

Modeling Distributed Data Representation and its Effect on Parallel Data Accesses^{*}

Dejiang Jin and Sotirios G. Ziavras

Department of Electrical and Computer Engineering

New Jersey Institute of Technology

Newark, NJ 07102, USA

Address for Correspondence

Professor Sotirios G. Ziavras

Department of Electrical and Computer Engineering

New Jersey Institute of Technology

Newark, NJ 07102, USA

ziavras@adm.njit.edu

Tel: (973) 596-5651

Fax: (973) 596-5680

^{*} This work was supported in part by the U.S. Department of Energy under grant DE-FG02-03CH11171.

Abstract

PC clusters have emerged as viable alternatives for high-performance, low-cost computing. In such an environment, sharing data among processes is essential. Accessing the shared data, however, may often stall parallel executing threads. We propose a novel data representation scheme where an application data entity can be incarnated into a set of objects that are distributed in the cluster. The runtime support system manages the incarnated objects and data access is possible only via an appropriate interface. This distributed data representation facilitates parallel accesses for updates. Thus, tasks are subject to few limitations and application programs can harness high degrees of parallelism. Our PC cluster experiments prove the effectiveness of our approach.

Keywords: data encapsulation, distributed data, parallel data access, PC cluster, super-programming model.

1. Introduction

PC clusters have emerged as viable alternatives for high-performance, low-cost parallel computing [1]. However, they have weak communication capabilities that require substantial software support. Thus, the development of parallel programs faces major challenges. Data sharing among processes is essential in parallel programs [14] and the chosen techniques may affect performance. PC clusters distribute computation tasks and application data among the nodes. To efficiently utilize the distributed resources, workload balancing is applied. Since computation tasks ultimately need to access locally represented data, data representations can affect the effectiveness of task assignment. Computation threads access shared data in the order dictated by the program. Some threads may be stalled until preceding accesses have completed [3]. The long communication latencies in PC clusters increase significantly the penalty of thread stalls. There are two fundamental models for data sharing in parallel programming, message passing [14,16] and shared memory [8,13,14]. Object-oriented programming models may encapsulate both mechanisms [7,10]. However, there are substantial limitations when combining the former models for shared data representation.

In this paper we propose a novel distributed data representation scheme primarily for our *Super-Programming Model (SPM)* introduced in [4]; the implementation of a relevant runtime support system is discussed as well. SPM integrates both message passing and shared memory. Under SPM, an effective instruction-set architecture (ISA) is to be developed for each application domain. Frequently used operations in that domain should belong to this ISA. The *Super-Instructions (SIs)* in the ISA are to be developed efficiently in the form of program functions for individual PCs in the cluster. The sizes of the operands (i.e., function parameters) for SIs are limited by predefined thresholds. Application programs are modeled as *Super-Programs (SPs)* coded with SIs. Under SPM the parallel system is modeled as a virtual machine (VM) which is composed of a single super-processor that includes an instruction fetch/dispatch unit (IDU) and multiple instruction execution units (IEUs). For PC clusters, an IEU is a process running on a member node that provides a context to execute SIs. SIs are dynamically assigned to IEUs based on a producer-consumer protocol. The IDU assigns SIs, from a list of SIs ready to execute, to IEUs as soon as the latter become available. The super-processor can handle a set of “build-in” data types called *Super-Data Blocks (SDBs)*. All application data are stored in SDBs. SDBs are accessed through appropriate interfaces. The runtime support system provides the local

representation of SDBs. Each data entity can be incarnated into a set of objects distributed throughout the cluster. The SPM runtime support system controls the incarnated objects during their lifecycle based on their usage. Thus, the logic of scheduling and distributing tasks is decoupled from the data distribution and the impact of the latter on workload balancing is reduced dramatically. The aforementioned set of incarnated objects represent a coherent data entity. They cooperate with each other with the mediation of the runtime system. Such multiple distributed representations can serve efficiently the demands for data of multiple SIs. Our approach increases the number of SIs executing in parallel by minimizing thread stalling.

We introduce in Section 2 our data representation scheme under SPM and the corresponding data access model. In Section 3, we examine the effect of our distributed data objects on update operations by implementing an SP for matrix multiplication. Section 4 presents a theoretical analysis. In Section 5, a comparison with existing programming models is presented.

2. Distributed Data Representation and Data Access Operations

For a given application program under SPM, all data are entities existing in its global logical space. At runtime any data entity is represented by one or more incarnated data objects and/or data file(s) in external storage. These objects exist in the context of member processes which are distributed among the system's nodes and can be executed in parallel. Accessing a logical data entity is achieved by accessing one of its incarnated data objects, either locally or remotely through an interface. For data coherence, all accesses are controlled by the runtime support system. It maintains information about the state of all application data and grants access privileges as needed. The state diagram applied by the runtime support system individually to each data entity is shown in Fig. 1. A state change can be triggered by various events, such as an SI issue or commitment. During the process of changing the data state, the runtime support system may incarnate a new object, notify relevant incarnated objects to adjust their internal contents or persist/move data to external files. Only the entire set of relevant objects can represent the corresponding data entity completely and correctly; an individual object may contain partial data.

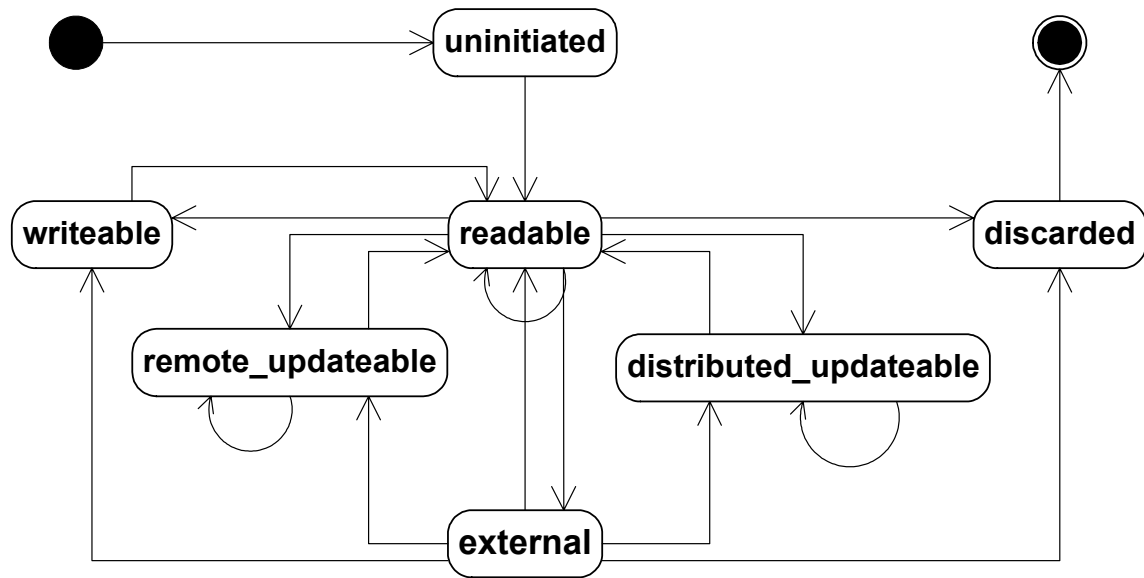


Fig. 1. State transition diagram for logical data entities.

More specifically, data can be in any of the following states during their lifecycle.

1. **Uninitiated:** the actual data does not exist and, therefore, cannot be incarnated. Thus, no access is allowed. There is just an ID that can be used to create “readable” data.
2. **External:** the data resides in external storage and is not represented by any incarnated object. Data cannot be accessed directly by application programs.
3. **Readable:** the data is incarnated. There exists at least one incarnated object (called the *master object*) associated with this data and there may be more incarnated objects (called *slave objects*); each incarnated object resides in the context of an IEU. The IEU that holds the master incarnated object is called the *host of the data*. All clients of this data, which are generally SIs in SFs, can get the content of this data but they cannot modify it. These objects may have different internal formats but they contain identical information for the same data. An SI can get complete information of the data from a local incarnated object. If an incarnated object does not exist locally, a slave object will be created by the runtime system. Maintaining data consistency is the responsibility of the runtime support system. Master data objects may be floating around IEUs as deemed appropriate by the runtime support system in attempts to improve the overall performance. It is important to differentiate between the master and slave objects. If the runtime system needs to create another slave object, it can always turn to the host of the data to create a clone of the master object. If it needs to free

space in the cache, it can discard any slave object; however, it cannot discard a master object before it transfers this title to a slave object or copies the data into external storage and changes the data state into “external.”

4. **Writeable:** the data is incarnated in an IEU. Unlike “readable” data, there exists only a single incarnated object that a local SI can access exclusively. The local interface enables both read and write operations by the SI. Once an SI gets the access privilege for “writeable” data, no other SI will be issued for this data. The only event that can change the state is commitment of this SI that sends the data into the “readable” state. Therefore, there is no arrow shown that comes back to the “writeable” state in Fig. 1.
5. **Remote_updateable:** similar to “writeable” data, there exists only one incarnated object. However, data can be updated by either local or remote SIs. If the incarnated object does not exist in the process context that the respective SIs are executing, data can be accessed through a specified remote access interface provided by the runtime support system.
6. **Distributed_updateable:** similar to “readable” data, a data entity is incarnated in the form of objects distributed among IEUs. However, individual incarnated objects may contain partial information. Only the collection of all of its incarnated objects can completely represent a logical data entity. SIs can modify data through their local incarnated objects. If there is not a local incarnated object, the runtime system can create one on demand. This state differs from the “remote_updateable” state in one more way. A synchronization operation is needed to go to the “readable” state after all distributed update accesses have been completed.
7. **Discarded:** this is the final state of logical data in their lifetime. In this state the logical data entity is no longer accessible and its incarnated objects are no longer needed. The runtime support system can recycle the resources previously allocated to these incarnated objects.

When an SI is issued or committed, a state change is normally triggered for used data. For a particular SI, however, one of its operands may not change state. For example, when an entity is “readable,” issuing an SI reading this data will keep the data in the “readable” state. This is shown by a one-state loop in the state diagram. This is also true for the “remote_updateable” and “distributed_updateable” states. An SI that wants to access data for an update or distributed update computes and passes a modifier for data merging through an interface provided by the runtime system. Modifier computation is separated from data operation. The real incarnated

object is encapsulated under the interface and may be hosted locally or remotely. Thus, SIs are freed from hosting data objects.

At runtime the system grants data access privilege to an SI on demand based on the state of the requested data. The state is kept unchanged until the SI commits. The SI may be granted a “read,” “write,” “update” or “distributed_update” privilege when the corresponding data is in the “readable,” “writeable,” “remote_updateable” or “distributed_updateable” state, respectively. If the accessed data is in the required state or can change to that state, the SI will be issued; otherwise, it will not. Only one kind of access privilege is granted for given data at any moment. Since no SI accesses data in the “external” state, a change to another state will occur if needed. The “write” privilege can be granted exclusively to a single SI. A privilege for the other three types of access can be shared by multiple SIs. When the SI commits (i.e. finishes execution), the system revokes its privileges and either keeps the data state unchanged or changes it to “readable.” Thus, there is no state change among “writeable,” “remote_updateable” and “distributed_updateable.”

3. Experimental Results

To demonstrate the effect of our approach, we have implemented the proposed runtime support system and carried out experiments. Our experiments were performed on a PC cluster with 8 dual-processor nodes. Each node has two 1.2 GHz AMD Athlon processors, 1GB of memory, 64KB level-1 cache and 256KB level-2 cache. The PCs run Red Hat 9 and are connected via a full-duplex 100Mbps Ethernet switch. An NFS shared file system is implemented.

3.1. Experimental Setup

We implemented a runtime support environment for our SPM model. A process is launched on each PC to implement an IEU [4]. Once an SI is assigned to an IEU, based on the meta data of the SI and its operands the runtime system prepares incarnated objects, picks up a computation method for realizing the SI and then launches a thread to execute the SI. Threads are scheduled by the local operating system. The type of data access for an operand is included in the meta data. SIs are not allowed to communicate with other SIs or to access external storage. All required data become available to them before they begin execution. The runtime

environment and the SIs were implemented in the Java language. An IEU can handle multiple SIs simultaneously, up to a predefined limit. The actual number of SI threads running on each node was monitored. Information was logged for SI data access types, start and stop times, and execution times. Statistical data were extracted from the logged information. Wall clock time was used for the total execution time.

We used matrix multiplication in our experiments. The SPs were developed using the SFs, SIs and SDBs described by us in [5]. The matrices are represented as collections of SDBs, where an SDB contains a sub-matrix block of limited size. An SI multiplies a pair of SDBs. We adopted the conventional matrix multiplication algorithm for the SF. To produce uneven workloads for the multiplication of subblocks, large sparse matrices were chosen. The algorithm for the implementation of an SI is normally determined at runtime. However, in our experiments we always choose an algorithm that can exploit the sparsity of SDBs for better performance.

To study the effect on the performance of distributed data representation and parallel data accesses, we created several versions of the SP. They differ in the way that SIs update blocks in the result. In the “SimpleUpdate” version, SIs assume the “writeable” state for resulting SDBs. In “RemoteUpdate,” SIs assume the “remote_updateable” state for resulting SDBs. In “DistributedUpdate,” SIs assume the “distributed_updateable” state for these SDBs. In the “Mixture” version, SIs assume the “writeable” state until computation begins for the last SDB in the resulting matrix; the remaining SIs assume the “distributed_updateable” state. Two scheduling policies were used to distribute SIs based on task groups [5]. All SIs in a task group contribute to the computation of the same SDB in the resulting matrix. In the Basic Scheduling Policy (BSS) that was used with all SPs, when an IEU becomes available then the IDU tries to issue an SI that belongs to the same task group with previously issued SIs to the same IEU. If the IDU cannot find such a task, then it picks up the next task from an untouched task group. If no untouched group exists, no SI is issued in “SimpleUpdate.” In “RemoteUpdate,” “DistributedUpdate” and “Mixture,” the IDU tries to issue an SI from another unfinished task group. The Smart Scheduling Policy (SSP) was used only with “SimpleUpdate” and “Mixture.” SSP and BSS differ in the selection of an untouched task group; SSP considers the history of the destination IEU in attempts to reuse already cached data [5].

We used a set of random sparse matrices in our experiments. They are irregular with 12.5% non-zero elements. Their properties are listed in Table 1. Multiplications are carried out

for non-zero elements only. The SP versions applied to these matrices are shown in Table 2. S, R, D and M stand for the “SimpleUpdate,” “RemoteUpdate,” “DistributedUpdate” and “Mixture” SPs, respectively, with BSS. G and N represent the “SimpleUpdate” and “Mixture” SPs, respectively, with SSP. We employed 1, 2, 4, 6 and 8 nodes in the experiments. The maximum number of SIs assigned simultaneously to an IEU was six, with an exception for the 256r1 case that varies this limit from one to eight; this approach was used to confirm that our chosen number of threads is reasonable.

Table 1. Matrix properties.

| Matrix name | Size | Number of blocks | SDB size |
|--------------------|-------------|-------------------------|-----------------|
| M11 | 4096x4096 | 32x32 | 128x128 |
| M12 | 1024x16384 | 8x128 | 128x128 |
| M13 | 16384x1024 | 128x 8 | 128x128 |
| M22 | 4096x4096 | 16x16 | 256x256 |
| M23 | 1024x16384 | 4x64 | 256x256 |
| M24 | 16384x1024 | 64x 4 | 256x256 |
| M25 | 2048x32768 | 8x128 | 256x256 |
| M26 | 32768x2048 | 128x8 | 256x256 |

Table 2. Experimental setup for matrix multiplication.

| Case name | Operands | Simultaneous threads limit | SP version |
|------------------|-----------------|-----------------------------------|-------------------|
| 256r1 | M25, M26 | 1, 4, 6, 8 | S, D, M, G, N |
| 256r2 | M23, M24 | 6 | S, R, D |
| 128r | M12, M13 | 6 | S, R, D |
| 256s | M22, M22 | 6 | S, R, D |
| 128s | M11, M11 | 6 | S, R, D |

3.2. Results

The net CPU time to execute an SI that multiplies a pair of sub-matrix blocks was measured by running the SI 100-1000 times with several operands preloaded in the memory using a separate pilot program, and averaging the resulting times. The time is 1.0ms and 18.5ms for SDBs of size 128x128 and 256x256, respectively. 256x256 SDBs take longer time because of two reasons. Completely caching these larger SDBs in the level-1 cache of a node is impossible. Also, the spatial locality of used data is low in sparse matrices for accessing indirectly non-zero elements. The speedup of R and D relative to that of S is shown in Fig. 2 (S is normalized to 1). The average number of simultaneous threads is shown in Fig. 3 and represents the actual degree of parallelism in the SPs. The local cache miss ratio is shown in Fig. 4.

For the 256r1 case, we varied the limit on the number of concurrent threads from one to eight. The execution time of the SPs and the actual degree of parallelism are shown in Fig. 5 and 6, respectively. After executing a set of SIs that access the same data for “distributed_updates,” a synchronization operation is needed to merge the distributed information into a master SDB before changing the data state to “readable.” The average synchronization time under various conditions is shown in Fig.7.

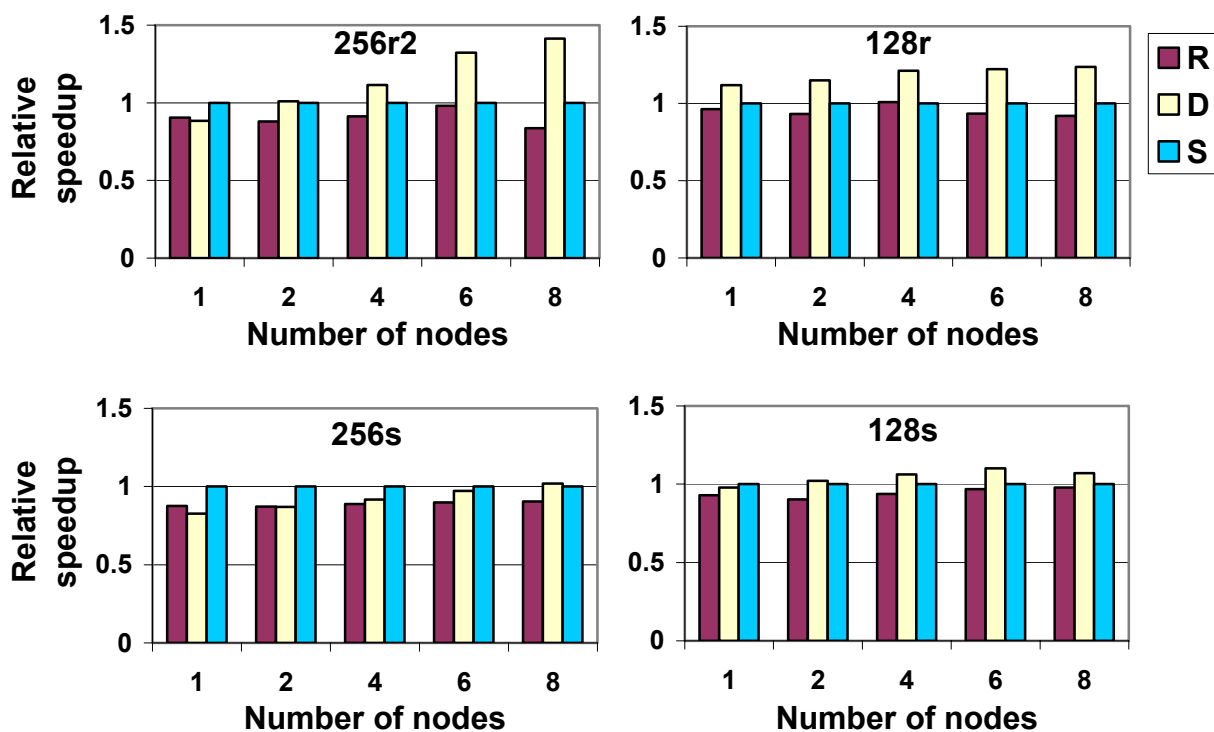


Fig. 2. Performance of R and D relative to S for various data writing methods.

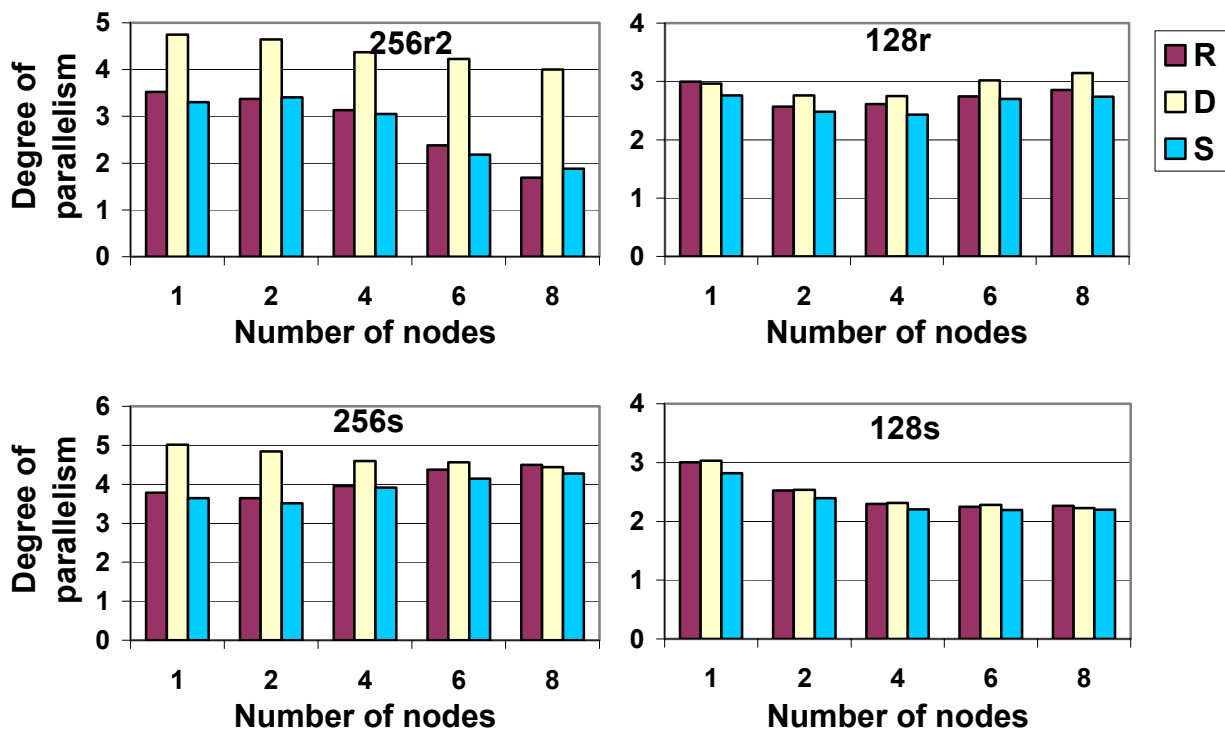


Fig. 3. The effect of the data access method on the actual degree of parallelism.

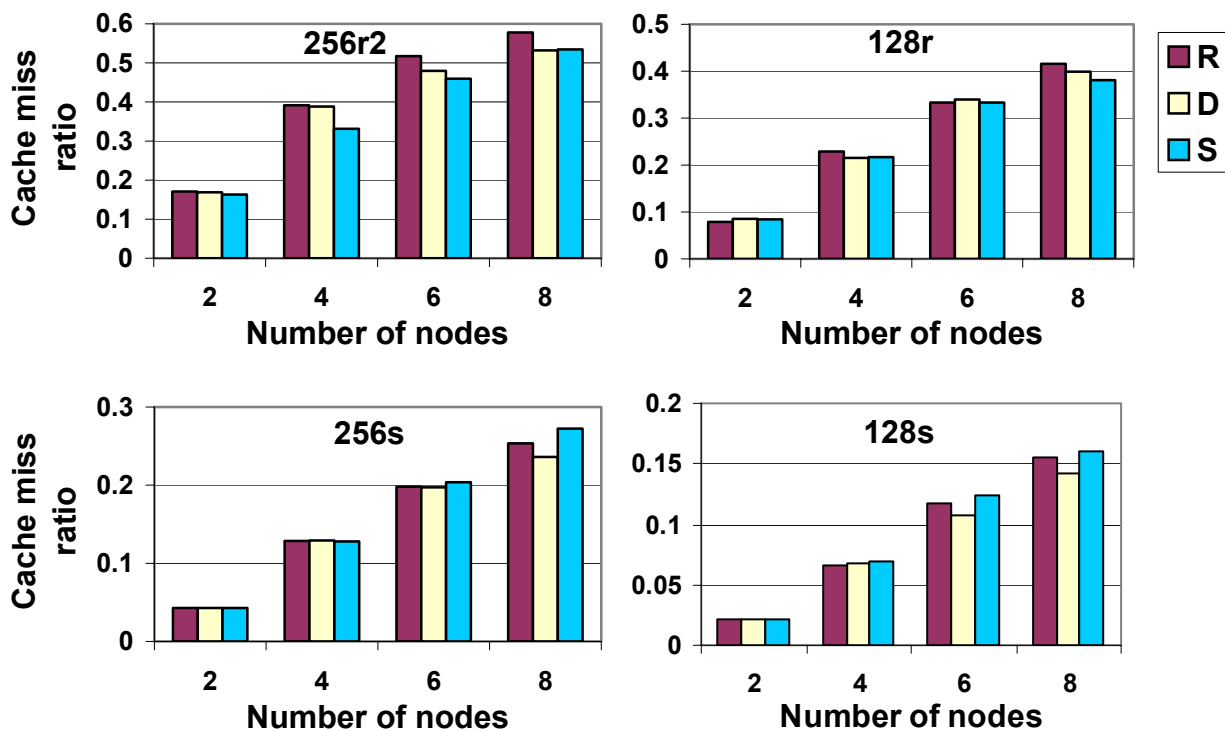


Fig. 4. The local cache miss ratio for data accesses.

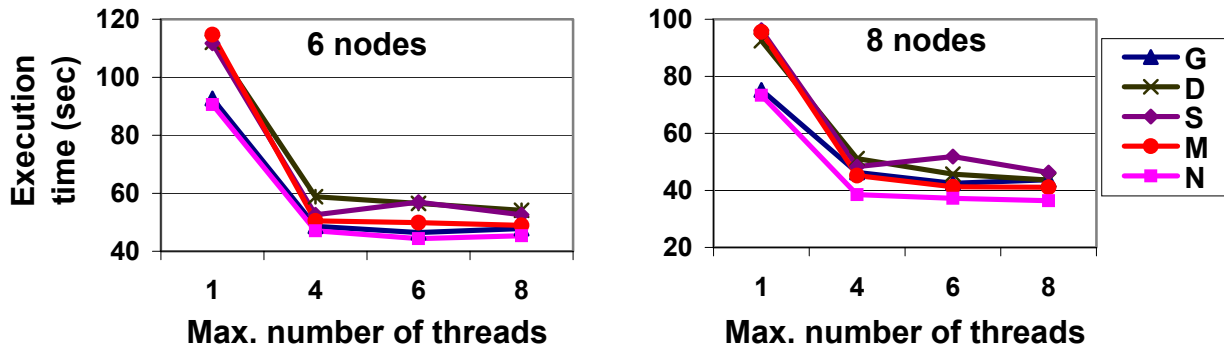


Fig. 5. The effect of multithreading on the execution time for 256r1.

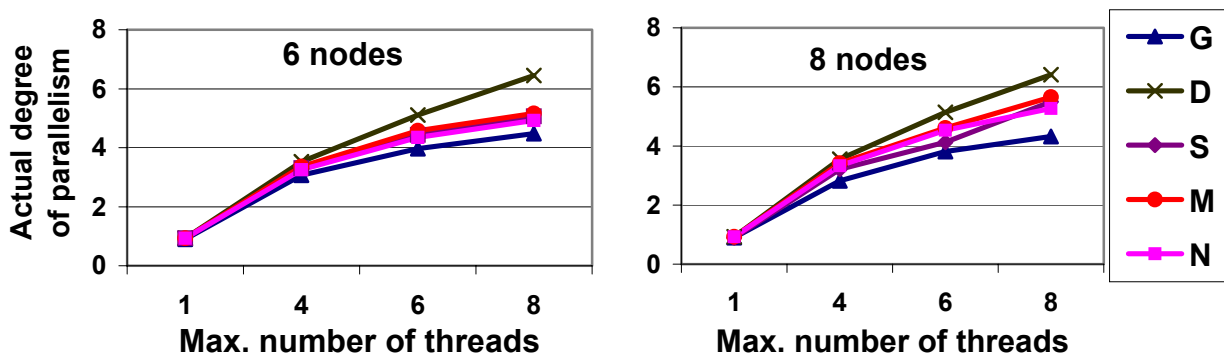


Fig. 6. The effect of the limit of threads per node on the actual degree of parallelism for 256r1.

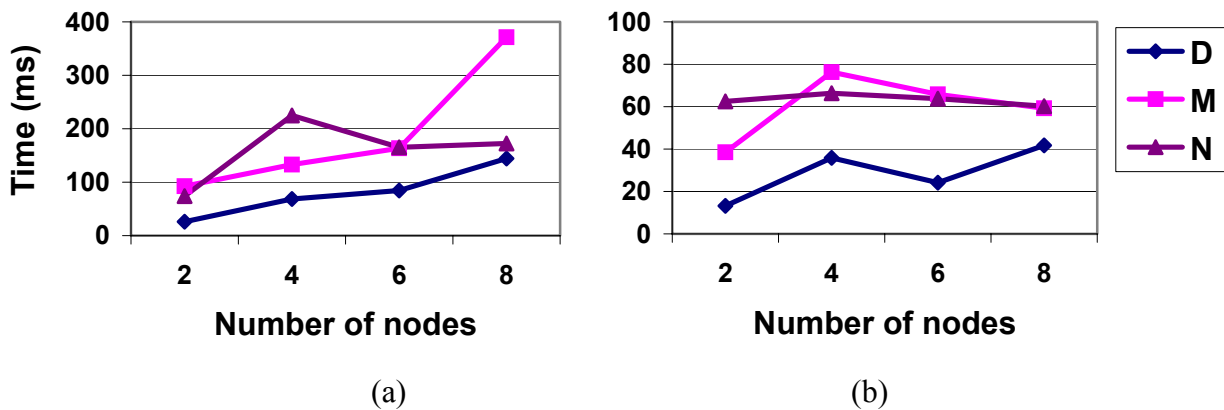


Fig. 7. Average synchronization time for a sub-matrix block. Block size: (a) 256x256. (b) 128x128.

4. Performance Analysis

R is slower than S in Fig. 2 because it suffers from high communication overhead for remote accesses. In most cases D is faster than S because it allows concurrent updates and decreases the penalty via a single synchronization at the end. However, when the number of nodes is very small the former benefit of D may not compensate for the synchronization overhead. From Fig. 2 and 3 we can see that there is correlation among the data access method for writes/updates, the degree of parallelism and the performance of SPs. The long communication latency has an adverse effect on the performance in PC clusters. SPM uses multithreading to hide this latency and employs coarse-grain tasks (i.e., SIs) to decrease its effect [4,5]. Since efficient multithreading requires enough parallelism, we must investigate the factors that affect it. Although the limit on the concurrent SIs for an IEU is six in Fig. 3, the actual number of concurrent threads is significantly lower. In the 128s and 128r cases, it is less than 3. One reason is the time needed for an SI to reach the IEU; this time depends on both the network latency and the efficiency of the IDU. Another reason relates to the way that SPs exploit the intrinsic parallelism in the problem.

In block-wise matrix multiplication, there are two types of parallelism. The first is inter-block parallelism (eP) appearing when all SDBs in the resulting matrix are computed in parallel. The second is intra-block parallelism (aP) involving the multiplication of pairs of sub-matrix blocks for a resulting SDB. For $n \times k$ and $k \times m$ SDBs, these degrees of parallelism are equal to $n \times m$ and k , respectively. Because of its exclusive writes, “SimpleUpdate” can only exploit inter-block parallelism. “RemoteUpdate” and “DistributedUpdate” can exploit both types of parallelism since SIs separate their computations from actually updating the resulting SDBs. The eP and aP parallelism in each experiment are shown in Table 3. In the 256s and 128s cases, the total eP is 256 and 1024, respectively. With 8 nodes and equibalancing, the theoretical limit on eP per node is 32 and 128, respectively; these numbers are much larger than 5 (the upper bound on the actual parallelism, as shown in Fig. 3). The actual parallelism drops because of the overhead in delivering SIs. This explains the small difference in actual parallelism among SPs. In contrast, in the 256r2 case the theoretical eP is only 16. When the number of nodes equals 4, 6 and 8, the eP per node is only 4, 2.66 and 2, respectively. However, by combining eP with aP in the D SP, the parallelism increases substantially (see Fig. 3). This also explains why the actual

parallelism drops significantly in S while it decreases slightly in D with increases in the number of nodes.

Table 3. Intrinsic degree of parallelism in our experiments.

| Experiment | Inter-block parallelism (eP) | Intra-block parallelism (aP) |
|-------------------|-------------------------------------|-------------------------------------|
| 256r2 | 16 | 64 |
| 256r1 | 64 | 128 |
| 128r | 64 | 128 |
| 256s | 256 | 16 |
| 128s | 1024 | 32 |

We can now investigate the effect of parallelism on performance. SPM allows data communications only in the beginning and at the end of an SI's execution [4]. The execution of multiple SIs under this scheme can follow a pipelined manner. Multithreading overlaps the pre-fetching of remote operands with the execution of previous SIs, and also result storing/updating with the execution of new SIs. Assume that the latency to get a remote SDB is t_r ; the probability to need a remote access is r ; the average number of input operands per SI is n_0 ; and the net CPU time to execute an SI is t_e . If the actual degree of parallelism is larger than $p_c = (r * n_0 * t_r) / t_e$, then the communication latency can probably be hidden.

Based on experiments with SDBs of size 128x128, the value of p_c is 1.41, 5.46, 9.86 and 12.97 with 2, 4, 6 and 8 nodes, respectively. Thus, these SDBs have fine granularity and the long communication latency cannot be hidden well. The system works under the delay bound condition [4]. For SDBs of size 256x256, the value of p_c is 0.57, 1.58, 2.18 and 2.60 with 2, 4, 6 and 8 nodes, respectively. In these cases the communication latency can be completely hidden. The system works under the CPU bound condition [4]. For this reason, the cases of 256s and 256r2 yield better performance than 128s and 128r, respectively. For matrix multiplication with same sized matrices, the former take less than half the time (case 256s versus 128s and case 256r2 versus 128r). In the case of 256r2 the difference in the actual degree of parallelism between the SPs is significant. The actual parallelism in "DistributedUpdate" is greater than 4 (see Fig. 3), and greater than or close to the critical value p_c . The actual parallelism in

“SimpleUpdate” is lower; in fact, it is lower than p_c for 6 and 8 nodes. This explains the large difference in performance for the 256r2 case shown in Fig. 2.

Based on this analysis, increasing the actual parallelism can improve performance up to a point under the delay bound condition. Further increasing parallelism would increase the number of SIs competing for the local CPUs and would extend the lifespan of SIs. This behavior is verified in Fig. 5 and 6. From Fig. 6 we can see that the actual degree of parallelism increases by increasing the maximum number of threads per node. The execution time of the SPs, however, decreases significantly only when the maximum number of threads increases from 1 to 4. Further increasing the maximum number of threads results in insignificant execution time decreases.

Although “distributed updates” increase parallelism, they require synchronization operations. Their overhead is insignificant compared to the gains. The available parallelism may vary during execution. In our experiments, the intrinsic eP decreases gradually as execution progresses. The inter-block parallelism is not enough only close to the end of the execution. Thus, expensive “distributed update” operations are only needed during the latter part of the SP’s execution. We could then combine “write” with “distributed update” operations to get the benefit of the latter with a reduced overhead. Fig. 5 testifies to this effect. Good performance is obtained by combining multithreading with the distributed representation of application data.

5. Comparison with Other Programming Models

Let us now compare our SPM model with others facilitating concurrent accesses. SPM adopts shared-memory concepts. Its key differences lie in granularity size and access constraints. Other models normally assume fine-grain data. They usually directly map data onto specific memory locations in a universal/program address space [15]. For shared-memory architectures, this mapping is direct. For distributed shared-memory architectures, the mapping is implemented by middleware [8,15]. A logical address is mapped to multiple physical addresses and the runtime support system handles data consistency among the latter [2]. Thus, application data are ultimately associated with physical locations. However, physical memory models limit data accesses to reads and writes. In SPM we use much coarser data represented by incarnated objects that facilitate more access operations.

We also use a message-passing approach to exchange messages between IEUs. However, our overall data representation approach is different than those used by message-passing

programming models. In the latter cases, application data is partitioned into several disjoint sets where each process holds a set. Data is represented locally with incarnated data structures in the particular process context, using a sequential model. There does not exist a universal global logical address space. Instead, each process has its own local logical space. Other application data can be requested via message-passing channels. We cannot consider received data as another representation of the original data but as newly created application data. This is because programmers control data. The communication logic is embedded into the application code. Local modifications do not reflect directly on the original data. Contrary to this, any data entity in SPM can be split and distributed; data coherence is maintained by the runtime support system.

In our runtime support system, incarnated data objects are cached in IEUs similar to COMA (Cache Only Memory Access) machines [6]. The data state information is held by the IDU. The host of the data can be changed if the current host cannot longer afford to host the master incarnated object. The key differences are: 1) The grain to be cached in SPM is an incarnated object; it is a memory line in COMA. 2) Data coherence in COMA is implemented in hardware; SPM employs software with communications via the general-purpose network. 3) COMA distributes only data but not their representations. Any cached copy of data either includes all information or is invalid. On the other hand, SPM distributes both data and their representation. In some states, some incarnated objects may not provide complete information. This distributed representation makes parallel updates possible.

Data in SPM resemble objects in object-oriented models. All of them encapsulate their contents, are accessed through interfaces and can even be accessed remotely. CORBA [12] and EJB [11] are good examples. EJB is closer since it can associate dynamically abstract data with incarnated objects. However, they differ as follows: 1) SDBs have limited size; EJB entities do not. 2) The representation of EJB entities is centralized; an EJB entity never appears in multiple containers and cannot be accessed concurrently [9]. However, SPM data may be represented by multiple incarnated objects in different IEUs and can be accessed in parallel. 3) An EJB entity always resides in a deployed container and relevant computations are always executed in this container. In contrast, SIs in SPM are free to execute in any IEU and the incarnated objects of a single data entity may float among IEUs.

6. Conclusions

In distributed computing environments, such as PC clusters, data virtualization can benefit performance. Using our SPM technique, not only do we distribute different application data throughout the system but we can also distribute the representation of a single logical data entity. This representation is split into a set of incarnated objects. Different incarnated objects in the same set may reside in different computer nodes. Each incarnated object may include the entire content of the logical data or partial information. The distributed representation of logical data facilitates the sharing of logical data among multiple processes. It alleviates various restrictions on accessing data in parallel and eventually increases the parallelism of application programs. This reduces the chance of a node being idle while waiting for requested data and eventually improves the overall performance.

References

- [1] G. Bell, J. Gray, What's next in high-performance computing? *Communications ACM* 45 (2002) 91-95.
- [2] A. Cox, R. Fowler, The implementation of a coherent memory abstraction on a NUMA multiprocessor: experiences with Platinum, *ACM Operating Sys. Reviews* 23 (1989) 32-44.
- [3] E.W. Felten, D. McNamee, Improving the performance of message-passing applications by multithreading, in: *Proc. Scalable High Perf. Comp. Conf.*, 1992, pp.84 - 89.
- [4] D. Jin, S.G. Ziavras, A super-programming approach for mining association rules in parallel on PC clusters, *IEEE Trans. Paral. Distrib. Sys.* 15 (2004) 783-794.
- [5] D. Jin, S.G. Ziavras, A super-programming technique for large sparse matrix multiplication on PC clusters, *IEICE Trans. Info. Systems* E87-D (2004) 1774-1781.
- [6] T. Joe, J.L. Hennessy, Evaluating the memory overhead required for COMA architectures, *ACM SIGARCH Computer Architecture News* 22 (1994) 82-93.
- [7] G. E. Kaise, B. Hailpern, An object-based programming model for shared data, *ACM Trans. Prog. Lang. Sys.* 14 (1992) 201-246.
- [8] P. Lee, Z. Kedem, Automatic data and computation decomposition on distributed memory parallel computers, *ACM Trans. Prog. Lang. Sys.* 24(2002) 1-50.
- [9] Y.J. Liu, Performance and scalability measurement of COTS EJB technology, in: *Proc. 14th Symp. Computer Archit. High Performance Computing*, 2002, pp.212 – 219.

- [10] J. Maassen, R. V. Nieuwpoort, R. Veldema, H. Bal, T. Kielmann, C. Jacobs, R. Hofman, Efficient Java RMI for parallel programming, ACM Trans. Prog. Lang. Sys. 23 (2001) 747-775.
- [11] V. Matena, S. Krishnan, L. DeMichiel, B. Stearns, Applying Enterprise JavaBeans™: Component-Based Development for the J2EE™ Platform (2nd Ed), Addison Wesley Professional , May 2003
- [12] Object Management Group, CORBA™/IIOP™ 3.0 Specification, OMG, Needham, MA June 2002
- [13] D.J Scales, K. Gharachorloo, A. Aggarwal, Fine-grain software distributed shared memory on SMP clusters, in: Proc. 4th Int'l Sym. High-Perf. Computer Arch., 1998, pp.125 – 136.
- [14] D.B. Skillicorn, D. Talia, Models and languages for parallel computation, ACM Computing Surveys 30 (1998) 123-169.
- [15] A. Skousen, D. Miller, Using a distributed single address space operating system to support modern cluster computing, in: Proc. 32nd Hawaii Int'l Conf. System Sci., 1999, pp.1-10.
- [16] D.W. Walker, Portable programming within a message-passing model: the FFT as an example, in: Proc. 3rd conf. Hypercube Concur. Computers Applications. Vol. 2, Pasadena, CA, 1989, pp.1438-1450.



Dejiang Jin received the B.Sc. in Chemical Physics from the University of Science and Technology of China, Hefei and the M.Sc. in Materials Science from Wuhan University of Technology. He has worked at the latter university and in the industry in the development of new materials. He also received the M.S. in Computer Science from NJIT in 2000. He is currently a Ph.D. student in Computer Engineering at NJIT.



Sotirios G. Ziavras received the Diploma in EE from the National Technical University of Athens, Greece, the M.Sc. in Computer Engineering from Ohio University and the Ph.D. in Computer Science from George Washington University. He was with the Center for Automation Research at the University of Maryland from 1988 to 1989. He was a visiting Assistant Professor at George Mason University in Spring 1990. He joined NJIT in Fall 1990 where he is now a Professor. He is an Associate Editor of the Pattern

Recognition journal. His main research interests are computer architecture, reconfigurable computers and parallel/distributed architectures and algorithms.