar{s}_d \sum_{k=1}^{m_d} d_k + \bar{s}_i \sum_{k=1}^{m_i} d_k}{l} && (12) \\ &= \frac{m_d \bar{s}_d \bar{d}_d + m_i \bar{s}_i \bar{d}_i}{l} && (13) \end{aligned}$$

where the subscript d and i stand for the direct message and indirect message, respectively, and the $\bar{s}$ and $\bar{d}$ represent the average size and the average distance of messages. In the derivation of Eq. (12) from Eq. (11) we replace the individual message size by the average size of all messages.

The ratio of the message traffic density of CSVM to that of GDSM is

$$R_\rho = \frac{\rho_C}{\rho_G} = \frac{(m_d \bar{s}_d \bar{d}_d + m_i \bar{s}_i \bar{d}_i)_C}{(m_d \bar{s}_d \bar{d}_d + m_i \bar{s}_i \bar{d}_i)_G} \tag{14}$$

Since the program is executed in the same environment in both systems except the address space distribution, the messages generated by the systems and their size are almost the same (The messages generated for the creation of CSVS are not included and are treated as overhead). In addition, the average distances of direct messages of the two systems are the same since those messages are independent of the distribution of address space. Considering this fact, we get the formula

$$R_\rho = \frac{1 + R_m R_s \left(\frac{\bar{d}_{i_C}}{\bar{d}_{d_G}}\right)}{1 + R_m R_s \left(\frac{\bar{d}_{i_G}}{\bar{d}_{d_G}}\right)} \tag{15}$$

where R_m, R_s are the ratio of the number of messages and the ratio of the average message size of two types which are defined as $R_m = m_i/m_d$ and $R_s = \bar{s}_i/\bar{s}_d$.

Calculation of the average distance is difficult because it depends not only on the program behavior but also on the task distribution on the system. To remedy the difficulties, we make the following assumptions.

1. The network structure is symmetric for all nodes.
2. Memory accesses are uniformly distributed across the address space, and direct messages are uniformly sent to all members as well.
3. The tasks of the program are distributed on the PE's which are as close as possible to one another.

Most interconnection network architectures used today such as bus, mesh, torus, hypercube etc. satisfy the Assumption 1 very well, even though some of them introduce "edge effect" which violates the symmetry for edge nodes. We ignore the edge effect in this paper.

The uniform access assumption (Assumption 2) simplifies the analysis greatly. Under this assumption, the probability $p(d)$ that a node sends a message to a node at the distance d is proportional to the number of nodes at the distance d from the node. Hence, the average distance will be

$$\bar{d} = \frac{1}{m}\sum_{i=0}^{m} d_i \tag{16}$$

$$= \sum_{d=0}^{d_{max}} d\, p(d) \tag{17}$$

$$= \frac{1}{N}\sum_{d=0}^{d_{max}} d\, n(d) \tag{18}$$

so that we get

$$\bar{d}_{i_G} = \frac{1}{N_i} \sum_{d=0}^{d_{max}} d\, n_i(d) = \frac{1}{N} \sum_{d=0}^{d_{max}} d\, n(d) \tag{19}$$

$$\bar{d}_{d_G} = \frac{1}{N_d} \sum_{d=0}^{d_{max}} d\, n_d(d) = \frac{1}{M} \sum_{d=0}^{d_{max}} d\, n_d(d) \tag{20}$$

where N and M are the number of nodes in the system and the number of member. $n(d)$ is the number of nodes at the distance d in the system, and $n_d(d)$ is the number of nodes at the distance d in the subsystem which consists of only member PE's and all communication links of the system. The distribution function $n(d)$ and $n_d(d)$ satisfy the relation.

$$\sum_{d=0}^{d_{max}} n(d) = N \tag{21}$$

$$\sum_{d=0}^{d_{max}} n_d(d) = M \tag{22}$$

Note that under the Assumption 1, $\bar{d}_{i_C} = \bar{d}_{d_G}$. By substituting Eq. (19), Eq. (20) to Eq. (15), we get the final expression for R_ρ.

$$R_\rho = \frac{1 + R_m R_s}{1 + R_m R_s R_d} \tag{23}$$

with

$$R_d = \frac{M}{N} \frac{\sum_{d=0}^{d_{max}} d\, n(d)}{\sum_{d=0}^{d_{max}} d\, n_d(d)} \tag{24}$$

The Eq. (23) shows that the merit of CSVM vs. GDSM with respect to message traffic density is the function of the ratios of the number of messages, average message size and average distance. R_d is the major factor which determines the merit or demerit of CSVM on message traffic. R_d can have the values greater than, equal to, or less than 1, depending on the distribution of tasks. Actually, if the tasks are distributed to the processors that are separated far apart, the message traffic of CSVM increases compared with that of GSDM. In most cases, however, the system scheduler distributes the tasks to relatively close processors for efficient inter-processor communication, so that CSVM can reduce the message traffic.

While $n(d)$ is a fixed value determined by network structure, $n_d(d)$ is the function not only of the network structure, but also of the distribution of tasks. Under the Assumption 3, we derive R_ρ as the explicit function of the ratio M/N. Assumption 3 eases the dependency of $n_d(d)$ on task distribution, so it is used only for the convenience of calculations.

Under the Assumption 3, Eq. (24) becomes

$$R_d \simeq \frac{M}{N} \frac{\sum_{d=0}^{d_{max}} d\, n(d)}{\sum_{d=0}^{l} d\; n(d)} \tag{25}$$

where l is the smallest integer satisfying

$$\sum_{d=0}^{l} n(d) \geq M \tag{26}$$

Straightforward calculations using the Eq. (21), (23), (25) and (26), and approximating the summations of the equations by integrations, give the explicit form of the value R_ρ. We derived the explicit form of R_ρ, for three distribution functions.

1. Uniform distribution : $n(d) = \frac{N}{d_{max}}$, $\qquad 0 \leq d \leq d_{max}$.

$$R_\rho = \frac{1 + R_m R_s}{1 + R_m R_s \left(\frac{M}{N}\right)^{-1}} \tag{27}$$

2. Triangular distribution as Fig. 9.

$$R_\rho = \begin{cases} \frac{1+R_m R_s}{1+R_m R_s \left(\frac{8}{9}\frac{M}{N}\right)^{-\frac{1}{2}}} & \text{if } 0 \leq \frac{M}{N} \leq \frac{1}{2} \\ \frac{1+R_m R_s}{1+R_m R_s \frac{M}{N}\left[\frac{2\sqrt{2}}{3}\left(1-\frac{M}{N}\right)^{\frac{3}{2}}+2\frac{M}{N}-1\right]^{-1}} & \text{if } \frac{1}{2} < \frac{M}{N} \leq 1 \end{cases} \tag{28}$$

3. Polynomial distribution : $n(d) = a\ d^n$, $\qquad 0 \leq d \leq d_{max}$, $\qquad a$ is a constant.

$$R_\rho = \frac{1 + R_m R_s}{1 + R_m R_s \left(\frac{M}{N}\right)^{-\frac{1}{n+1}}} \tag{29}$$

Most of DSM systems use system-specific network architecture like mesh, torus, hypercube, ring and so on. Many of these architectures roughly follow the triangular distribution. As shown in Fig. 9, the triangular distribution of $n(d)$ is a good approximation for finite size k-dimensional mesh or torus architecture, and roughly for low-dimensional ($k \leq 10$) hypercubes. One level ring

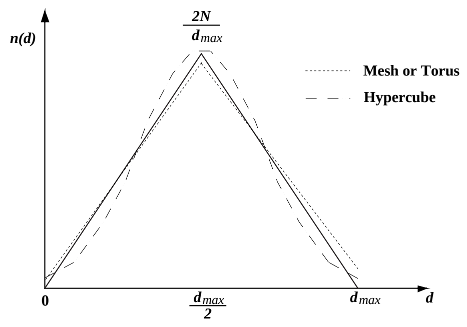


Figure 9: Triangular distribution of $n(d)$. Mesh, torus and hypercube approximately follow this distribution.

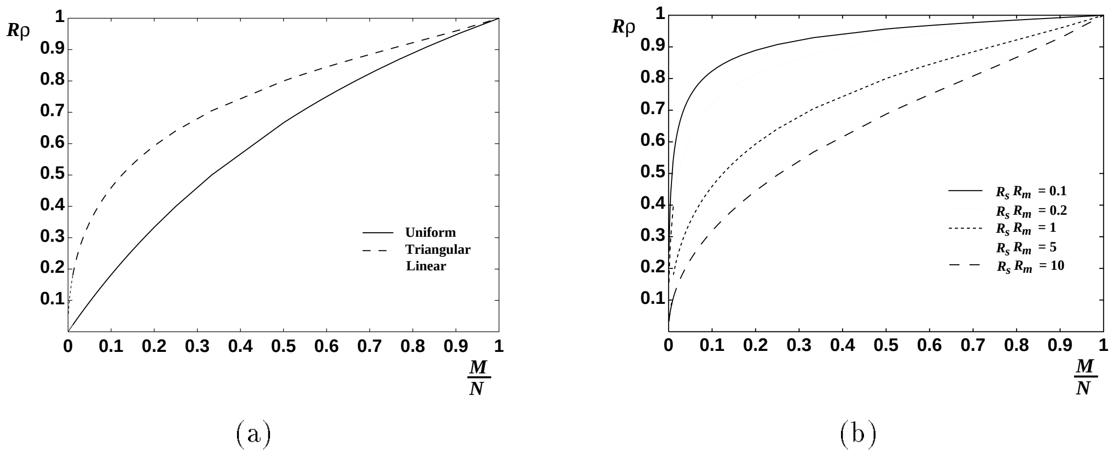


Figure 10: CSVM to GSDM ratio of message traffic density. (a) Comparison of the three distribution functions with $R_sR_m = 1$. (b) The effect of the value R_sR_m on the message traffic reduction of CSVM for the triangular distribution.

architecture is an example of the uniform distribution, but we hardly find a system with this type of distribution in a real system. It is, however, helpful in comparing the effectiveness of CSVM in reducing the message traffic. The polynomial distribution is used for the approximation of the network in NPC. If NPC is built in WAN and the density of computer (node/unit area) is uniform in the area, the number of computers at the distance d from a computer is proportional to the distance. If we assume that the number of links between two nodes is proportional to the physical distance, the distribution $n(d)$ is approximately a linear function of the distance.

The Fig. 10-(a) shows the effectiveness of CSVM in reducing the total message traffic of the system. It shows that CSVM is equally effective for both specially designed DSM systems and NPC with linear distribution. If a program is running on the half of the nodes of the system, CSVM saves about 20% of message traffic of GDSM. It reduces more message traffic as the ratio

M/N is getting smaller. For the large DSM systems, we expect to reduce the message traffic by more than 50% because the number of members will be much smaller than the number of nodes ($M/N < \frac{1}{10}$).

The Fig. 10-(b) shows the effect on R_ρ of the size and ratios of direct and indirect messages ($R_s R_m$). As we can expect from the Eq. (28), R_ρ becomes a function of M/N alone, if $R_s R_m \gg 1$. If $R_s R_m \ll 1$, the advantage of CSVM is apparent only when $M/N < \frac{1}{10}$.

6 Concluding Remarks

A new configurable shared virtual memory scheme, which facilitates scalability, is proposed. CSVM supports shared memory spaces for the collection of independent computers so that it can be applied to any size of distributed memory system and network-based parallel computers. According to the analytical result, CSVM is very effective in reducing the message traffic compared to the conventional DSM system.

References

[1] G. Bell, "Ultracomputer: A Teraflop Before Its Time," *Commun. ACM,* 35(8) pp. 27-47, 1992.

[2] B. Nitzberg and V. Lo, "Distributed Shared Memory: A Survey of Issues and Algorithms," *IEEE Computer,* 24(8), pp. 52-60, 1991.

[3] K. Li and P. Hudak, "Memory Coherence in Shared Virtual Memory Systems," *ACM Trans. Computer Systems*, Vol. 7, No. 4 pp. 321-359, Nov. 1989.

[4] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing," *Proc. Int'l Conf. Parallel Processing*, pp. 94-101, 1988.

[5] R. Bisiani and M. Ravishankar, "Plus: A Distributed Shared-Memory System," *Proc. 17th Int'l Symp. Computer Architecture* pp. 115-124, 1990.

[6] D. Chaiken, J. Kubiatowicz, and A. Agarwal, "LimitLESS Directories: A Scalable Cache Coherence Scheme,", *Proc. Fourth Int'l Conf. Architectural Support for Programming Languages and Operating Systems,* ACM, pp. 224-234, 1991.

[7] D. Lenoski, et al, "The Stanford Dash Multiprocessor," *IEEE Computer,* pp. 63-79, Mar. 1992.

[8] A. J. Smith, "Cache Memories", ACM Computing Surveys 14(3), pp. 473-530, Sep. 1982.

[9] H. T. Kung, et al, "Network-based Multicomputers : An Emerging parallel architecture," *Proceedings Supercomputing '91i*, pp. 664-673, 1991.

[10] R. G. Minnich and D. V. Pryor, "Mether: Supporting the Shared Memory Model on Computing Clusters," *IEEE COMPCON Spring '93*, pp. 558-567 1993.

[11] Akira Jinsaki, "A Fast Distributed Shared Virtual Memory System: NET-VMS," *Fujitsu Scientific and Technical Jounal,* 29, 3, pp. 286-295 Sep., 1993.

[12] S. Zhou, M. Stumm, K. Li, and D. Wortman, "Heterogeneous Distributed Shared Memory," *IEEE Trans. Parallel and Distributed Systems,* vol. 3 pp. 540-554, 1992.

[13] A. Forin, J. Barrera and R. Sanzi, "The Shared Memory Server," *USENIX winter* pp. 229-243, 1989.