Adapting Distributed Shared Memory Applications in Diverse Environments

Daniel Potts and Ihor Kuz
National ICT Australia[*] and University of New South Wales
Sydney Australia
{danielp,ikuz}@cse.unsw.edu.au

Abstract

A problem with running distributed shared memory applications in heterogeneous environments is that making optimal use of available resources often requires significant changes to the application. In this paper we present a model, dubbed the view model, that provides an abstraction of shared data and separates the concerns of programming model, consistency, and communication. Separating these concerns makes it possible for applications to easily be adapted to different execution environments, allowing them to take full advantages of resources such as high speed interconnects and hardware-based memory coherence, and to be optimised for specific network topologies. Furthermore, it allows different data consistency protocol implementations to be used without requiring changes to the application code itself. We also present an implementation of the view model and provide experimental results showing how the view architecture can be used to improve the performance of a distributed shared memory application running in a heterogeneous multi-cluster environment.

1. Introduction

Grids, multi-clusters, NUMA systems, and ad-hoc collections of distributed computing devices all present diverse environments in which distributed computing applications can be run. Due to the diversity of features provided by these environments a distributed application that is to perform well must be specifically designed and optimised for the environment in which it is deployed. Such optimisations generally affect the application's communication structure, its consistency protocols, and its communication protocols.

Since applications are optimised in environment specific ways, reusing an application in different environments poses some problems. First of all, when an application's execution environment does not match the environment that it was developed for, problems with performance and poor resource utilisation arise. For example, an application designed to run on a cache-coherent NUMA (ccNUMA) machine may not run efficiently in a cluster environment due to the application's inability to compensate for added latency penalties resulting from the cluster's slower interconnects [10]. To avoid this, applications must be optimised for every new environment that they run in. Given the large differences between environments, most distributed applications need significant and time-consuming modifications if they are to achieve reasonable performance.

The second problem concerns the execution of distributed applications in non-uniform environments, that is, environments that consist of other sub-environments each with significantly different characteristics. Take, for example, a two-cluster system where one cluster consists of nodes connected by a high speed interconnect (such as Myrinet) while the other cluster consists of nodes connected by Ethernet. To achieve good performance an application running in this environment should be aware of the available interconnects and take full advantage of them [1, 9]. Thus, when communicating within the Myrinet cluster, a Myrinet specific protocol rather than TCP/IP over Ethernet should be used. Furthermore, the application should be partitioned in such a way that it takes advantage of the high speed interconnects while avoiding communication over the slower interconnects.

In both of these cases the main problem is that applications, and in particular their consistency protocols, require modifications that are highly tailored to the environment in which they run. Any significant change in the environment requires a change in the application. We present a solution to this problem in the form of a model (which we call the *view model*) that allows an application's consistency protocol, communication protocol, and communication structure to be modified without requiring significant (if any) changes to the application itself.

We provide an overview of the view model in Section 2.

[*]National ICT Australia is funded by the Australian Government's Department of Communications, Information Technology, and the Arts and the Australian Research Council through Backing Australia's Ability and the ICT Research Centre of Excellence programs.

A discussion of the model's properties and how these can be used to adapt applications to their environments follows in Section 3. An implementation of the view model is presented in Section 4 and in Section 5 we present experimental results that demonstrate that use of the view model can lead to performance and resource utilisation improvements. Work related to this project is briefly discussed in Section 6 followed by an overview of the current status and future work in Section 7 and conclusions in Section 8.

2. The View Model

The view model provides a shared data space abstraction based on the concept of *views*. A view represents a region of a shared data space. Examples of such shared data spaces include tuple spaces, shared memory, and name spaces such as those found in unix file-systems.

Shared data spaces consist of data elements. The format and structure of a data element is data space dependent. For example, a tuple space represents data in the form of tuples. An important characteristic of a shared data space is that it provides a way to reference, or name, its data elements. For example, in a tuple space, tuples are referenced by a key.

Besides the shared data space, a view also specifies the data sharing behaviour of that space. A view's data sharing behaviour determines how a view interacts with its environment, how it reacts to any external interaction, and how it manages shared data. More specifically, this includes behavioural characteristics such as replication of data, consistency of data, the timeliness of interactions, and the format and structure of interactions. For example, a behaviour specification may determine that a view provides a release consistent memory model with an eager update policy.

A view's data sharing behaviour specification partially determines that view's data sharing protocols. Data sharing protocols address the data consistency, communication and storage aspects of the behaviour specification. Hence, the implementation of a behaviour specification requires the selection of one or more data sharing protocols to correctly model the different aspects of the specification. For example, a view behaviour specification that includes eager-update release consistency requires an eager-update release consistency protocol and a suitable communication protocol in order to build a complete model of the specification. The selection of a communication protocol generally depends on the runtime environment. For example, for a cluster with an Ethernet interconnect, a TCP/IP communication protocol would be appropriate. Several suitable data sharing protocols matching a behaviour specification may be provided by a system. This allows the system to select a protocol based on environmental parameters and constraints.

In our model a distributed application accesses and modifies shared data through a view. The different threads of a distributed application act as the view's data producers and consumers and are known as *view clients*. Each view client is normally associated with a single view. View clients are independent of view details. A view's behaviour specification does, however, place restrictions on the interactions between the view and its view clients. This means that, a view client must always be compatible with a view's behaviour specification.

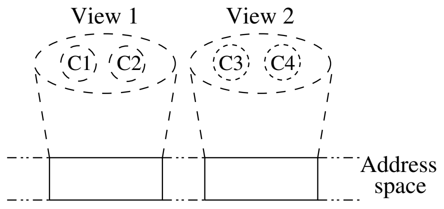

Figure 1. Several views representing regions of a traditional address space. Each view may provide different data sharing behaviour for the region they represent.

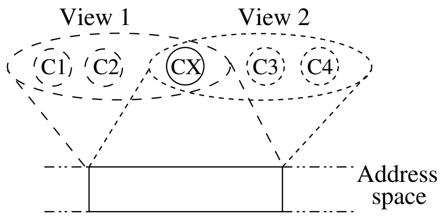

Figure 2. Overlapping views. Each view may provide different data sharing behaviour for the overlapping region.

Several views may be present within a single data space with each view representing a separate region of that space. This is illustrated in Figure 1 where the shared data space is a traditional address space. In this example, view 1 and view 2 represent separate regions of the address space, possibly providing different data sharing behaviour within each region. View 1 has two view clients named C1 and C2 which are able to access data in the region covered by view 1. Likewise, view 2's view clients are named C3 and C4 and access data in the region provided by view 2.

Clients within the same view see updates to shared data whenever the view's specified data sharing protocol makes them available. Hence, an update to data made by client C1 will be available to client C2 only when view 1's specified data sharing protocol makes it visible to its clients.

Two or more views may also overlap, that is they may represent the same region of data while providing different data sharing semantics to their view clients. These are known as *overlapping views*.

Figure 2 illustrates data sharing interactions between two overlapping views. When two views overlap the effect of a modification in one view on the other view is a result of the interaction between the views' corresponding data sharing protocols. This interaction is best explained through the introduction of a conceptual view client (client CX, in Figure 2). This view client is a client of both views. When client C1 performs a modification through view 1, client CX will be informed of the change according to view 1's data sharing behaviour. Once client CX is informed of a modification from view 1, it performs that modification in view 2. The other clients of view 2 (C3 and C4) will be informed of the change according to view 2's data sharing behaviour.

In Figure 2 the overlapping views represent regions of the same data space. It is also possible for views to represent the same data in different data spaces with *mapped views*. Mapped views share data by converting it using a mapping function which is implemented by a *view mapping client*. This mapping function may provide translation of data representation from one data space to another, and may interpret interactions between views in a manner that suits the mapped views. The view clients in each view are able to access the same data elements, but with different data space representations and view behaviour specifications.

The inherent differences between some data space representations make it difficult and unnatural to provide a generic solution to map a data element in one data space to another. Hence, a specialised mapping view client must be implemented to suit the specific sharing requirements of any distributed applications that wish to share or interact using different data spaces.

3. View Model Properties

The view model is similar to middleware in that it separates applications from the programming model API and implementation. Unlike traditional middleware, which generally provides a single layer of abstraction between the application and implementation, the view model defines four different layers and the interfaces between them. This is illustrated in Figure 3. At the top level in this figure is a distributed application such as a matrix multiply. This application is written according to a particular (view independent) programming model API, such as OpenMP [5].

The next level provides an implementation of this API. This is known as the *programming model client* level because it implements a specific view client. This level is implemented in terms of a *view abstraction interface*, which provides a programming model independent way for the view client to invoke the view specific behaviour implemented at the next level. This level does not implement the actual programming model behaviour, it simply provides a mapping from the programming model API to the view abstraction interface. The view abstraction interface will be described in greater detail in Section 4.

Below the view abstraction interface, the behaviour im-

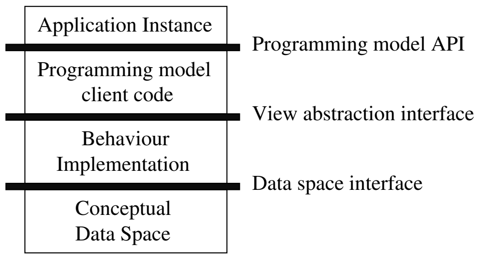

Figure 3. Conceptual separation of middleware of the view model.

plementation level provides the view's data sharing behaviour. This level includes implementations of consistency protocols, communication protocols and other behaviour related functionality.

The final level of abstraction represents the data space used by the behaviour implementation. The data space has a defined data element structure and an interface for storing, retrieving, and modifying shared data.

The structured layers and interfaces of the view model give us the flexibility necessary to provide several interesting properties. The most important of these is protocol selection, which is the ability to change the behaviour implementation of a view in order to better suit an application's underlying runtime environment or data sharing characteristics. The selection of a different protocol changes the view behaviour implementation without requiring changes to any of the interfaces or other layers. This is akin to replacing the implementation of a traditional middleware system without making changes to the application. However, the view model approach is more precise in that it allows the replacement of a single part of a programming model implementation, such as the communication protocol, without replacing other components.

The following example illustrates the benefits of protocol selection. Prior to run-time in a Grid environment, a distributed application does not always know what kinds of network interconnects will be available. The application must rely on protocols that are known to work across the whole execution environment. In this case it is safest to assume a generic TCP/IP communication interface. However, when the run-time environment includes specialised interconnects, such as shared memory or Infiniband, a generic protocol will fail to make full use of the improved resources. In such as case, allowing the application to change protocols can lead to marked performance and efficiency improvements.

Furthermore, on multi-processor machines with coherent shared memory, the application should use the available hardware support rather than rely on conventional software-based protocols. As such, protocol selection mechanisms provide a method for optimising and adapting a distributed application to its runtime environment and resources.

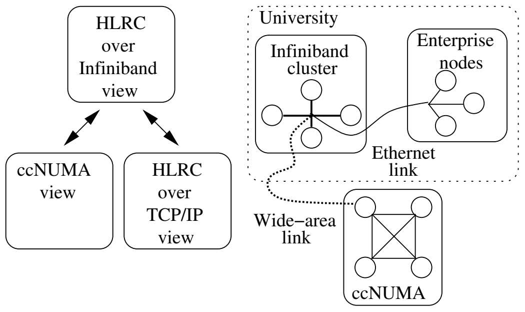

Figure 4. An example scenario where a program runs in a Grid comprising of several different systems. On the left is the view structure for the wide-area network on the right.

In heterogeneous environments such as multi-clusters and Grids, it is important to utilise the available underlying resources and communication capabilities. Overlapping views, combined with protocol selection, can be used in these environments to select protocols that best suit each of the distinguishable sub-environments.

For example, consider the scenario shown in Figure 4 where several clusters and nodes are spread out over a wide-area network. In this scenario it is desirable to utilise the available underlying resources of each cluster and to favour local area communication over wide-area communication. This can be achieved using the view model by encapsulating each sub-environment within a view, with each view using protocols best suited to its environment. For a distributed shared memory (DSM) application running in this environment, a suitable view structure is shown in Figure 4. In this figure, the Infiniband cluster utilises a view that provides a software-based protocol implementation of home-based lazy release consistency (HLRC) [8]. A view encapsulating the enterprise nodes selects HLRC over TCP/IP, and the ccNUMA view uses a protocol implementation that communicates directly using hardware-supported shared memory. Access to shared data within the ccNUMA system results in external communication only when the data is determined to be inconsistent by the view behaviour implementation. Any subsequent access to the same data within the ccNUMA system will occur directly over shared memory, without any unnecessary software intervention.

Another important property of the view model is programming model independence. The view model does not enforce any specific programming model. Hence, techniques using protocol selection and overlapping views can be employed and reused regardless of the programming model used by the application.

For example, the experiments presented in Section 5 use the same optimisation techniques, including the use of overlapping views, that we use for applications written using different programming models such as one-sided MPI. Furthermore, the implementation of communication protocols are reused between different programming models.

Programming model interoperability allows construction and execution of distributed applications that use more than one programming model. It allows interoperability between distributed applications that use different programming models such as visualisation applications.

4. View Architecture – An Implementation

The *view architecture* is an implementation of the view model. It defines a set of interfaces and operations that can be performed between views and view clients. This implementation also specifies a constraint. This constraint restricts all data space operations to utilise a large, single address space. That is, each datum is indexed by a unique address. Many programming model implementations map well to this model, hence, we choose to take this simple approach. Note that this does not prevent the use of view mapping clients which map to different data spaces which is important for supporting the use of programming models such as MPI or tuple spaces.

In our view architecture, the behaviour implementation is provided within a *view pager*. The view pager implements programming model functionality and protocols for managing consistency of a particular programming model. For example, most shared memory consistency protocols fit into this category, while the communication protocol used by the consistency protocols does not.

A view pager interacts with other view pagers and view clients when it needs to convey changes to consistency and programming model state. All interaction occurs through the view abstraction interface. The view pager does not differentiate between different view clients or view pagers. That is, it appears to communicate only with view clients via the view abstraction interface.

To understand how communication occurs between view client and pager, Figure 5 shows the interactions between two address space (AS) clients that are running a DSM application across two nodes. Each AS client behaves as a user-level pager for shared memory regions. It catches page faults within a view and maps data into the address space of the application based on instructions from the view pager.

In this example, a page-fault occurs in *AS client 1* for a data page that will be requested from *AS client 2*. At step (1), *AS client 1* catches the page fault and converts it into a view-interface operation known as an *update request*. This operation is used to request access to data within a view.

At step (2), the view-interface operation is invoked on the view pager. The view pager processes the operation according to its implemented consistency protocol. The invocation may be a remote procedure call (RPC) or a local function call depending on the configuration of the soft-

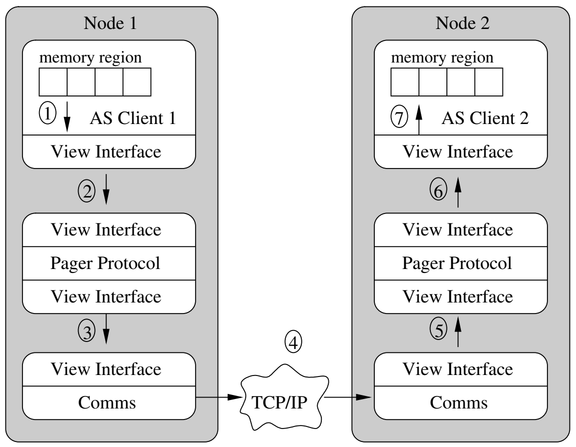

Figure 5. The path of a page fault request between two clients interacting using views.

ware. AS Client 1 then waits until it receives a reply for the corresponding data page.

At step (3) the view pager invokes a view interface operation directed at the target client. The view pager is not aware that communication will occur over TCP/IP since it uses the same view interface for local and remote clients. Hence at step (4), a view client that implements TCP/IP communication converts the view interface operation and any payload into a message suitable for sending over the network. The destination node, *node 2*, receives the message and converts it back into a view-interface operation. At step (5), the view pager on *node 2* receives the message as an *update request* operation that appears to be from *AS client 1*.

At step (6), the view pager determines that it must invoke the operation on the destination client. At step (7), *AS client 2* processes the operation and returns the data via an *update propagate* operation by invoking its view pager. Note that for local view pager and client interactions the data payload relies on zero-copy to avoid unnecessary copying.

When the *update propagate* arrives back at *node 1's* view pager, the view pager writes this operation's payload into the view data space. Depending on the current state of the target (*AS client 1*), the view pager may invoke an explicit *update propagate* operation or a cheaper synchronisation operation to the target view client. As it is already blocking, the operation is invoked immediately.

4.1. View Interface Operations

The view architecture implements the view abstraction interface using a set of operations known as *view interface operations* (VIOs). These operations are sent as messages between view clients and their view pager. The VIOs are divided into three core categories as shown in Table 1. First, *data coherence operations* specify actions to be performed on a region of data. Second, *synchronisation operations* control the flow of data and specify possible dependencies between data and nodes that modify the data. Third, high-level *view manipulation operations* allow for views to be created, manipulated and destroyed.

Table 1. View-interface operations.

update request	requests updates for given region
update propagate	propagate updates for given region
protection request	request access for given region
protection propagate	indication of new region access
token request	request a synchronisation token
token response	receive a synchronisation token
view create	create a new view
view select	select a view for use
view unselect	release a view

All data coherence operations specify a SAS address to index data on which the operation is performed. The SAS provides a global name space for identifying the data or the communication channel of a data stream. The treatment of an address within the SAS depends on the programming model and how addresses are interpreted. Conceptually a message or piece of data may be read and written from a given SAS address, independent of the programming model which determines how this operation will behave. This provides the mechanism for referring to data elements between view clients and their view pager.

The data coherence operations specify attributes other than a SAS address including length of data, access flags and optional protocol-specific state. Access flags allow the passing of permissions for on-going access to a region of data for programming models that require this feature.

Protocol-specific state may be used to convey additional information about the operation to a view pager. For example, view pagers that implement the same consistency protocol, such as HLRC, may include vector time-stamps as an additional consistency hint with data transfers.

4.2. Shared Memory Client Example

The following example, illustrated by Figure 6, clarifies the interactions between a view pager and a client that implements a DSM or other load/store programming model. Writing to memory (which, in this case maps directly to the SAS) causes the client to issue an `update_request` specifying an attribute for ongoing write access to that memory area. The view pager replies with an `update_propagate` granting that access. Any future writes to memory at that address occur without generating a new VIO until access is later revoked. The view pager implementation will behave like a typical DSM consistency protocol by mapping each operation to a particular consistency protocol action.

Other programming models may use and interpret VIOs

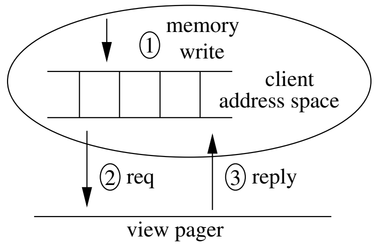

Figure 6. A distributed application client writing to shared memory triggers communication to a view pager via VIOs.

differently. For example, the previous approach is not adequate for message passing models, which can not easily represent data using a unique address in a SAS. Furthermore, message passing models do not have the concept of granting access to a region until it is revoked. Instead they rely on a flow of operations that represent a flow of messages. Hence, for a message passing model, the flow of VIOs between clients and pager is more important than the location in SAS that it occurs at. The SAS address is then used to name a communication channel rather than a location in memory.

As we mentioned earlier, interaction between views occurs via a specialised view client. For overlapping views the view client takes an incoming VIO from one view and forwards it to the other view. As our implementation of the view architectures uses SAS addressing for all data operations we avoid having to change the representation of data through our use of a SAS to represent data across all views. Hence, our view client for this purpose acts as a simple proxy that does not need to translate the data addressing method of one data space to that of the other.

Mapped views require a more complicated view client. For example, a distributed application that uses both two-sided MPI and shared memory models, must translate data space operations via a specialised view mapping client. Both programming models have different methods of referring to data elements. In this case, a view client that maps VIOs from one programming model to another needs to interpret how data elements are referred to in each model. In general, this is dependent on the application in question. We do not address mapped views further in this paper.

5. Evaluation

We evaluate the view model by running a distributed application in different configuration scenarios.

5.1. Experiment Environment

These experiments were performed across a small multi-cluster system that consists of two clusters of heterogeneous Itanium nodes running Linux 2.6 kernels. The first cluster consists of a single four-way SMP Itanium node. All benchmarks started execution from this node. The second cluster consists of a four-way ccNUMA Itanium2 node and six two-way SMP Itanium nodes connected through a 1000Mbit Ethernet switch. The two clusters are on separate networks connected by a 100Mbit Ethernet link. Internally each node in this multi-cluster provides hardware coherent shared memory between processors.

To demonstrate the benefits of the view model we compare the performance speedup of a 1200 by 1200 matrix multiplication in three different scenarios. Each scenario provides a different view configuration. In choosing the scenarios we focus on configurations that demonstrate the use of views for environment adaption, using hierarchical views and protocol selection.

The first scenario provides a traditional configuration with a single client process running on each processor. This configuration reflects a traditional DSM implementation where the underlying system implements a single consistency protocol that utilises a single communication protocol between all nodes. In this scenario, there is a single view that uses a strict consistency behaviour implementation where all communication between clients is over TCP/IP. Strict consistency ensures that all writes are instantaneously visible to all processes and an absolute global time order is maintained. This is implemented using a single-writer, multi-reader, page-based protocol. This implementation of strict consistency relies on a single home pager that manages the state of all pages. Any request for data that cannot be satisfied by a view client's own view pager, results in communication with the home pager.

In the second scenario we take advantage of views to establish domains of locality. In this scenario we define a domain of locality for each cluster by creating two separate views. This is illustrated in Figure 7. Internally each view implements a strict consistency protocol.

The final scenario shown in Figure 8 is similar to the previous scenario in that it provides a domain of locality for each cluster. However, in this scenario we also add another view per node. This new view uses a shared memory protocol implementation called multi-reader/multi-writer (MRMW) and takes advantage of the hardware-based coherent communication available on each node. Hence communication between clients within each node occurs over shared memory, while communication between nodes of each cluster and between clusters occurs using strict consistency over TCP/IP. The views are connected into a hierarchy that ensures coherent data access both within nodes and between nodes.

The matrix multiply application uses rows of 1200 double floating-point words for computation. Using a matrix row of this size ensures that page-based false-sharing oc-

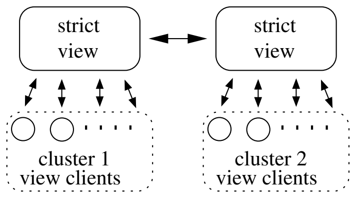

Figure 7. Multi-cluster view configuration for locality.

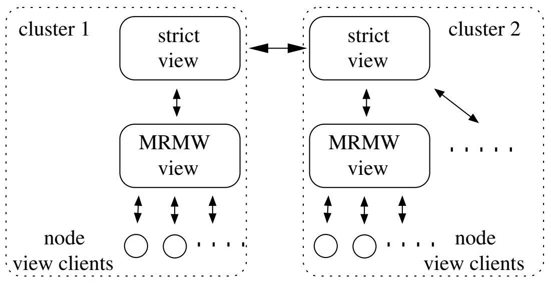

Figure 8. Multi-cluster view configuration for locality and protocol selection.

curs between nodes computing different adjacent rows. Because of this we expect to see significant neighbour communication, which is useful for demonstrating the affects of different view configurations.

We do not focus on optimising the matrix multiplication algorithm, nor on optimising consistency protocols, nor do we employ optimisation techniques such as multi-threaded processes. These methods are orthogonal to the use of views. The focus of these benchmarks is to demonstrate the use of views to improve the performance and efficiency of existing protocols in multi-cluster environments.

Speedup results for these experiments are shown in Figure 9. This figure shows the relative speed up of each configuration as the number of client processes increases. We allow only one client process per processor. In all experiments, the first four client processes run on the first cluster. As more client processes are added, we utilise more processors from the second cluster. Communication between clusters only occurs when more than four client processes are present.

5.2. Traditional Performance

The traditional performance shown in Figure 9 shows the matrix multiply reaching a maximum speedup of five times the single client performance when eight client processes are used. Adding more nodes causes performance to drop. This has a number of causes. Firstly, the bandwidth between clusters is limited to 100Mbps. This is quickly saturated due to the communication characteristics of the strict consistency protocol since the data used by each client process is sent separately to each node. Furthermore, false sharing generates excessive traffic since data pages are transfered between all client processes that are attempting to update the same page of data. This is illustrated in Figure 10 which plots the amount of data coming into the first cluster from the second cluster against the number of client processes. The result for the traditional scenario shows a large amount of traffic to due poor protocol behaviour as the number of nodes increase. Specifically, as the contention for false-shared pages increases, the amount of data transfered between nodes increases.

5.3. Two Cluster Domains

The performance of the two cluster domain scenario as shown in Figure 9 demonstrates improvements in performance and scalability with a relative speedup of over six times the single client case. The speedup remains reasonably constant as more clients are added but additional clients not contribute to improved performance.

In this scenario performance is improved due to a significant reduction in communication between clusters. Since each cluster is encapsulated into a view, communication of the matrix data set into the second cluster occurs only once rather then once-per-processor as in the traditional scenario. Furthermore, there is also a reduction in strict consistency protocol operations between clusters since many protocol interactions, such as requesting data, can be satisfied locally within the cluster.

The effect on the amount of inter-cluster traffic is clearly illustrated in Figure 10. Communication remains low and constant, demonstrating the effectiveness of views for encapsulating communication within clusters. This effect is largely due to a significant reduction in neighbour communication. Neighbour communication between clients of the second cluster now occurs without any communication into the first cluster.

5.4. Two Cluster Domains with Intra-node Views

The final view scenario shows greater performance improvements by adapting communication within each shared-memory node to utilise the available hardware coherent communication mechanisms. Figure 9 shows improvements in performance as more client processes are added to the computation. We see a peak relative speedup of eleven which is a significant improvement over the performance of the traditional scenario.

This improvement is due to the use of an internal view for communication between clients within each node, ensuring that data is fetched only once for each node rather than once for each processor. Since this results in a lower

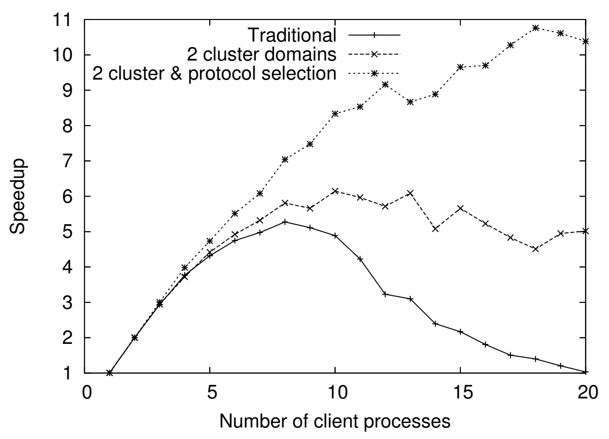

Figure 9. Speedup of matrix multiply for different view configurations.

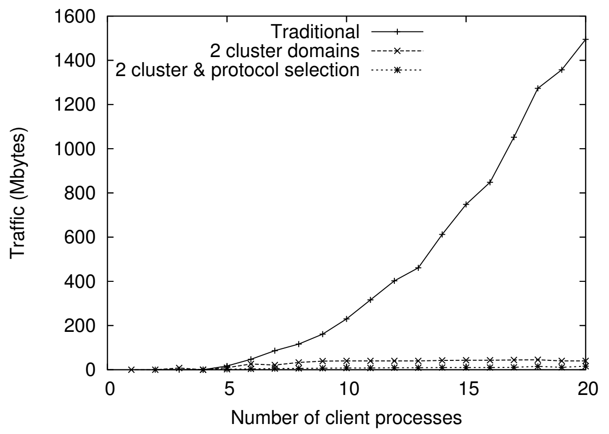

Figure 10. Inter-cluster receive network traffic for matrix multiply.

amount of external communication, the effects of the poor scalability of the underlying strict consistency protocol are reduced. This also reflected in a further decreased network utilisation as shown in Figure 10.

Furthermore, since the nodes communicate internally using shared memory, they are able to access shared data without any software intervention. Software intervention is only required when communication occurs outside the shared memory resulting is less time spend handling synchronisation in software.

6. Related Work

Many distributed applications implement their own data management mechanisms or utilise existing middleware that provides appropriate abstractions. In particular, distributed data stores attempt to provide a common platform for the efficient distribution of data. InterWeave [4] allows the programmer to map shared segments into programs running on multiple heterogeneous machines. It provides three levels of sharing to deal with different interconnect topologies: hardware-coherent multiprocessors, tightly-coupled clusters using software-based lazy release consistency and more coarsely grained version-based consistency for distributing shared segments. This partly addresses one of the problems tackled in this paper: taking advantage of local interconnects by using multiple consistency methods. However, it is not a general solution since it is specifically tailored towards particular environments and makes restrictive assumptions about consistency in data sharing.

Khazana [2] is a peer-to-peer data service that provides a common infrastructure for managing distributed shared data, allowing applications the flexibility of trading off consistency for availability and performance. It provides this using aggressive replication and customisable consistency management. Unfortunately, khazana lacks the ability to use a protocol specifically designed for optimal execution within a particular environment or to meet specific application requirements. Furthermore, it is not designed to allow multiple environments, each using its own optimal internal protocol, to interact consistently.

Some implementations of the message passing interface (MPI) including MPICH-G2 and Interoperable MPI (IMPI) [6], address issues of wide-area and heterogeneous computing by optimising point-to-point communication, particularly within specific environments such as clusters via protocol selection techniques. However, between clusters, TCP/IP communication is used which can be an inefficient use of the available interconnects.

Teamster [3] is a hybrid thread architecture that provides a transparent DSM system. It has been extended to Grid systems in Teamster-G by relying on a consistency protocol that implements eager-update page-based consistency. This protocol is suitable for wide-area access where latency is not a critical function of the program. Unlike the view architecture, it is not able to adapt to the specific environment utilised by the distributed application.

Tempest [11] is a communication interface that defines a set of mechanisms for implementing shared-memory policies. The tempest mechanisms include low-overhead messaging, bulk data transfer and fine-grained memory management. Using these mechanisms, a program can use various programming models ranging from message passing to shared memory models. The view architecture differs from Tempest in several ways. Firstly, the view architecture does not attempt to define mechanisms for protocol implementation. Instead, it proposes an interface for encapsulating protocols leaving the details of implementation up to the programmer. Furthermore, the details of fine-grained access or virtual memory management remain the task of the view client, rather than restricting the implementation to a particular programming model.

The concept of a view is also used in view-oriented parallel programming (VOPP) [7]. Our view model abstraction and VOPP are mostly orthogonal. The focus of VOPP is to assist the application programmer to better organise shared data in order to performance optimise the application. It achieves this by requiring the user to group data with similar attributes, such as frequency of access, into objects called views. Views become the granularity of access and are accessed exclusively. This behaviour is provided by the view-consistency protocol.

Our view model does not focus on techniques used in VOPP such as data partitioning. Instead the role of a view is to encapsulate and abstract a shared region of data so that it is possible to use existing techniques that better adapt the application its environment. As such, the view-consistency protocol and VOPP primitives could be implemented in our model as a view pager and programming model API, respectively, while providing their intended benefits.

7. Current Status and Future Work

Currently our implementation of the view architecture supports DSM and one-sided MPI programming model interfaces. For the DSM models, there are hardware-based protocol implementations suitable for ccNUMA and SMP machines, and software-based protocols for communication over TCP/IP or raw Ethernet. We plan to explore view implementations of HLRC which has weaker consistency than the current strict consistency implementation.

We are in the process of implementing a two-sided MPI programming model interface on views. We plan to explore the use of views for broadcast and performing collective operations in heterogeneous environments. We also plan to look at multi-model applications that simultaneously use two-sided MPI and shared memory programming models. We will also explore the use of views for other aspects of distributed, multi-cluster and Grid computing, such as bulk data transfer, fault tolerance and fault detection.

8. Conclusion

The view architecture is based on a flexible and generalised model for controlling and adapting shared data to a distributed application's underlying environment. It relies on a separation of concerns between the client application, programming model, consistency and communication protocols, and sharing interactions. We believe that this approach is suitable for distributed application data sharing in wide-area environments such as multi-clusters and Grids, and, in particular, that it provides mechanisms for improving the performance of existing DSM applications and protocols in such environments.

This paper has demonstrated the performance advantages of using the view architecture for a typical distributed shared memory application. In particular, we have shown a significant performance advantage in environments that offer specialised interconnects such as ccNUMA and SMP systems. Furthermore, we have also demonstrated the ability to optimise communication across heterogeneous environments easily.

References

[1] H. Bal, A. Plaat, M. Bakker, P. Dozy, and R. Hofman. Optimizing parallel applications for wide-area clusters. In *Proceedings of the 12th International Parallel Processing Symposium*, pages 784–790. IEEE Computer Society, 1998.

[2] J. Carter, A. Ranganathan, and S. Susarla. Khazana: An Infrastructure for Building Distributed Services. In *Proceedings of the 18th IEEE International Conference on Distributed Computing Systems*, pages 562–571, Amsterdam, The Netherlands, May 1998.

[3] J. B. Chang and C. K. Shieh. Teamster: a transparent distributed shared memory for cluster symmetric multiprocessors. In *Proceedings of the 1st IEEE International Symposium on Cluster Computing and the Grid*, pages 508–513, 2001.

[4] D. Chen, S. Dwarkadas, S. Parthasarathy, E. Pinheiro, and M. L. Scott. InterWeave: A middleware system for distributed shared state. In *Languages, Compilers, and Run-Time Systems for Scalable Computers*, pages 207–220, 2000.

[5] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, 1998.

[6] W. L. George, J. G. Hagedorn, and J. E. Devaney. IMPI: Making MPI interoperable. *Journal of Research of the National Institute of Standards and Technology*, 105(3):343–428, 2000.

[7] Z. Huang, M. K. Purvis, and P. Werstein. Performance evaluation of view-oriented parallel programming. In *Proceedings of the 34th International Conference on Parallel Processing*, pages 251–258, 2005.

[8] L. Iftode. *Home-based Shared Virtual Memory*. PhD thesis, Dept. of Computer Science, Princeton University, June 1998.

[9] N. T. Karohis, B. Toonen, and I. Foster. MPICH-G2: A grid-enabled implementation of the message passing interface. *The Journal of Supercomputing*, 63(5):551–563, 2003.

[10] A. Plaat, H. E. Bal, and R. F. H. Hofman. Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects. *Future Generation Computer Systems*, 17(6):769–782, 2001.

[11] S. K. Reinhardt, L. R. Larus, and D. A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 325–336. IEEE, 1994.