# Enabling the Efficient Use of SMP Clusters: The GAMESS/DDI Model

Ryan M. Olson, Michael W. Schmidt and Mark S. Gordon
Department of Chemistry and Ames Laboratory
Iowa State University
Ames, IA 50011 USA

Alistair P. Rendell
Department of Computer Science
Australian National University
Canberra ACT 0200
Australia

*An important advance in cluster computing is the evolution from single processor clusters to multi-processor SMP clusters. Due to the increased complexity in the memory model on SMP clusters, new approaches are needed for applications that make use of distributed-memory paradigms. This paper presents new communications software developments that are designed to take advantage of SMP cluster hardware. Although the specific focus is on the central field of computational chemistry and materials science, as embodied in the popular electronic structure package GAMESS (General Atomic and Molecular Electronic Structure System), the impact of these new developments will be far broader in scope. Following a summary of the essential features of the distributed data interface (DDI) in the current implementation of GAMESS, the new developments for SMP clusters are described. The advantages of these new features are illustrated using timing benchmarks on several hardware platforms, using a typical computational chemistry application.*

## 1 Introduction

High performance computing has played a major role in the advancement of science and the application of scientific theory to solving modern problems. The evolution of the supercomputer from megaflops, to gigaflops, to teraflops and beyond has stimulated improvements in both the size and accuracy of the theoretical models than can be solved. In the field of computational chemistry, the smallest molecular systems can now be studied at levels equaling or even surpassing the abilities of measurable experiments. However, due to the computational cost of these state-of-the-art methods, even on small systems and using the best available hardware, calculations may take weeks or even months to complete. Inevitably chemists are forced to balance the accuracy of their calculations by the need to obtain the results in a timely fashion. For this reason, coupled with the desire to perform ever larger and more accurate calculations, computational chemists were motivated to become early proponents of high performance computing (HPC).

GAMESS is a widely used computational chemistry package [1] that has constantly sought to exploit the latest HPC platforms, in particular massively parallel processors (MPPs). As well as achieving high performance, the GAMESS developers have been equally keen to ensure that their code is also portable across a wide range of sequential and parallel platforms, thus enabling the community to use the same code on platforms that range from PC to MPP.

The parallel model used in GAMESS is constantly evolving to take into account advances in HPC hardware and software. Initially the parallel code was based on a replicated data message passing approach that used TCGMSG [2], but this moved to MPI-1 [3] as that standard became widely accepted and available. Then to make better use of the large aggregate memory available on parallel machines the replicated data approach has been progressively replaced by algorithms that use the Distributed Data Interface (DDI) [4]. The

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SC'03, November 15-21, 2003, Phoenix, Arizona, USA
Copyright 2003 ACM 1-58113-695-1/03/0011...$5.00

latter essentially provides an interface by which all processes in a parallel job can independently access and modify any data element in a distributed data array. This is true even when the array in question is stored across the memory of a physically distributed memory computer. In this respect DDI is similar to, and was in part inspired by the early work of Nieplocha et al. on Global Arrays (GA) [5,6].

The initial version of DDI was developed for Cray T3D systems, and used the one-sided communication capabilities of Cray's SHMEM library [7] to implement the distributed arrays. However, to provide portability to other systems in which native one-sided communications were not supported, a data server model was developed [4]. In this scheme two processes are (nominally) assigned to each CPU, with one process performing the traditional computational tasks, while the other exists solely to store and service requests for the data associated with the distributed arrays. Depending on the platform, communications between the compute and data server processes occur either via TCP/IP socket connections or MPI.

With the GAMESS/DDI data server model much useful computational chemistry has been performed. This has been particularly true in recent years with the emergence and widespread availability of inexpensive Beowulf cluster systems [8]. Evidence for this is provided by the increasing number of literature citations of GAMESS, now averaging 30-50/month. The data server model was well suited to early Beowulf clusters as these were constructed from single CPU PCs, and the pairs of compute and data server processes could be naturally constrained to run on each PC node. More recently, however, clusters are increasingly being assembled using symmetric multi-processor (SMP) nodes containing a few (typically ≤4) CPUs coupled with high performance and potentially intelligent interconnect networks like Gigabit Ethernet [9], Myrinet [10], SCI [11], or Infiniband [12]. A similar trend is also evident in dedicated supercomputers, where, for example, large-scale IBM SP and HP SC systems now use SMP nodes. Indeed, very large shared memory computers, like the SGI Origin 3000 or HP GS, usually have Non-Uniform Memory Access (NUMA) architectures that can be viewed as a cluster of uniform memory SMPs linked via a network, albeit a very good network.

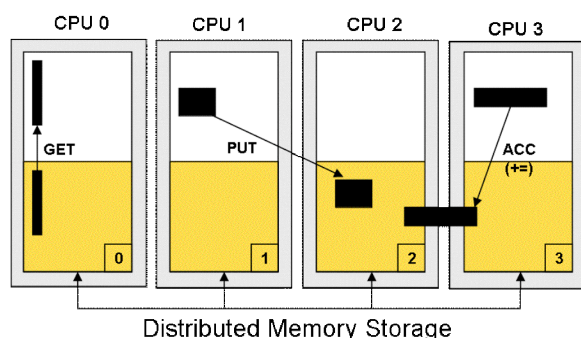


**Figure 1:** The virtual shared-memory model. Each large box (grey) represents the memory available to a given CPU. The inner boxes represent the memory used by the parallel processes (rank in lower right). The gold region depicts the memory reserved for the storage of distributed data. The arrows indicate memory access (through any means) for the distributed operations: get, put and accumulate.

With this move away from single processor toward multi-processor based clusters we are confronted with a considerably more complicated memory model than that which was present when DDI was originally conceived. Now small groups of processes have equally fast access to chunks of memory, while accessing memory between groups of processes is slower. Recognizing this plus the success and popularity of distributed-memory programming models, it is pertinent to consider how these models might be extended to better exploit SMP clusters. *The aim of this paper is to begin to address this issue, presenting an enhanced version of DDI that includes new functionality specifically targeting SMP clusters.* Using both the new and original versions of DDI, performance results are presented and discussed for a typical GAMESS computation run on a variety of MPP systems. First, however, we begin with a brief discussion of the existing DDI data server model used in GAMESS.

## 2 The Distributed Data Interface

The Distributed Data Interface (DDI) developed by Fletcher et al. [4] became the parallel interface for the GAMESS computational chemistry program in 1999. Prior to this, as mentioned above, parallelization used a replicated data approach and the TCGMSG message-passing library [2]. The initial parallelization permitted a reduction in execution times. However, the problem size was still limited by the resources available on the smallest node. DDI was developed as a means to support distributed-memory programming, while also providing a level of abstraction between the user program (i.e., GAMESS) and the underlying communication libraries and hardware.

### 2.1 DDI Programming Model

The DDI programming model is based on the idea of "virtual shared-memory", where a portion of the physical memory available to each processor is designated for the storage of distributed data (Figure 1). In this model, there are two types of distributed-memory: local and remote. Local distributed-memory is defined as the memory a given process uses to store its portion of the distributed data, while remote distributed-memory is the memory reserved by all the remaining

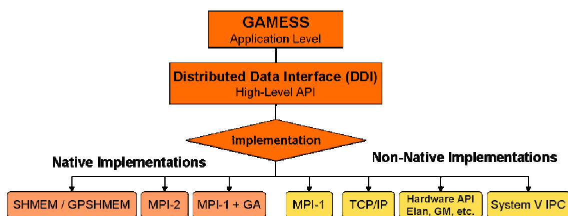


**Figure 2:** The implementation hierarchy for a DDI application; in this case GAMESS. Native implementations of DDI fully support the communication requirements of DDI; non-native implementations require explicit programming or special models to achieve full functionality.

parallel processes for their portions of the distributed data. Every process in a parallel job is allowed to access/modify any element in the distributed memory segment (regardless of its physical location); however, access to local distributed-memory is assumed to be faster than access to remote distributed-memory. Thus the DDI programming strategy aims to maximize the use of local distributed data while minimizing remote data requests. Note that the performance penalty for accessing distributed-memory (local or remote) is completely dependent on the underlying machine architecture and the parallel library/libraries used in the implementation of DDI.

## 2.2 DDI Implementation

The DDI framework consists of a small set of functions required for replicated and distributed memory programming. These include common point-to-point and collective operations, along with distributed-data operations, such as one-sided gets, puts and accumulates. There are essentially two types of DDI implementations: native and non-native, as illustrated in Figure 2. A native implementation implies that the underlying parallel library (or libraries) fully supports all the necessary communications operations required by DDI, i.e. DDI becomes a wrapper to this library. A non-native implementation of DDI implies the underlying parallel libraries are deficient in some manner and that further explicit programming is required to reach full DDI functionality. Any implementation must support all of the required DDI functions.

In the following sections we discuss the evolution from the native SHMEM implementation of DDI developed for the Cray T3D/T3E systems to the non-native data server model used on early clusters. We then discuss the benefits and limitations of the data server model, as this relates directly to the improvements made for DDI on SMP clusters.

## 2.3 Native SHMEM Implementation

The DDI "virtual shared-memory" model (Figure 1) most resembles the SHMEM programming model that originated on the Cray T3D. On these systems, memory is physically distributed and processes do not migrate between CPUs, however the address space is shared and by using the SHMEM library any process can read, write or accumulate to any memory location. In a parallel job, a distributed array is created with the array divided into N disjoint sub-patches that are stored in the memory associated with each of the parallel processes. In order to perform distributed-memory operations (get/put/acc), each process maintains a mapping of the distributed array across the set of parallel processes. To guard against simultaneous accesses to the same memory location SHMEM locks are used.

## 2.4 Non-Native Implementations

To support the "virtual shared-memory" model on clusters in which one-sided access to remote memory is not directly available, DDI adopts a data server model. In this model a portion of the memory that is nominally associated with each CPU is designated for the storage of a distributed data object. However, in contrast to the SHMEM model there is no requirement that this memory be directly accessible to the process executing the parallel task (the compute process). Instead, to simulate one-sided communications this memory is associated with a second, data serving process. Moving the local patch of the distributed-memory segment to a separate process means that, subject to the scheduling policy of the underlying operating system, this process will always be available to service requests for its data. An illustration of this model is given in Figure 3.

As with the SHMEM model each compute process maintains a mapping of the disjoint sub-patches of each

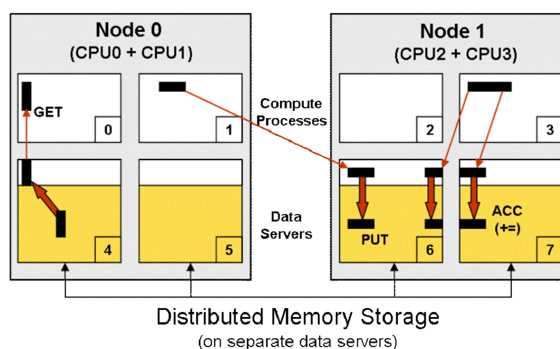


**Figure 3:** Data server model on two 2-CPU SMP nodes. Each large (grey) box represents the available memory on the node and the inner boxes represent the memory used by the parallel processes (rank in lower right). The memory reserved for distributed-data (gold) is held in the local address space of the data servers. Thin arrows indicate data transfers via the message-passing library, while thick arrows indicate direct memory copies.

distributed array to the process responsible for that sub-patch. However, in contrast to the SHMEM model, what was previously the "local" patch of a distributed array no longer resides in the address space of that compute process, but must be fetched from the associated data server process. Thus when running on an N processor machine, each CPU executes one compute process and one data server process. For convenience the ranks of the compute processes are [0,N-1] while the ranks of the data servers are [N,2N-1], with the rank of the data server associated with compute process A being A+N.

Technically, the data server processes are the same executable as the user's parallel program, but specialized to become data servers on DDI initialization. Thus the code maintains a Single Process Multiple Data (SPMD) model. The role of the data server is to respond to data requests initiated by the compute processes. To do this each data server process waits for incoming requests to arrive. In the TCP/IP implementation while waiting for a request, each data server process is put to sleep, thus essentially yielding full CPU access to the compute process. Achieving the same effect with MPI is not usually possible, since most MPI implementations continuously poll for incoming receive calls and *in so doing compete directly with the compute process for CPU cycles*. In this respect the TCP/IP data server implementation is usually found to outperform the MPI implementation.

In the TCP/IP data server model when the data server receives a request from a compute process it is woken up and responds appropriately. If a get operation is issued, the requested portion of the distributed array is packed into a contiguous message and sent back to the requesting compute process. Similarly, if a put or accumulate operation is requested, the data server receives the incoming data segment from the compute process, unpacks it, and places it in the relevant location. In this respect the data server model also has some advantages over the SHMEM model. In the SHMEM model accessing a remote patch of memory is likely to give rise to many SHMEM calls corresponding to accesses to different rows or columns of the distributed array. In contrast in the data server model a single message can be sent to the remote process with the contents of the remote memory request packed into a single message.

The dual process data server model guarantees exclusive access to distributed memory during one-sided operations, since a data server can only handle a single request at a time. The model was well suited to early Beowulf clusters, since each node usually comprised a single processor PC and this naturally constrained each compute and data server process to execute on the same CPU. In contrast, on a multi-processor SMP node, in the absence of process to processor binding, the possibility exists for the compute and data server processes to execute on any available CPU. The dual model is also very portable, since all inter-process communications rely only on point-to-point operations and, in the absence of anything better, TCP/IP sockets are widely supported.

Forcing all inter-process communication to run through some sort of message-passing library makes for a simple and portable model, but seriously degrades the rate at which a compute process can access its own sub-patch of a distributed array. That is, unless the message passing library makes use of the fact that both the compute process and its associated local data server are co-located in the same memory space and uses this fact to achieve improved performance, the advantage of local data over remote data is greatly reduced. Similarly this model also ignores the improved availability each compute process has to memory associated with any of the data servers located within its own SMP node - because each CPU is effectively treated as a separate node, with the total memory divided equally amongst the processes in that node. In the following section we discuss how we have used shared memory segments and semaphores to substantially enhance the performance of the DDI data server model on both single processor and SMP based clusters.

## 3 Improved DDI Data Server Model

As mentioned above, the DDI programming model encourages the user to maximize the use of local distributed memory while minimizing the use of remote distributed memory. However, the original data server implemented all communications using message passing, and this is likely to substantially degrade the advantage of the local over remote distributed data. When this model was first developed, this was a

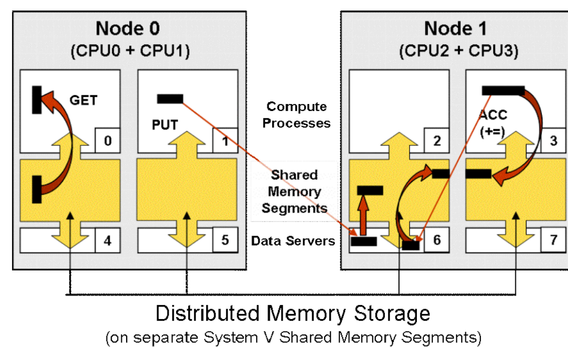


**Figure 4:** The FAST DDI model is a modification on the original data-server model (Figure 3) in which the memory used for distributed data storage is removed from the local address space of the data servers and implemented as a shared-memory segment. As shown, only one compute process/data server pair is attached to a given shared-memory segment. In this model, the get operation can be accomplished entirely through a memory copy, while the accumulate operation can be partially completed through shared-memory; the remaining portion must use the message passing library and run through the data server.

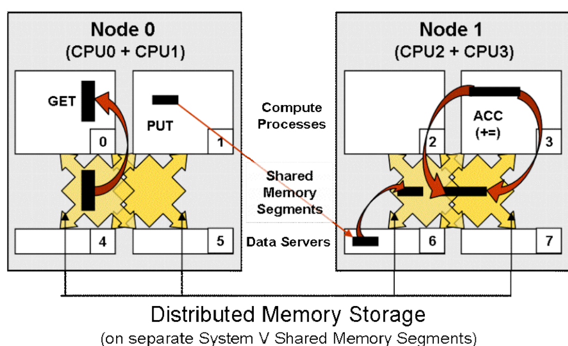


**Figure 5:** The FULL shared-memory model develops from the FAST model (Figure 4), but now all DDI processes within a node attach to all the shared-memory segments. The accumulate operation shown can now be completed directly through memory.

workable option since for most GAMESS calculations the time spent communicating was a small fraction of the total wall time. The increasing use of highly correlated calculations means this is no longer true. Our first improvement has been to use shared memory segments to store all distributed data quantities, while maintaining one memory segment per data server. Each shared memory segment is then attached to both the compute and data server processes. Using this model, the compute process now has direct access to its local distributed-memory, and only needs to use message-passing when accessing remote portions of the distributed array. This model is illustrated in Figure 4 and will be referred to as the FAST implementation. This model provides a substantial performance boost over the original data server model even on single processor clusters, since it enhances the speed at which a compute process can access its local portion of a distributed array.

In this new model the data server process behaves in the same manner as in the original model in responding to data requests from compute processes. However, exclusive access to the distributed-memory during data requests is no longer guaranteed as both the data server and the associated compute process can now compete to access data from the shared memory segment to which they are both attached (Figure 4). Thus, to control access to the shared memory segments we have used an array of general semaphores. One semaphore is used to independently control access to each distributed array with two types of access permitted: read-access and write-access. Thus before either the DDI compute or data server process is allowed to operate (get/put/acc) on a portion of a distributed array, it must first gain access via the appropriate semaphore. A process that is granted read-access is not guaranteed exclusive access. This is useful to permit multiple get operations from the same segment at the same time. Write-access is exclusive access and needs to be acquired before a process can put or accumulate to a distributed array. Since access is granted based on array handles rather than entire shared memory segments, simultaneous access to different arrays is allowed, i.e. the compute process could be accumulating to one array while the data server is accumulating to another. This multi-level access provides another significant advantage over the original implementation which effectively locks the entire local distributed-memory segment (all arrays) whenever any type of distributed data operation is performed.

The next step is to realize that use of shared memory segments permits intra-node communication on multi-processor systems to be considerably improved by allowing all compute and data servers within a given node to attach to all shared memory segments on that node. This enables the data in all of these segments to be accessed at a rate that is equivalent to it being local. While to date the DDI programming philosophy has been to assume that only one block of the distributed array is local to any given compute process, it is possible that future DDI based algorithms may benefit from knowing that on SMP clusters more than one block is effectively local. To this end we have extended the DDI functionality to include two new functions; DDI_NDISTRIB and DDI_NNODE. The former provides the user with information as to what portion of a distributed array is local to that SMP node, while the latter returns the total number of SMP nodes being used and the rank of the SMP node that the calling process is running on.

The strategy taken to develop full SMP support for the data server model is a small, but important, step forward from that used in the FAST DDI scheme (Figure 4) in which shared-memory segments are shared only between pairs of compute and data server processes. The new scheme is illustrated diagrammatically in Figure 5, and will be referred to as FULL SMP support. Data access in the FULL SMP model is no different from that in the FAST model, because each shared memory segment already has its own array of semaphores to control access. The only (non-trivial) difference is that now the possibility for multiple simultaneous access has increased further.

The role of the data servers in the FULL SMP implementation of DDI is now considerably different from the original implementation. There are two important implications that result when each data server is given access to all the shared-memory segments on that node. First, each data server can now handle data requests for any patch of the distributed array that resides on that node, not just the data owned by its associated compute process. The second implication is that all the data servers on a node are equivalent.

The equivalence of the data servers within a node means that we can now change how a compute process on one node requests distributed-data from a remote

**Table 1:** DDI benchmark platforms.

| Platform | #CPUs/Node | Clock Speed | Memory/Node | Interconnect | Peak Bandwidth |
|---|---|---|---|---|---|
| HP SC | 4 | 1.0 GHz | 4GB | Quadrics | 340 MB/s |
| IBM Power3-II | 4 | 375 MHz | 16GB | Gigabit Ethernet | 125 MB/s |
| Compaq | 1 | 667 MHz | 1GB | Myrinet | 240 MB/s |
| HP GS1280 (EV7) | 16 | 1.1 GHz | 16GB | Shared Memory | |

node, i.e. how inter-node distributed data requests are handled. In the original model, a compute process would send a data request to each data server that was responsible for part of the distributed data it was seeking. On SMP nodes this could mean multiple requests being sent to the same node. In the FULL SMP implementation, all compute processes now maintain a mapping of each distributed array both by processor and by node. Thus when performing a distributed memory operation, the compute process checks the node mapping rather than the processor mapping and sends one request to just one of the remote node data servers. This data server then returns all the requested data even if it was gathered from multiple shared memory segments. The implication is that there can be a significant reduction in the total number of point-to-point operations needed in order to receive the same amount of remote data.

Finally we note that the current decision to use multiple shared-memory segments in the full model instead of one large shared memory segment is partly prompted by recent trends in NUMA shared memory machines, like the HP GS1280, SGI Origin 3000, and 64-bit AMD Opteron. In these architectures, each CPU has a dedicated memory controller and a local bank of memory. Shared-memory between processors is accomplished via high-speed communications links in the processors and some means of cache coherency. Because of the disjoint nature of the memory in NUMA machines, the creation of a single large shared memory segment relies too heavily on the OS to correctly distribute the shared-memory segment throughout the memory of the machine. The creation of multiple shared-memory segments helps ensure that the memory that is meant to be local for a given CPU is actually present in that CPU's local memory.

# 4 Results / Timings

To test the performance of the two new DDI data server implementations, the new codes were benchmarked against the original data server and, when available, the SHMEM code. For reasons noted above, the underlying message-passing library for all data server implementations is TCP/IP sockets. The results (discussed below) labeled "DDI-Socket" and "DDI-SHMEM" refer to the original DDI implementations, while those labeled "DDI-Fast" and "DDI-Full" correspond to the two new DDI implementations described in Section 3.

Performance data was obtained for a variety of platforms that used various high-performance interconnects. Details of these platforms are given in Table 1. The HP SC and the IBM Power3 platforms were chosen because they represent the building blocks for modern cluster based supercomputers, but differ in their network capabilities. The IBM uses commodity Gigabit Ethernet while the HP SC uses the much more expensive Quadrics interconnect. The single processor node Compaq cluster built on Myrinet is included to test the improvements gained by having fast access to local distributed-memory in the new shared-memory model. Finally the HP GS1280 is a NUMA shared-memory machine that is scalable up to 64 CPUs. We include this system to assess the suitability of the new FULL shared-memory data server model for a genuine large scale shared memory system.

The GAMESS calculation that we have used to assess performance is a second order perturbation theory (MP2) gradient evaluation on a benzoquinone derivative used in the synthesis of hongconin, a cardioprotective natural product. This is the same benchmark used by Fletcher et al. [4] in previous work. It uses 245 atomic basis functions and requires a total of ~1250 MB of distributed-memory to run. The computational cost of this type of calculation, as implemented in GAMESS, scales as $O(N^5)$, where N is the number of atomic basis functions. The memory requirements for the distributed arrays scale as $O(N^4)$. The MP2 method is widely used in quantum chemistry to provide accurate information on the energetics, kinetics and infrared spectra of molecules.

The distributed-memory MP2 gradient [13,14] was chosen as a benchmark because the algorithm performs a significant number of local and remote distributed data operations, thus stressing the underlying DDI implementation. By current standards in the chemistry community, the MP2 gradient for benzoquinone is a relatively small calculation, but for benchmarking purposes it is necessary that the calculation can fit into memory available on a single CPU on each of the various platforms available to us. Even though this benchmark is regarded as relatively small, the

**Table 2:** DDI communication breakdown showing the usage of shared memory transfers for the socket, fast and full versions of DDI as a function of number of processors and nodes on the HP SC. Values have been averaged over all compute processes. Results given under "% of All" represent the total number of DDI calls and as such these values are the same for all three DDI implementations.

| | Number of Processors / Number of Nodes | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | 4 / 1 | | | 8 / 2 | | | 16 / 4 | | |
| | Calls | Mbytes | % of All | Calls | Mbytes | % of All | Calls | Mbytes | % of All |
| **All (via socket + shared memory)** | | | | | | | | | |
| DDI_Get | 90329 | 4567 | 28.6 | 45165 | 2284 | 28.6 | 22582 | 1142 | 28.4 |
| DDI_Acc | 6480 | 10736 | 67.4 | 6480 | 5369 | 67.4 | 6480 | 2684 | 67.4 |
| DDI_Put | 3238 | 637 | 4.0 | 1619 | 319 | 4.0 | 809 | 159 | 4.0 |
| **Local (via shared memory)** | | | | | | | | | |
| DDI_Get | | | | | | | | | |
| DDI-Socket | 0 | 0 | 0.0 | 0 | 0 | 0.0 | 0 | 0 | 0.0 |
| DDI-Fast | 71650 | 4207 | 92.1 | 33603 | 2073 | 90.8 | 16009 | 1028 | 90.0 |
| DDI-Full | 90329 | 4567 | 100.0 | 39374 | 2164 | 94.7 | 17912 | 1052 | 92.1 |
| DDI_Acc | | | | | | | | | |
| DDI-Socket | 0 | 0 | 0.0 | 0 | 0 | 0.0 | 0 | 0 | 0.0 |
| DDI-Fast | 1620 | 2684 | 25.0 | 810 | 671 | 12.5 | 404 | 168 | 6.3 |
| DDI-Full | 6480 | 10736 | 100.0 | 2684 | 2684 | 50.0 | 1620 | 671 | 25.0 |
| DDI_Put | | | | | | | | | |
| DDI-Socket | 0 | 0 | 0.0 | 0 | 0 | 0.0 | 0 | 0 | 0.0 |
| DDI-Fast | 3238 | 573 | 90.0 | 1619 | 281 | 88.1 | 809 | 139 | 87.4 |
| DDI-Full | 3238 | 637 | 100.0 | 1619 | 298 | 93.4 | 809 | 143 | 89.9 |

calculation still requires approximately 62 GB of DDI data transfers, i.e. *over 50 times more data is transferred to and from the DDI arrays than is stored in them*. The detailed profiling information for the distributed-memory operations in the benzoquinone benchmark is given in Table 2. The data, although similar to that presented previously by Fletcher et al. [4], augments their results by showing how the profile varies with numbers of processors used.

Consider first the combined socket and shared memory data transfers on 4 CPUs and 1 node of the HP SC, i.e. the rows entitled "All (via socket + shared memory)". This shows that based on the number of calls, the MP2 gradient is dominated by get operations; these exceed the sum of all accumulate and put operations by ~9:1. However, based on the total volume of data transferred, get operations account for only ~30%, while accumulates correspond to roughly 67%. The implication of this is that get operations involve a large number of relatively short data transfers (average 0.05MB), while accumulates involve a much smaller number of larger transfers (average 1.6MB). Put operations, on the other hand, are much smaller in number and have an average size (0.2MB) that lies between that of the get and accumulate operations.

For the get and put operations, increasing the number of processors from 4 to 16 results in a linear decrease in both the number of calls and volume of data transferred, indicating good scalability. For the accumulates, however, only the volume of data decreases while the number of calls remains constant at 6480. Thus, accumulates scale in that the data transfers decrease with the number of processors, but do not scale in the sense that regardless of the number of processors used, each GAMESS process will always require a fixed number of accumulates. This will ultimately limit scalability, which will also be impacted by the increasingly smaller average data transfer sizes making less efficient use of the underlying communication hardware.

Table 2 further breaks down the distributed-data operations by listing those portions that can be accomplished via shared-memory. The results given as "calls" correspond to the total number of GAMESS DDI calls that give rise to at least one shared memory data transfer, while the corresponding "mbytes" relates only to that fraction of the total data transfers that use shared memory. That is, if we consider a one to one mapping between compute and data sever processes, DDI calls can be classified as giving rise to exclusively local, exclusively remote, or mixed (global) data transfers [4]. The number of calls given in Table 2 when using shared memory is an aggregate of all the DDI calls that are either exclusively local or mixed, but the mbytes transferred is just that fraction of the total data

**Table 3:** Elapsed times (sec) for the entire GAMESS benchmark run using the various versions of DDI on a variety of platforms with a range of different numbers of CPUs and nodes.

| DDI Model | ——— Number of CPUs / Number of Nodes ——— | | | | | |
|---|---|---|---|---|---|---|
| **HP SC** | **1/1** | **2/1** | **4/1** | **8/2** | **16/4** | **32/8** |
| DDI-Full | 3568 | 1915 | 925 | 658 | 432 | 267 |
| DDI-Fast | 3532 | 2018 | 1081 | 759 | 520 | 354 |
| DDI-Socket | 4259 | 2284 | 1272 | 863 | 597 | 422 |
| DDI-SHMEM | N/A | 2340 | 1502 | 868 | 491 | 281 |
| DDI-Full (Modified)[a] | - | - | 925 | 688 | 501 | 348 |
| | | | | | | |
| **HP GS1280** | **1/1** | **2/1** | **4/1** | **8/1** | **16/1** | |
| DDI-Full | N/A | 1550 | 757 | 387 | 239 | |
| DDI-Fast | N/A | 1847 | 1407 | 1166 | 458 | |
| DDI-Socket | N/A | 2196 | 1796 | 1058 | 732 | |
| | | | | | | |
| **IBM Power3** | **1/1** | **2/1** | **4/1** | **8/2** | **16/4** | |
| DDI-Full | 7860 | 4033 | 2059 | 1161 | 635 | |
| DDI-Fast | 7933 | 4145 | 2180 | 1238 | 707 | |
| DDI-Socket | 8670 | 4434 | 2353 | 1351 | 774 | |
| | | | | | | |
| **Compaq** | **1/1** | **2/2** | **4/4** | **8/8** | **16/16** | |
| DDI-Fast/Full | N/A | 4650 | 2521 | 1352 | 739 | |
| DDI-Socket | N/A | 4338 | 2365 | 1393 | 688 | |

a) Specially modified version of DDI-Full to show effects of coalescing inter-node messages and data transfers. See text for details.

transferred by these calls that occurs via shared memory.

From the shared memory breakdown for DDI-Fast on 4 CPUs and 1 node of the HP SC it is evident that gets are heavily biased towards local transfers, with over 92% of the transfers (by volume) occurring via shared memory. A similar conclusion is reached for put operations, although in this case the difference between the number of DDI-Fast and DDI-Full calls (3238-637) shows that there are a substantial number of DDI put operations that give exclusively remote transfers but with relatively little data (637-573MB). Meanwhile accumulates are more evenly balanced, with DDI-Fast recording exactly one quarter of all accumulate calls and transferring one quarter of the total accumulate data transfers.

The profile results obtained with increased processor count further reflect the locality of the data transfers. That is, for both the get and put operations the fraction of transfers than can be achieved via shared memory remains relatively flat, while for accumulates it decreases linearly. This would suggest that the biggest advantage of DDI-full over DDI-sockets will be on one SMP node, and that this performance difference will decrease as SMP nodes are added. However, due to the locality of the get and put operations, there will always be some advantage in using DDI-Full over DDI-Socket regardless of the number of SMP nodes. Thus, for this benchmark as SMP nodes are added, the percentage of distributed-data operations that can be accomplished via shared-memory reaches a practical limit corresponding to ~90% of all get and put operations and essentially none of the accumulate operations; this translates to roughly 25-30% of all DDI operations.

Timing data on the various platforms is given in Table 3. This shows that the FAST and FULL implementations consistently outperform the original DDI implementations on SMP systems. The advantage of intra-node data transfers through shared-memory vs. the TCP/IP layer is best measured in the 1 CPU benchmark, where one sees an improvement of 727 seconds (~15%) on the HP SC. This improvement results from both faster transfer rates (via shared-memory copies) and lower latency, i.e. the time to acquire access via a semaphore vs. the time required by the OS to wake the data server. The importance of shared-memory transfers is further emphasized by the 4 CPU DDI-Full results which show a time reduction of ~25% compared to the original socket implementation. In this case, the original implementation has 4 compute processes stressing the TCP/IP layer (on the same machine) which are also competing for access to the 4 data servers. In the DDI-Full implementation, the data servers remain inactive and all the data is transferred via shared-memory.

On SMP nodes, where multiple distributed-memory segments exist within the same node, DDI-Full has two distinct advantages over DDI-Fast: i) the ability to

perform operations directly on all intra-node shared-memory segments and ii) the reduction in the number of communications needed to perform remote inter-node data operations. On the HP SC and in comparison with the original socket code, an improvement of 25%-35% in the total execution is measured for DDI-Full over a range of 1-8 nodes (4-32 processors), whereas DDI-Fast achieves only 10-15% improvement.

To ascertain the speedup associated with each of the above factors we modified DDI-Full so that all intra-node distributed-data operations are performed via shared-memory, while all remote data operations use separate socket connections for each remote data sever. The timing difference between this "modified" DDI-Full and DDI-socket is due to the use of shared-memory for all intra-node transfers, while that with DDI-Fast is due to the performance gained from using shared-memory for all intra-node shared-memory segments rather than just one. Meanwhile the difference between DDI-Full (Modified) and the original DDI-Full is due to both a reduction in inter-node communications (from one message per remote data server to just one) and coalescing of the associated data transfers into one long message instead of multiple shorter ones.

Timing results on the HP SC for DDI-Full (Modified) are also given in Table 3. On 2 nodes and 8 CPUs DDI-Full outperforms DDI-sockets by 205s. Half of this performance difference (104s) is accounted for by moving to DDI-Fast and allowing fast transfers to the local DDI sever. Comparing DDI-Fast with DDI-Full (Modified) shows that permitting fast transfers to all DDI servers on the same SMP node gives a further 71s improvement, while coalescing intra-node messages and data transfers gives rise to a 30s performance gain. Moving to 8 nodes and 32 processors changes this picture somewhat. Now the difference between DDI-socket and DDI-full is 155s; 68s of this difference is due to fast local data transfers, 6s to using fast data transfers to all shared memory segments on the same SMP node and 81s due to the coalescing of inter-node messages and data transfers. This result is probably not surprising, as reducing inter node data traffic will naturally become more important as the number of nodes being used is increased.

Examining the benefit of DDI-Full over DDI-socket on different platforms, we find improvements of 9, 9 and 12% on 1, 2 and 4 processors of the IBM 375 MHz Power3 system compared to 16, 16 and 27% on the HP SC. The larger percentage improvement on the SC is due to its significantly better CPU performance. This emphasizes the benefits that arise from communication improvements. This effect is even more apparent on the HP GS system where the performance improvements are 29, 58, 63 and 67% on 2, 4, 8 and 16 processors respectively. The HP GS has slightly faster processors than the SC, but considerably better memory bandwidth (12.6 GB/s per CPU compared to 1.6 GB/s per CPU) and also lower memory latency.

In some respects it can be argued that the FULL SMP data server model reduces to the SHMEM model on single node SMP systems. That is, with one SMP node the data servers in DDI-Full are essentially idle, consuming clock cycles during DDI initialization but soon after becoming dormant and remaining so for the rest of the calculation. In future implementations, we plan to remove the data servers entirely for single node parallel tasks. It is therefore somewhat surprising to find that the performance of DDI-SHMEM on 2 CPUs of the HP SC is significantly worse than any of the other DDI implementations. Furthermore, this difference becomes greater on 4 CPUs and 1 node. We suspect that this difference is partly due to the SHMEM library accessing the network interface card even though all SHMEM transfers occur solely within an SMP node. Indeed it is also interesting to note that the performance of DDI-SHMEM only begins to approach that of DDI-Full when using 32 CPUs across 8 nodes.

The one anomaly in Table 3 is the Compaq timings. This is the only cluster that is not an SMP system, and here the original DDI-socket code performs slightly better than the improved versions. At this writing it is not known whether this is a flaw in DDI-Fast/Full, or an anomaly that was caused by machine issues. This will be discussed in more detail at the conference.

## 5 Conclusions

In the paper we have presented the approach used by DDI to increase parallel efficiency on SMPs and clusters of SMPs for high-performance distributed data computations by maximizing the advantage of shared-memory. Considerable reductions in the communication overhead have been achieved for many different hardware platforms and communication fabrics.

It is important to emphasize that the accomplishments reported here have broad implications. While the results are presented in context of computational chemistry using the GAMESS package with the Distributed Data Interface, the approach is easily extended to other distributed-memory models in all areas of computational science and engineering.

## 6 Acknowledgments

The authors thank Drs. Ricky Kendall and Dmitri Fedorov for their many helpful discussions throughout the course of this work.

Funding for this work was provided by the Australian Partnership for Advanced Computing (APAC), by the US Department of Energy (USDOE) via a SciDAC (Scientific Discovery Through Advanced Computing) grant administered by the Ames Laboratory, by the Australia-America Fulbright

Association, and by a travel grant from the National Science Foundation (NSF).

Computational facilities were graciously provided by the APAC National Facility for the use of the HP SC and HP GS systems and the Scalable Computing Laboratory in the U.S. Dept. of Energy Ames Laboratory for the use of the IBM Power3 and Compaq clusters. The IBM Power3 cluster was provided by a Shared University Research grant to Iowa State University.

## 7 References

[1] M.W. Schmidt, K.K. Baldridge, J.A. Boatz, S.T. Elbert, M.S. Gordon, J.H. Jensen, S. Koseki, N. Matsunaga, K.A.Nguyen, S. Su, T.L. Windus, M. Dupuis, J.A. Montgomery. J. Comput. Chem. 14 (1993) 1347-1363.

[2] R.J. Harrison. Int. J. Quantum Chem. 40 (1991) 847-863.

[3] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, J. Dongarra. MPI – The Complete Reference, Vol 1, The MPI Core. (MPI Press, Cambridge, MA 1998).

[4] G.D. Fletcher, M.W. Schmidt, B.M. Bode, M.S. Gordon. Comm. Phys. Chem. 128 (2000) 190-200.

[5] R.J. Harrison. Theoret. Chim. Acta. 84 (1993) 363-375.

[6] J. Nieplocha, R.J. Harrison, R.J. Littlefield, in: Proceedings of Supercomputing 1994 (IEEE Computer Society Press, Washington D.C., 1994) p. 340.

[7] SHMEM Library Reference, Cray Research Incorporated (1994).

[8] G.F. Pfister. In Search of Clusters, 2nd ed. (Prentice-Hall, Upper Saddle River, N.J., 1998).

[9] Seifert, Rich, *Gigabit Ethernet: Technology and Applications for High Speed LANs* . Reading MA: Addison-Wesley, 1998.

[10] N.J. Boden, D. Cohen, R.E. Felderman, A.E. Kulawik, C.L. Seitz, J.N. Seizovic, W.K. Su. *Myrinet - A Gigabit-per-Second Local Area Network.* IEEE Micro. 15 (1995) 29-36.

[11] Hermann Hellwagner. The SCI Standard and Applications of SCI . In Hermann Hellwagner and Alexander Reinfeld, editors, SCI: Scalable Coherent Interface, volume 1291 of Lecture Notes in Computer Science, pages 95--116. Springer-Verlag, 1999.

[12] *Infiniband Architecture Specification Volume* 1, Release 1.0. Infiniband Trade Association, 2000.

[13] G.D. Fletcher, A.P. Rendell, P. Sherwood. Mol. Phys. 91 (1997) 431-438.

[14] G.D. Fletcher, M.W. Schmidt, M.S. Gordon. Adv. Chem. Phys. 110 (1999) 267-294.