Porting and Optimizing Performance of the Global Arrays Toolkit on the Cray X1

Vinod Tipparaju, Manojkumar Krishnan, Bruce Palmer, Jarek Nieplocha
Computational Sciences and Mathematics Department
Pacific Northwest National Laboratory
Richland, WA 99352

The Cray X1 represents an example of a scalable distributed-shared memory architecture; however, most programming models supported by Cray present it to application developers as a distributed memory or a global address space system. The Global Arrays (GA) toolkit supports a portable shared memory programming environment. With GA, the programmer can view distributed data structure as a single object and access it as if it resided in shared memory. This approach helps to raise the level of abstraction and program composition, as compared to the programming models with a fragmented memory view (e.g., MPI, Co-Array Fortran, SHMEM, and UPC). In addition to other application areas, GA is a widely used programming model in computational chemistry. We will describe how the GA toolkit is implemented on the Cray X1, and how the performance of its basic communication primitives were optimized based on the shared memory access capabilities of the hardware.

1. Introduction

The two predominant classes of programming models for MIMD concurrent computing are distributed memory and shared memory. Both memory models have advantages and shortcomings. Shared memory model is much easier to use but it ignores data locality/placement, which can negatively impact performance, while message passing is generally much more scalable but also much more difficult to program. The Global Arrays toolkit [2]-[4] provides a higher level abstraction for managing distributed data that attempts to offer the best features of both programming models. It implements a shared-memory model while at the same time allowing data locality to be managed by the programmer. This management is achieved by calls to functions that transfer data between a global address space (a distributed array) and local storage. In this respect, the GA model has similarities to the distributed shared-memory models that provide an explicit acquire/release protocol. However, the GA model acknowledges that remote data is slower to access than local data and allows data locality to be specified by the programmer. The GA model exposes to the programmer the hierarchical memory of modern high-performance computer systems [7], and by recognizing the communication overhead for remote data transfer, it promotes data reuse and locality of reference. The implementation strategies for the GA toolkit are not the same on all shared memory architectures. Instead, they exploit architecture-specific details like memory hierarchies and native network communication protocols.

Being the first of its kind to combine vector processing with DSM system architecture, the Cray X1 [1] has a few unique features in comparison with other shared memory architectures, such as the SGI Altix [23]. Although the Cray X1 architecture provides an excellent support for the GA shared memory programming model, the unique characteristics of this machine required us to consider a whole new realm of performance issues. As an example, although the X1 implements a directory based cache coherency protocol in hardware, similar to the Altix, it is different because it doesn't cache remote memory. The caching of only local memory makes the implementation of cache coherency more scalable but the tradeoff is that accessing remote memory now incurs latency. The current paper provides an overview of the functionality and performance of the Global Arrays toolkit with emphasis on its implementation the Cray X1. Although the GA model provides a much higher level of abstraction than MPI, the preliminary performance numbers we present for GA match or exceed that of MPI. The remainder of the paper is organized as follows. Sections 2 and 3 provide an overview of the Cray X1 and GA toolkit, respectively. Section 4 describes the main characteristics of ARMCI, a portable communication layer

that GA is implemented upon. Porting and optimization efforts of GA to the Cray X1 are described ion Section 5. Section 6 provides experimental results. Finally, the paper is concluded in Section 6.

2. Overview of the Cray X1

The Cray X1 combines the features of previous Cray vector systems and massively parallel-processing (MPP) systems into one design. Like the Cray T90, the X1 has high memory bandwidth. Along the lines of the Cray T3E, the X1 has a high-bandwidth, low-latency, scalable interconnect. The X1 design utilizes commodity CMOS technology and incorporates non-traditional vector concepts, like vector caches and multi-streaming processors. X1 addresses two of the major issues with large scale NUMA machines by providing a scalable cache coherency protocol and a scalable address translation mechanism.

The X1 has a hierarchy in processor, memory, and network design. It has two level processor hierarchies, MSP (multi-streaming processor) and SSP (single-streaming processor). Each MSP is comprised of 4 SSP's (Figure 1). Each MSP is capable of 12.8 GFlop/sec for 64-bit operations. Each SSP contains 32 vector registers holding 64 double-precision words and one 2-way super-scalar unit and can deliver 3.2 GFlop/sec. The SSP uses two clock frequencies, 800 MHz for the vector units and 400 MHz for the scalar unit. The four SSPs share a 2 MB "Ecache". The "Ecache" is a mechanism to fill the gap between the bandwidth to main memory and the operation rate of vector units. Each X1 processor is designed to have more single-stride bandwidth than an NEC SX-6 processor. Each node determines whether a memory reference is local to the node or whether the reference is to a memory location on a remote node. Remote memory references are made through the node interconnect. Each node contains 32 network ports, and each port supports 1.6 GBps peak per direction

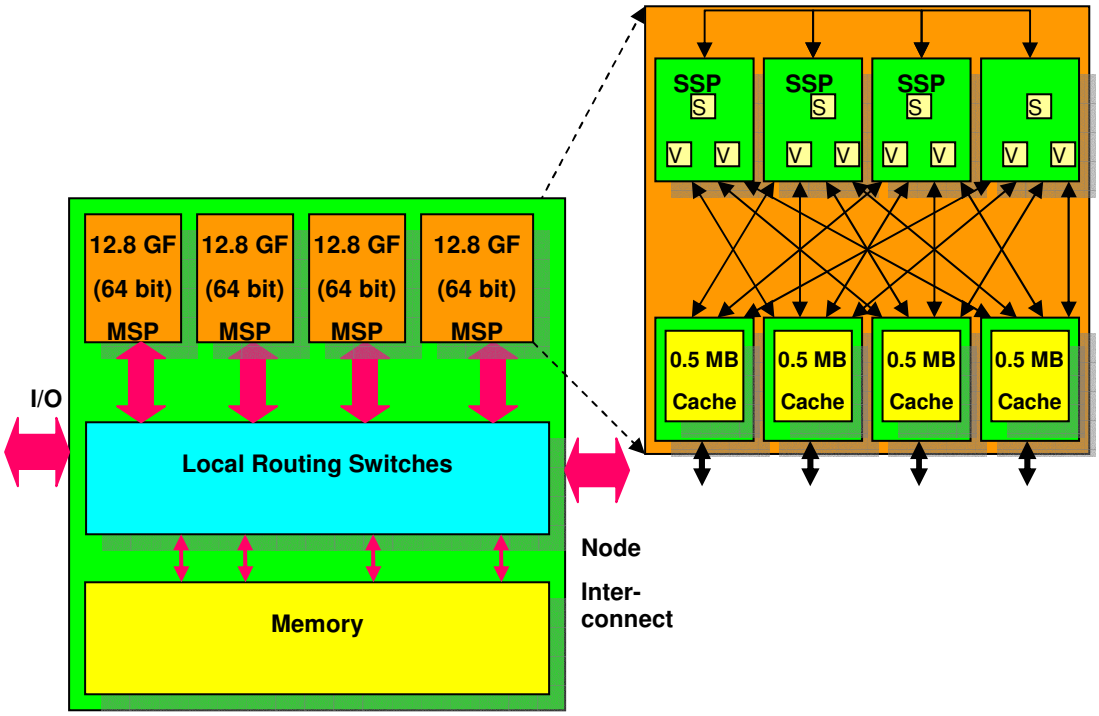

Figure 1: The X1 system, 4 MSP's per node and structure of each MSP

X1 runs the UNICOS/mp operating system. The UNICOS/mp operating system is based on the IRIX 6.5 operating system with Cray enhancements. It is designed to sustain the high-capacity, high-bandwidth throughput provided by Cray X1 systems. A single UNICOS/mp kernel image runs on the entire system or on each partition of the system. The Cray compilers automatically perform register allocation, scheduling, vectorization, and multistreaming to give optimal performance. Compiler vectorization provides loop level parallelization of operations and uses the high performance vector processing hardware. Multistreaming code

generation by the compiler permits tightly coupled execution across the four SSPs of an MSP on a block of code. As discussed later, this has its pros and cons. Concepts like vectorization and multistreaming can be intermixed, and multistreaming can extend beyond loop boundaries. The X1 compiler's primary strategies for using the eight vector units of a single MSP are parallelizing a (sufficiently long) vectorized loop or parallelizing an unvectorized outer loop. The effective vector length of the first strategy is 256 elements, the same as the NEC SX-6. The second strategy, which attacks parallelism at a different level, allows a much shorter vector length of 64 elements for a vectorized inner loop. Cray also has an option to support treating each SSP as a separate processor. X1 implements a completely new instruction set that supports both 64- and double-speed 32-bit operations and more and larger vector registers.

3. Overview of the Global Arrays

The classic message-passing paradigm of computer programming not only transfers data but also synchronizes the sender and receiver. Asynchronous (nonblocking) send/receive operations can be used to diffuse the synchronization point, but cooperation between sender and receiver is still required. The synchronization effect is beneficial in certain classes of algorithms, such as parallel linear algebra, where data transfer usually indicates completion of some computational phase; in these algorithms, the synchronizing messages can often carry both the results and a required dependency. For other algorithms, this synchronization can be unnecessary and undesirable, and a source of performance degradation and programming complexity. The MPI-2 one-sided model tries to relax some of the synchronization requirements between sender and receiver while imposing some constraints on progress and remote data access rules. Despite programming difficulties, the message-passing memory model maps well to the distributed-memory architectures used in scalable MPP systems. Because the programmer must explicitly control data distribution and is required to address data-locality issues, message-passing applications tend to execute efficiently on such systems. However, on systems with multiple levels of remote memory, for example networks of SMP workstations or computational grids, the message-passing model's classification of main memory as local or remote can be inadequate. A hybrid model that extends MPI with OpenMP attempts to address this problem but is very hard to use and often offers little advantage over the MPI-only approach.

In the shared-memory programming model, data is located either in "private" memory (accessible only by a specific process) or in "global" memory (accessible to all processes). In shared-memory systems, global memory is accessed in the same manner as local memory. Regardless of the implementation, the shared-memory paradigm eliminates the synchronization that is required when message-passing is used to access shared data. A disadvantage of many shared-memory models is that they do not expose the NUMA memory hierarchy of the underlying distributed-memory hardware. Instead, they present a flat view of memory, making it hard for programmers to understand how data access patterns affect the application performance or how to exploit data locality. Hence, while the programming effort involved in application development tends to be much lower than in the message-passing approach, the performance is usually less competitive. The shared memory model based on Global Arrays combines the advantages of a distributed memory model with the ease of use of shared memory. It is able to exploit SMP locality and deliver peak performance within the SMP by placing user's data in shared memory and allowing direct access rather than through a message-passing protocol.

The Global Arrays approach attempts to combine the best features of the shared and distributed memory models. It implements a shared-memory style programming model while still allowing the programmer to exploit data locality. This is achieved by function calls that provide information on which portion of the distributed data is held locally and the use of explicit calls to functions that transfer data between a global address space (a distributed array) and local storage. The combination of these functions allows users to make use of the fact that remote data is slower to access than local data and to optimize data reuse and minimize communication in their algorithms. Another advantage is that GA, by optimizing and moving only the data requested by the user, avoids issues such as false sharing or redundant data transfers present in some DSM solutions. The GA model exposes to the programmer the hierarchical memory of modern high-performance computer systems, and by recognizing the communication overhead for remote data transfer, it promotes data reuse and locality of reference.

The GA model allows the programmer to control data distribution and makes the locality information readily available to be exploited for performance optimization. For example, global arrays can be created by 1) allowing the library to determine the array distribution, 2) specifying the decomposition for only one array dimension and allowing the library to determine the others, 3) specifying the distribution block size for all dimensions, or 4)

specifying an irregular distribution as a Cartesian product of irregular distributions for each axis. The distribution and locality information is always available through interfaces to query 1) which data portion is held by a given process, 2) which process owns a particular array element, and 3) a list of processes and the blocks of data owned by each process corresponding to a given section of an array.

The primary mechanisms provided by GA for accessing data are copy operations that transfer data between layers of memory hierarchy, namely global memory (distributed array) and local memory. In addition, each process is able to access directly data held in a section of a Global Array that is locally assigned to that process and on SMP clusters sections owned by other processes on the same node. Atomic operations are provided that can be used to implement synchronization and assure correctness of an accumulate operation (floating-point sum reduction that combines local and remote data) executed concurrently by multiple processes and targeting overlapping array sections.

GA is extensible as well. New operations can be defined exploiting the low level interfaces dealing with distribution, locality and providing direct memory access. These were used to provide additional linear algebra capabilities by interfacing with third party libraries e.g., ScaLAPACK. The GA model, when implemented in library form, can be used in C, C++, Fortran 77, Fortran 90 and Python programs.

In shared memory programming, one of the issues central to performance and scalability is memory consistency. Although the sequential consistency model [10] is straightforward to use, weaker consistency models [11] can offer higher performance on modern architectures and they have been implemented on actual hardware. The GA approach is to use a weaker than sequential consistency model that is still relatively straightforward to understand by an application programmer. The main characteristics of the GA approach include:

- GA distinguishes two types of completion of the store operations (i.e., put, scatter) targeting global shared memory: local and remote. The blocking store operation returns after the operation is completed locally, i.e., the user buffer containing the source of the data can be reused. The operation completes remotely after either a memory fence operation or a barrier synchronization is called. The fence operation is required in the critical sections of the user code, if the globally visible data is modified.
- The blocking operations (load/stores) are ordered only if they target overlapping sections of global arrays. Operations that do not overlap or access different arrays can complete in arbitrary order.
- The nonblocking load/store operations complete in arbitrary order. The programmer uses wait/test operations to order completion of these operations, if desired.

3.1 Capabilities

There are three classes of operations in the Global Array toolkit: core operations, task parallel operations, and data parallel operations. These operations have multiple language bindings that provide the same functionality independently of the language. The GA package has grown considerably in the course of ten years. The current library contains over 150 operations that provide a rich set of functionality related to data management and computations involving dense, distributed arrays.

Core operations

The basic components of the Global Arrays toolkit are function calls to create Global Arrays, copy data to, from, and between Global Arrays, and identify and access the portions of the Global Array data that are held locally. There are also functions to destroy arrays and free up the memory originally allocated to them. The **`create`** function returns an integer handle that can be used to reference the array in all subsequent calculations. The allocation of data can be left completely to the toolkit, but if it is desirable to control the distribution of data for load balancing or other reasons, additional versions of the **`create`** function are available that allow the user to specify in detail how data is distributed between processors.

Task parallel operations

One of the most important features of the Global Arrays toolkit is the ability to easily move blocks of data between Global Arrays and local buffers. The data in the Global Array can be referred to using a global indexing scheme and data can be moved in a single function call, even the data is distributed over several processors. The **`get`** function can be used to move a block of distributed data from a Global Array to a local buffer. The only arguments that need to be specified for this are two arrays with bounding indices of the Global Array data and

the local buffer. The **put** call is similar and can be used to move data in the opposite direction. For a distributed data paradigm with message-passing, this kind of operation is much more complicated. The block of distributed data that is being accessed must be decomposed into separate blocks, each residing on different processors, and separate message-passing events must be set up between the processor containing the buffer and the processors containing the distributed data.

Finally, to allow the user to exploit data locality, the toolkit provides functions identifying the data from the Global Array that is held locally on a given processor. Two functions are used to identify local data. The first is the **distribution** function, which returns a set of lower and upper indices in the global address space representing the local data block. The second is the **access** function, which returns an array index and an array of strides to the locally held data. In Fortran, this can be converted to an array by passing it through a subroutine call. The C interface provides a function call that directly returns a pointer to the local data.

Data parallel operations

In addition to the communication operations that support task-parallelism, the GA toolkit includes a set of interfaces that operate on either entire arrays or sections of arrays in the data parallel style. These are collective operations that are called by all processes in the parallel job. For example, movement of data between different arrays can be accomplished using a single function call. The **copy_patch** function can be used to move a patch, identified by a set of lower and upper indices in the global index space, from one Global Array to a patch located within another Global Array.

A simple code fragment illustrating how these routines can be used is shown in Figure 2. A 1-dimensional, distributed array is created and initialized and then inverted so that the entries are running in the opposite order. The locally held piece of the array is copied to a local buffer, the local data is inverted, and then it is copied back to the inverted location in the global array. The chunk array specifies minimum values for the size of each

```
integer ndim, nelem
parameter (ndim=1, nelem=100)
integer dims, chunk, nprocs, me, g_a, g_b
integer a(nelem), b(nelem)
integer i, lo, hi, lo2, hi2, ld

me = ga_nodeid()       ! rank of the process
nprocs = ga_nnodes() ! total # of processes

dims = nprocs*nelem
chunk(1) = nelem
ld = nelem

call nga_create(MT_INT, ndim, dims,
                  'array A', chunk, g_a)
call nga_duplicate(g_a, g_b, 'array B')
!  INITIALIZE DATA IN GA (NOT SHOWN)
call nga_distribution(g_a, me, lo, hi)

call nga_get(g_a, lo, hi, a, ld)
! INVERT LOCAL DATA
do i = 1, nelem
   b(i) = a(nelem+1-i)
end do
! INVERT DATA GLOBALLY
lo2 = dims + 1 – hi
hi2 = dims + 1 – lo
call nga_put(g_b, lo2, hi2, b, ld)
```

```
#define   NDIM      1
#define   NELEM   100
int dims, chunk, nprocs, me, g_a;
int a[NELEM],b[NELEM];
int i, lo, hi, lo2, hi2, ld;
GA::GlobalArray *g_a, *g_b;

me       = GA::SERVICES.nodeid();
nprocs = GA::SERVICES.nodes();

dims   = nprocs*NELEM;
chunk = ld = NELEM;

// create a global array
g_a = GA::SERVICES.createGA(C_INT, NDIM,
                            dims, "array A", chunk);
g_b = GA::SERVICES.createGA(g_a, "array B");
// INITIALIZE DATA IN GA (NOT SHOWN)
g_a->distribution(me, lo, hi);
g_a->get(lo, hi, a, ld);
// INVERT DATA LOCALLY
for (i=0; i<nelem; i++)    b[i] = a[nelem-1 – i];
// INVERT DATA GLOBALLY
lo2 = dims – 1 – hi;
hi2 = dims – 1 – lo;
g_b->put(lo2,hi2,b,ld);
```

Figure 2: Example Fortran (left) and C++ (right) code for transposing elements of an array

locally held block and in this example guarantees that each local block is a 100 integer array. Note that global indices are used throughout and that it is unnecessary to do any transformations to find the local indices of the data on other processors.

The Global Array toolkit also contains a broad spectrum of elementary functions that are useful in initializing data or performing calculations. These include operations that zero all the data in a global array, uniformly scale the data by some value, and support a variety of (BLAS-like) basic linear algebra operations including addition of arrays, matrix multiplication, and dot products. The global array matrix multiplication has been reimplemented over the years, without changing the user interface. These different implementations ranged from a wrapper to the SUMMA [9] algorithm to the more recently introduced high-performance SRUMMA algorithm [12] that relies on a collection of protocols (shared memory, remote memory access, nonblocking communication) and techniques to optimize performance depending on the machine architecture and matrix distribution.

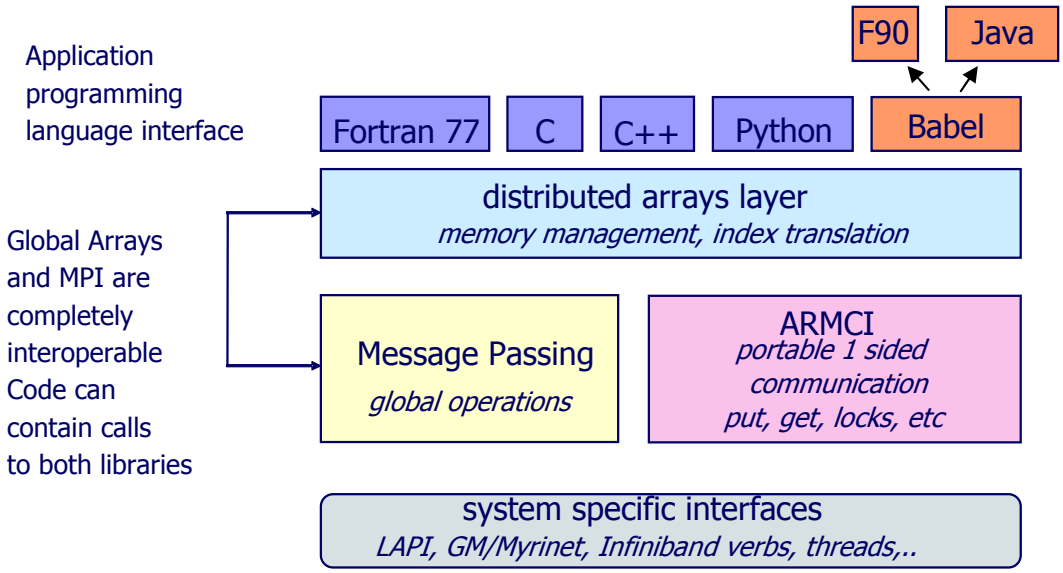

Figure 3: Diagram of Global Arrays showing the overall structure of the toolkit and emphasizing different language bindings

4. ARMCI

GA uses ARMCI (Aggregate Remote Memory Copy Interface) [8] as the primary communication layer. In addition, some collective operations are supported with MPI. Neither GA nor ARMCI can work without a message-passing library that provides the essential services and elements of the execution environment (job control, process creation, interaction with the resource manager).The fundamental model of Single Program Multiple Data (SPMD) program is inherited from MPI, along with the overall execution environment and services provided by the operating system to the MPI programs. ARMCI is currently a component of the run-time system in the Center for Programming Models for Scalable Parallel Computing project [13]. In addition to being the underlying communication interface for Global Arrays, it has been used to implement communication libraries such as Generalized Portable SHMEM [14], and compiler run-time systems such as PCRC Adlib [8] and the portable Co-Array Fortran compiler at Rice University [6]. ARMCI offers an extensive set of functionality in the area of RMA communication: 1) data transfer operations; 2) atomic operations; 3) memory management and synchronization operations; and 4) locks. Communication in most of the non-collective GA operations is implemented as one or more ARMCI communication operations. ARMCI was designed to be a general, portable, and efficient one-sided communication interface that is able to achieve high performance [15]-[19]. It also avoided the complexity of the progress rules and increased synchronization in the MPI-2 one-sided

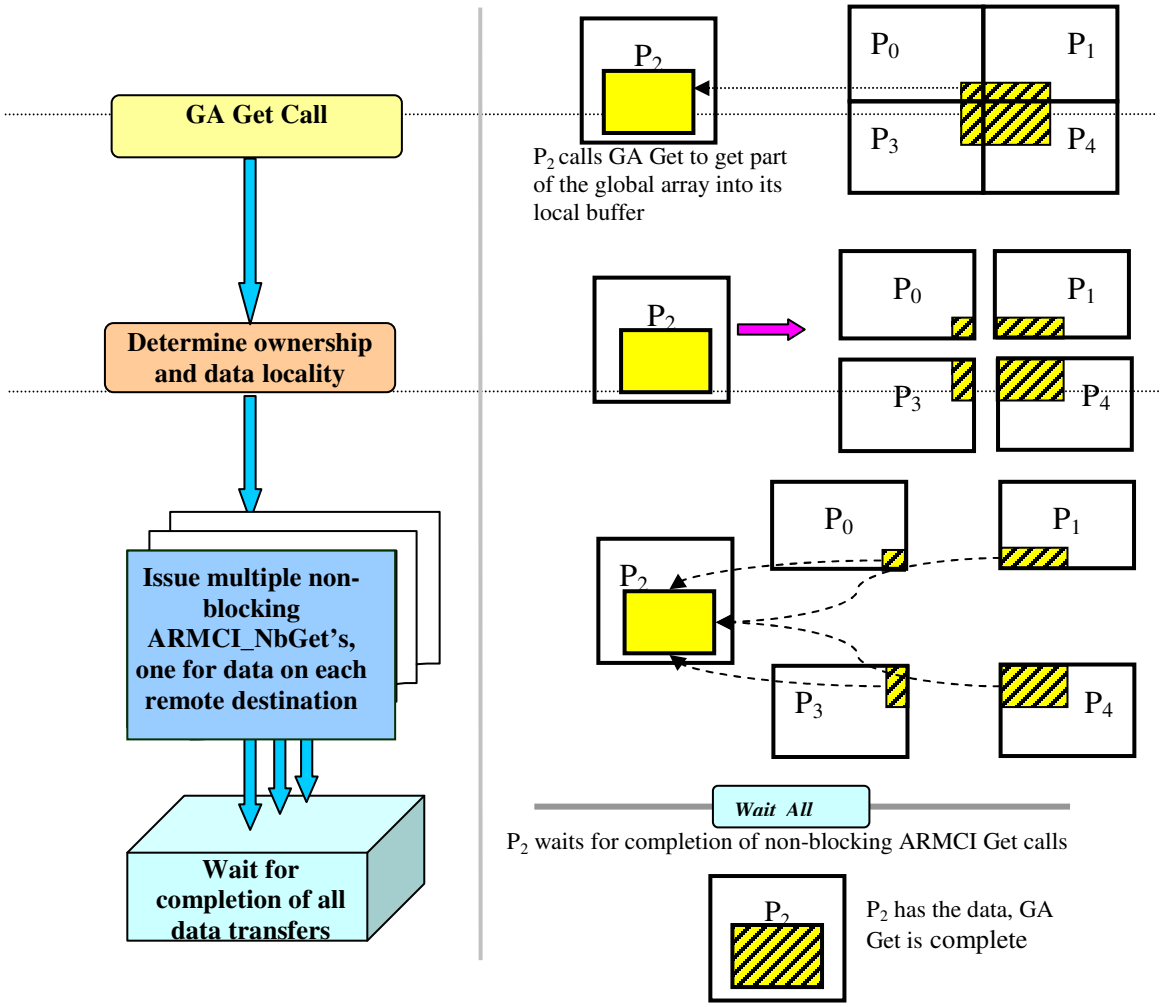

Figure 4: Schematic representation of the portable implementation of the GA_Get operation. Left: *GA_Get* flow chart. Right: An example: Process P_2 issues *GA_Get* to get a chunk of data, which is distributed (partially) among P_0, P_1, P_3 and P_4 (owners of the chunk). Since the Cray X1 does not provide hardware support for nonblocking communication thus ARMCI nonblocking calls turn into blocking ones.

model (introduced in 1997), that has been responsible for its delayed implementations and still rather limited adoption by the applications (as of 2004).

With the exception of the global array index translation layer, GA's communication interfaces form a thin wrapper around ARMCI interfaces. The performance of GA is therefore proportionate to and in-line with ARMCI performance. ARMCI is also a mechanism to address one of the most important requirements - portability. GA achieves most of its portability by relying on ARMCI. This reduces the effort associated with porting GA to a new platform to porting ARMCI.

GA relies on the global memory management provided by ARMCI. ARMCI requires that remote memory be allocated via the memory allocator routine, ARMCI_Malloc (similar to MPI_Win_malloc in MPI-2). On shared memory systems, including the Cray X1, this approach makes it possible to allocate shared memory for the user data and consecutively map remote memory operations to direct memory references, thus achieving sub-microsecond latency and a full memory bandwidth [19]. To support efficient communication in the context of multi-dimensional arrays, GA requires and utilizes the non-contiguous vector and multi-strided primitives provided by ARMCI [8].

5. Porting and Optimization of GA to the Cray X1

On any platform, the data parallel operations in GA are implemented on top of task parallel operations such as **put/get**, message-passing collective operations, and direct access to GA data stored on the local processor. In turn, GA task parallel operations are implemented using ARMCI one-sided communication operations for the data transfers, see Figure 4. Hence most of the porting activity related to data movement operations has been done at the ARMCI level. In the case of the Cray X1, the implementation is derived from the existing port to generic shared memory systems. The user memory is allocated through the ARMCI_Malloc collective interface that calls system operation *malloc* to allocate memory and then exchanges addresses between participating processes using MPI. This technique allows us to avoid the limitations of the Cray SHMEM library regarding the symmetric addresses for the user data. For more efficient operation to minimize the number of system calls, ARMCI uses an intermediate heap manager that uses the "first-fit" scheme for memory management.

Since the ARMCI **put/get** model maps directly to the load and store operations, implementation of these operations is trivial for contiguous operations: it simply maps to the Cray memory copy operation: *memcpy*. The noncontiguous **put/get** operations are implemented as a (sequential) Fortran code for copying the noncontiguous data. The Cray compiler is able to generate an efficient vectorized code.

However, the hierarchy in the memory model of X1 and the fact that it does not cache remote data needed to be considered when optimizing some operations and core GA kernels like the matrix multiplication. From the system architect perspective, the advantage of not caching remote shared memory is that implementing scalable system wide cache coherency becomes simpler. From the application perspective, the main implication of this approach is that since remote data (outside the SMP node) is not cached, the programmer is responsible for deciding when remote data should be brought to the cache (through a an explicit data copy to a local buffer) and implementing the explicit data movement. Providing this level of control to the programmer is advantageous since without prior knowledge of the algorithm the system cannot always make the right decision (remote data should not be cached if it is not reused), however there are no mechanisms for offloading the explicit memory copy from the application to the system. The X1 does not offer any hardware support for block-based prefetching or poststoring which would be required for explicit overlapping communication with computations. Comparing to other architectures, although the cost of the explicit memory copy cannot be fully hidden, its impact on the application performance is often reduced thanks to the fact that the network interconnect bandwidth is relatively high.

Since the ARMCI model maps very well to the Cray X1 architecture and achieved performance is as good as the Cray *memcpy* operation, the main effort in optimizing performance of Global Arrays was focused on the latency of one-sided operations. In the initial port, the translation of global array indices to the processor and memory location took a disproportional larger fraction of time than on any other modern system. The reason for this is the relatively low performance of the scalar processor of the X1. Another significant implementation consideration is that by default the compiler on X1 generates multi-streaming code to efficiently run on the MSP (Multi-Streaming processor). This allows streaming of a block of code across 4 SSP's (Single-Streaming processor). The advantage with streaming a block of code across 4 units is that the amount of instruction level parallelism that can be achieved is very high. This can greatly improve the performance of a code that has loops with a lot of depth and hence is recommended for loops with a large loop count. However, this is not ideal for the parts of the code that have small loops. There are tradeoffs between multistreaming some parts of the code for which this technique is beneficial and others where the overhead of multistreaming is relatively high. As a result, the best approach is to enable multistreaming for most of the code and selectively disable it for the parts where it degrades performance. For example, by disabling multistreaming in a code containing both large and small loops, the latency of the **ga_get** operation dropped from 24 to 19 microseconds. However, by compiling the entire code with multistreaming enabled and then selectively turning multistreaming off using pragmas in a few short loops involved in the global array index translation code, the latency dropped from 24 to 8.4 microseconds. Despite achieving this substantial improvement, we are pursuing code optimization in other parts of the global array index translation code to reduce the latency even further.

5.1 Experimental results

We attempted to evaluate the effectiveness of our implementation on X1 with a few micro benchmarks and an application benchmark.

Micro Benchmarks

The motivation for the experiments described in this section was to demonstrate the performance of the implementation at the system level. The next section shows how much of these gains can be leveraged at the application level. Experiments discussed in the current section have been conducted for the GA **`put`** and **`get`** operations. The experimental results shown in Figures 5 and 6 compare performance of the GA and MPI and ARMCI operations. In principle, GA is tracking closely the performance of ARMCI. However, the performance model of GA also includes the cost of a global array index translation that requires analyzing the distribution information. That cost component is independent of the request size and is sensitive to the scalar performance of the CPU.

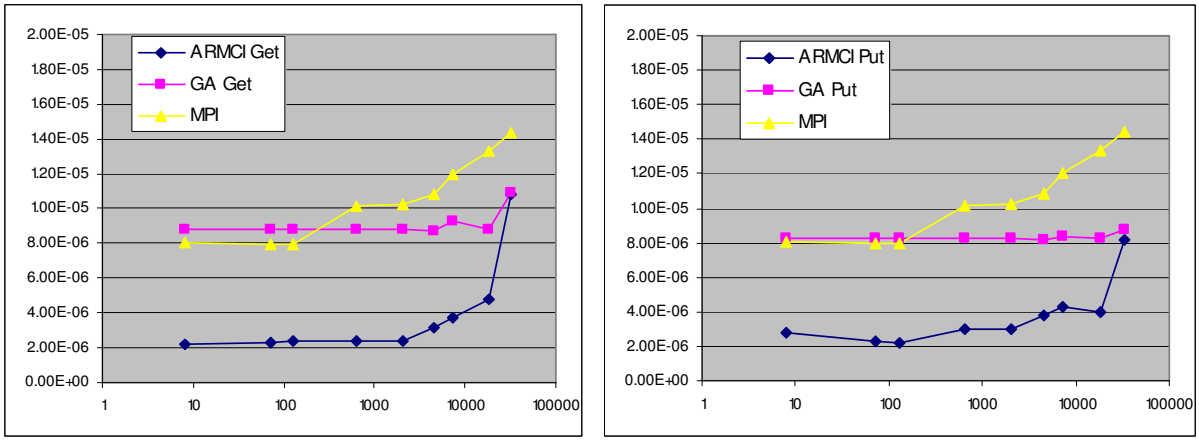

Figure 5: GA Get and GA Put latency in comparison to ARMCI Get/Put latency and MPI Send/Recv latency

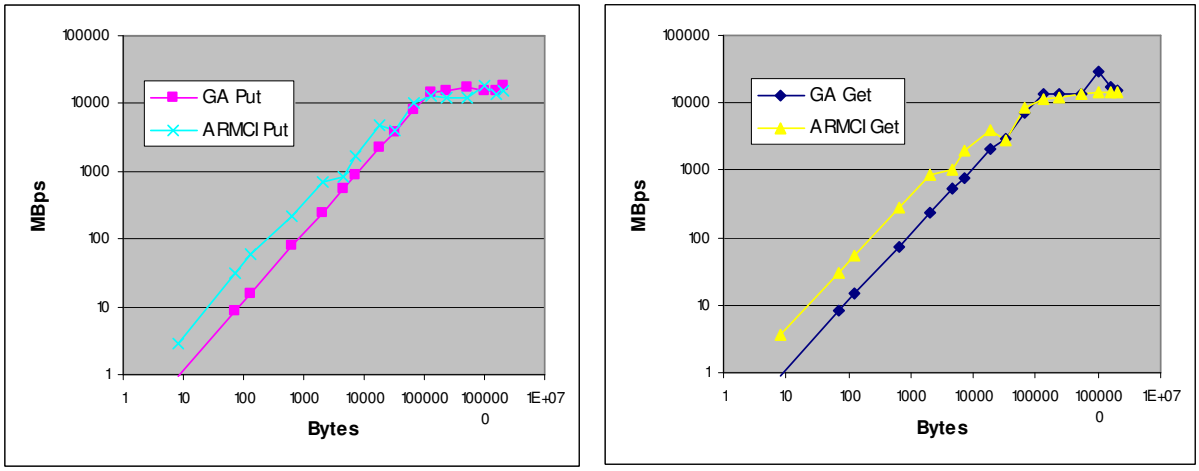

Figure 6: GA Get and GA Put bandwidth in comparison to ARMCI Get/Put bandwidth.

Application Benchmark

The ability of Global Arrays to manage large distributed arrays has made the toolkit useful in many application areas. Application that requires large, dense, multi-dimensional data grids or arrays can make use of Global Arrays. Examples are algorithms and applications built around dense matrices (e.g. electronic structure) and algorithms on multi-dimensional grids (hydrodynamics and other continuum simulations). Even algorithms that are based on sparse data structures (unstructured grids in finite element codes) often convert the original sparse data to a dense 1-dimensional array. For most of these applications it remains attractive to treat these data structures as single arrays using a global index space that maps directly onto the original problem. The GA toolkit provides mechanisms for identifying what data is held locally on a processor, allowing programmers to

make use of data locality when designing their programs, and even provides direct access to the data stored in the Global Array, which saves on the time and memory costs associated with duplication.

Molecular dynamics (MD) is a computer simulation technique where the time evolution of a set of interacting atoms is followed by integrating their equations of motion. For this application, the force between two atoms is approximated by a Lennard-Jones potential energy function *U(r)*, where *r* is the distance between two atoms. Good performance and scalability in the application require an efficient parallel implementation of the objective function and gradient evaluation. These routines were implemented by using GA to decompose the atoms over the processors and distribute the computation of forces in an equitable manner. The decomposition of forces between atoms is based on a block decomposition of the forces distributed among processors, where each processor computes a fixed subset of inter-atomic forces [20]. The entire array of forces (N x N) is divided into multiple blocks (m x m), where m is the block size and N is the total number of atoms. Each process owns N/P atoms, where P is the total number of processors. Exploiting the symmetry of forces between two particles halves the amount of computation. The force matrix and atom coordinates are stored in a global array. A centralized task list is maintained in a global array, which stores the information of the next block that needs to be computed.

To address the potential load imbalance in our test problem, we use simple and effective dynamic load-balancing technique called fixed-size chunking [21]. This is a good example illustrating the power of shared memory style management of distributed data that makes the GA implementation both simple and scalable. Initially, all the processes get a block from the task list. Whenever a process finishes computing its block, it gets the next available block from the task list. Computation and communication are overlapped by issuing a nonblocking get call to the next available block in the task list, while computing a block [22]. This implementation of the dynamic load-balancing technique takes advantage of the atomic and one-sided operations in the GA toolkit (see Figure 7). The GA one-sided operations eliminate explicit synchronization between the processor that executes a task and the processor that has the relevant data. Atomic operations reduce the communication overhead in the traditional message-passing implementations of dynamic load balancing based on the master-slave strategy.

The experimental results of the molecular dynamics benchmark on a X1 indicate that using the GA model not only eased the programming part of the problem but also resulted in improved performance over message-passing, see Figure 8. This benchmark problem scales well when the number of processors and/or the problem size is increased, thus proving the solution is cost-optimal.

```
Get task (i.e., block info) to be computed ( atomic fetch-and-add)
Issue nonblocking get call for the first block
do (until last block/task)
   determine what the next block/task is
   issue nonblocking get call for the next block
   wait for previously issued get call
   compute Function-Gradient
 (overlapping communication with computation.
i.e., receiving next block while computing previous block)
   accumulate function and gradient into respective Arrays
done
```

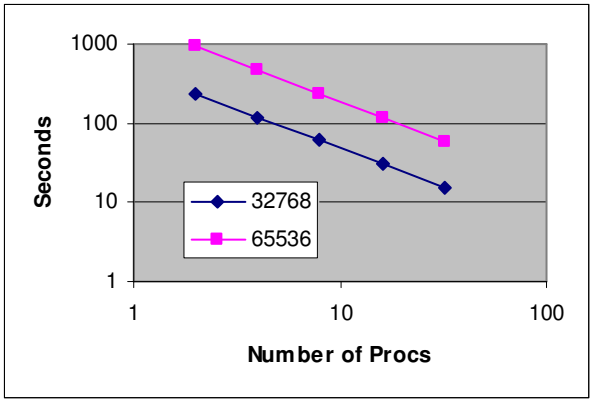

Figure 7: Function gradient evaluation using Global Arrays (left). Time consumed by Lennard-Jones potential energy calculation for 32,768 and 65,536 atoms (right).

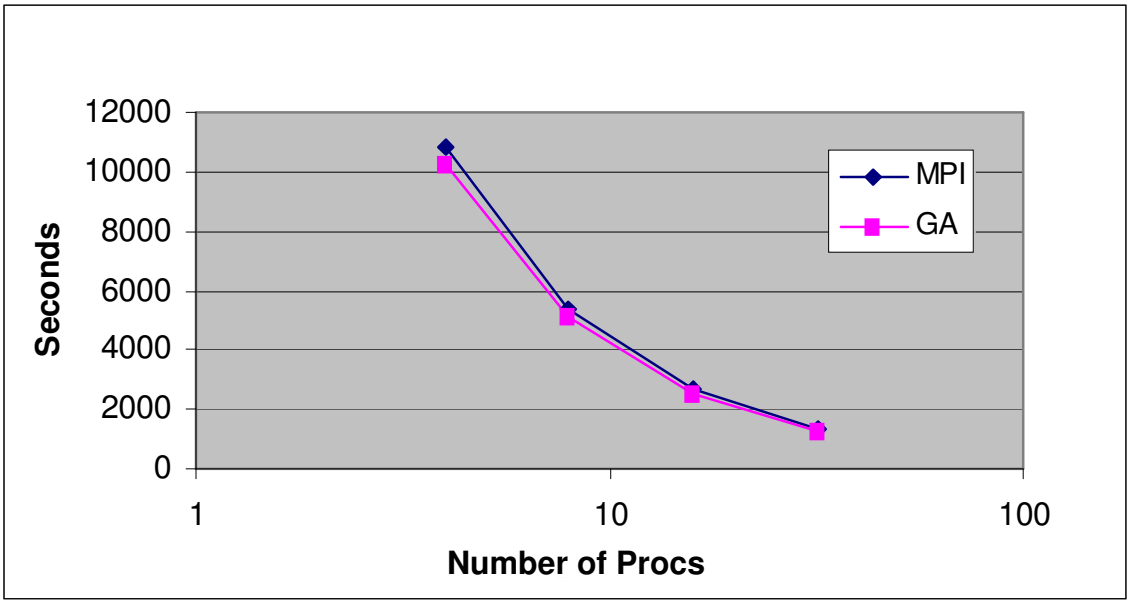

Figure 7: Performance the molecular dynamics simulation involving 314432 atoms

6. Conclusions

This paper provided an overview of the functionality and performance of the Global Arrays toolkit with emphasis on its implementation the Cray X1. The overall architecture of the Cray X1 is an excellent match for the GA programming model. Although the machine offers very high bandwidth, GA applications would benefit from availability of the hardware support for nonblocking communication as a mechanism for hiding data transfer costs. The current implementation was designed to optimize communication and minimize the influence of differences in memory access costs for different levels of memory hierarchy. Due to the relatively low performance of the scalar processor, the main effort in optimizing performance involved addressing compilation issues that affected the latency. Optimizations techniques that helped us to reduce the latency by a factor of three included copying global variable to local variables and selectively enabling and disabling multistreaming. Although the current GA bandwidth exceeds that of MPI and the latency matches MPI, we believe that further improvements of the latency numbers are still possible.

7. Acknowledgements

This work was performed under the auspices of the U.S. Department of Energy (DOE) at Pacific Northwest National Laboratory (PNNL) operated for DOE by Battelle Memorial Institute. It was supported by the DoE-2000 ACTS project, Center for Programming Models for Scalable Parallel Computing, both sponsored by the Mathematical, Information, and Computational Science Division of DOE's Office of Computational and Technology Research, and the Environmental Molecular Sciences Laboratory. The Global Arrays toolkit would not exist without invaluable contributions from many computational scientists who provided requirements, feedback, and encouragement for our efforts and in some cases even their direct involvement in the GA development. PNNL is operated by Battelle for the U.S. DOE under Contract DE-AC06-76RLO-1830.

8. References

[1] Cray X1 System http://www.cray.com/products/systems/x1

[2] J. Nieplocha, R.J. Harrison and R.J. Littlefield, "Global Arrays: A Portable Shared Memory Programming Model for Distributed Memory Computers", Proc. Supercomputing'94, IEEE CS Press, pp. 340-349, 1994.

[3] J. Nieplocha, R.J. Harrison, R.J. Littlefield, "Global Arrays: A Nonuniform Memory Access Programming Model for High-Performance computers", The Journal of Supercomputing, vol 10, pp. 197-220, 1996.

[4] J. Nieplocha, R. Harrison, M. Krishnakumar, B. Palmer, V. Tipparaju, "Combining shared and distributed memory models: Evolution and recent advancements of the Global Array Toolkit", Proc. POHLL'2002 workshop of ICS-2002, NYC, 2002.

[5] William W. Carlson and Jesse M. Draper and David E. Culler and Kathy Yelick and Eugene Brooks and Karen Warren, Introduction to {UPC} and Language Specification, Center for Computing Sciences CCS-TR-99-157, 1999.

[6] Robert W. Numrich and John K. Reid, ACM Fortran Forum, 2, 1-31, 1998.

[7] J. Nieplocha, R.J. Harrison, and I. Foster, "Explicit Management of Memory Hierarchy", in "Advances in High Performance Computing", Eds. J. Kowalik, L. Grandinetti, and M. Vajtersic, pages 185-200, Kluwer Academic, 1997.

[8] J. Nieplocha, B. Carpenter, ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-time Systems, *Proc. RTSPP of IPPS/SDP'99*, 1999.

[9] R. van de Geijn, R. and J. Watts, "SUMMA: Scalable Universal Matrix Multiplication Algorithm," *Concurrency: Practice and Experience*, vol. 9(4), pp. 255–274, 1997.

[10] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In Proceedings of the 14th Annual International Symposium on Computer Architecture, pages 234-243, June 1987.

[11] Michel Dubois, Christoph Scheurich, and Faye´ Briggs. Memory access buffering in multiprocessors. In Proceedings of the 13th Annual International Symposium on Computer Architecture, pages 434-442, June 1986.

[12] M. Krishnan, and J. Nieplocha, A Matrix Multiplication Algorithm Suitable for Clusters and Scalable Shared Memory Systems, accepted for International Parallel and Distributed Computing Symposium IPDPS'04. Santa Fe, NM. 2004.

[13] Center for Programming Models for Scalable Parallel Computing. www.pmodels.org

[14] K. Parzyszek, J. Nieplocha, R.A. Kendall, A generalized portable SHMEM library for high performance computing, Proc. Parallel and Distributed Computing and Systems PDCS-2000, pp. 401-406, 2000.

[15] J. Nieplocha, V. Tipparaju, A. Saify, D. Panda, Protocols and Strategies for Optimizing Remote Memory Operations on Clusters, Proc. CAC'02workshop of IPDPS'02. 2002.

[16] J. Nieplocha, V. Tipparaju, J. Ju, E. Apra, One-sided communication on Myrinet, Cluster Computing, 6, 115-124, 2003.

[17] V. Tipparaju, G. Santhmaraman, J. Nieplocha, D.K. Panda, Host-assised zero-copy remote memory access communication on Infiniband, accepted for International Parallel and Distributed Computing Symposium, IPDPS'04. Santa Fe, NM. 2004.

[18] J. Nieplocha, J. Ju, T.P. Straatsma, A multiprotocol communication support for the global address space programming model on the IBM SP, in A. Bode et al (Eds.). Proc. Euro-Par 2000 Parallel Processing, Springer Verlag LNCS 1900, 2000.

[19] J. Nieplocha, J. Ju, ARMCI: A Portable Aggregate Remote Memory Copy Interface, Version 1.1, October 30, 2000. http://www.emsl.pnl.gov/docs/parsoft/armci/armci1-1.pdf.

[20] S. J. Plimpton, "Scalable Parallel Molecular Dynamics on MIMD supercomputers," in *Proceedings of Scalable High Performance Computing Conference-92*, pages 246-251. IEEE Computer Society Press, 1992.

[21] C. Kruskal and A. Weiss, "Allocating Independent Subtasks on Parallel Processors," *IEEE Transactions on Software Engineering*, SE-11 (10), pp. 1001-1016, 1985.

[22] V. Tipparaju, M. Krishnan, J. Nieplocha, G. Santhanaraman, D. Panda, Exploiting Non-blocking Remote Memory Access Communication in Scientific Benchmarks, Proc. International Conference on High Performance Computng, HiPC'2003

[23] SGI Altix family of servers and super clusters http://www.sgi.com/servers/altix/