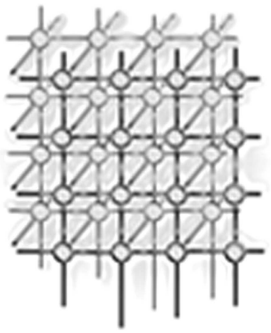
CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2009; **00**:1–7 *Prepared using* cpeauth.cls *[Version: 2002/09/19 v2.02]*



Optimizing communication for Charm++ applications by reducing network contention

Abhinav Bhatelé †, Eric Bohm, Laxmikant V. Kalé *

201, North Goodwin Avenue, MC-258,
Department of Computer Science
University of Illinois at Urbana-Champaign, Urbana, IL 61801, U.S.A.

SUMMARY

Optimal network performance is critical for efficient parallel scaling of communication-bound applications on large machines. No-load latencies do not increase significantly with number of hops traveled when wormhole routing is deployed. Yet, we and others have recently shown that in presence of contention, message latencies can grow substantially large. Hence task mapping strategies should take the topology of the machine into account on large machines. In this paper, we present topology aware mapping as a technique to optimize communication on three-dimensional mesh interconnects and hence improve performance.

Our methodology is facilitated by the idea of object-based decomposition used in Charm++ which separates the processes of decomposition from mapping of computation to processors and allows a more flexible mapping based on communication patterns between objects. Exploiting this and the topology of the allocated job partition, we present mapping strategies for a production code, OpenAtom to improve overall performance and scaling. OpenAtom presents complex communication scenarios of interaction involving multiple groups of objects and makes the mapping task a challenge. Results are presented for OpenAtom on up to 16,384 processors of Blue Gene/L, 8,192 processors of Blue Gene/P and 2,048 processors of Cray XT3.

KEY WORDS: Mapping; Performance Optimization; Mesh Topology

INTRODUCTION

A significant number of the largest supercomputers in use today, including IBM's Blue Gene family and Cray's XT family, employ a 3D mesh or torus topology. With tens of thousands of

*Correspondence to: 201, N. Goodwin Ave., Urbana, IL 61801, U.S.A.
†E-mail: bhatele@illinois.edu
Contract/grant sponsor: DOE; contract/grant number: B341494, DE-FG05-08OR23332

Received October 2009
Revised March 2010

nodes, messages may have to travel many tens of hops before arriving at their destinations. With the advances in communication technology, especially wormhole routing, it was observed that the *latency* of communication in absence of contention, was almost unaffected by the number of hops traveled by a message [1, 2]. However, the fraction of *bandwidth* occupied by the message is proportional to the number of hops (links) traversed. Increased contention for bandwidth results in longer latencies. This was not as significant an issue when the number of nodes was relatively small and the processors were slower. But for today's large machines with faster processors, the issue becomes much more significant. With faster processors, the need for delivered bandwidth is higher. As messages start to occupy a large fraction of the available bandwidth, the contention in the network increases and message delivery gets delayed [3].

In this context, it is important to map computation to the processors to not just minimize the overall communication volume, but also the average number of hops traveled by the bytes communicated. Even though the utility of doing this may be apparent to programmers, the significance of the impact is probably more than most programmers expect. Our methodology builds upon object-based decomposition used in Charm++ [4] and related programming models, including Adaptive MPI (AMPI). This separates the processes of decomposition from mapping of computation to processors and allows a more flexible mapping based on communication patterns between objects.

In this paper, we first present the abstraction of object-based decomposition in Charm++ and an API which provides a uniform interface for obtaining topology information at runtime on four different machines – Cray XT3, Cray XT4, IBM Blue Gene/L and IBM Blue Gene/P. This API can be used by user-level codes for task mapping and is independent of the programming model being used (MPI, Charm++ or something else). We then demonstrate topology aware mapping techniques for a communication intensive application: OpenAtom, a production Car-Parrinello *ab initio* Molecular Dynamics (CPAIMD) code. This application is used by scientists to study properties of materials and nano-scale molecular structures for biomimetic engineering, molecular electronics, surface catalysis, and many other areas [5, 6, 7].

We consider 3D torus topologies in this paper but not irregular networks or flat topologies. For logarithmic topologies (such as fat trees), the need to pay attention to topology may be smaller because the maximum number of hops between nodes tends to be small. Also, there is no support for user level derivation of topology for most fat-tree networks so any implementation would be specific to an individual cluster layout. This paper builds upon our previous work presented at Euro-Par 2009 in Amsterdam, The Netherlands. In this paper, we discuss the Topology Manager API and the time complexity of the mapping algorithms in more detail. We also delve into the issue of mapping multiple instances of OpenAtom on to the processor topology.

PREVIOUS WORK

There has been considerable research on the task mapping problem. The general mapping problem is computationally equivalent to the graph isomorphism problem which belongs to NP, neither known to be solvable in polynomial time nor NP-complete. Heuristic techniques like pairwise exchange were developed in the 80s by Bokhari [8] and Aggarwal [9]. These

schemes, however, are not scalable when mapping millions of objects to thousands of processors. This problem has been handled by others using recursive partitioning [10] and graph contraction [11]. Physical optimization techniques like simulated annealing [12] and genetic algorithms [13] are very effective but can take very long to arrive at optimal results. Results in the 80s and 90s were not demonstrated on real machines and even when they were, they were targeted towards small sized machines. They also did not consider real applications. With the emergence of large parallel machines, we need to revisit these techniques and build upon them, on a foundation of real machines, in the context of real applications.

The development of large parallel machines like Blue Gene/L, XT3, XT4 and Blue Gene/P has led to the re-emergence of mapping issues. Both application and system developers have evolved mapping techniques for Blue Gene/L [14, 15, 16, 17, 18, 19]. Yu [18] and Smith [17] discuss embedding techniques for graphs onto the 3D torus of Blue Gene/L which can be used by the MPI Topology functions. Weisser et al. [20] present an analysis of topology aware job placement techniques for XT3. However, our work is one of the first for task mapping on XT3. It presents a case study for OpenAtom, a production quantum chemistry code, and demonstrates high returns using topology aware schemes. Our earlier publication on OpenAtom [21] demonstrates the effectiveness of such schemes on Blue Gene/L. In this paper, we present results for XT3, Blue Gene/L and Blue Gene/P for multiple systems including a non-benchmark simulation.

On machines like Blue Gene/L and Blue Gene/P, obtaining topology information is simple and an interface is available to the programmers. The API described in this paper provides a wrapper for these and additional commonly needed functionality. However, on Cray XT machines, there is no interface for topology information, probably in accordance with the widespread, albeit mistaken idea, that topology mapping is not important on fast Cray machines. For XT machines, our API uses lower level system calls to obtain information about allocated partitions at runtime. To the best of our knowledge, there is no published work describing such functionality for the Cray machines. We believe that this information will be useful to programmers running on Cray machines. Also, the API provides a uniform interface which works on all these machines which hides architecture specific details from the application programmer. This API can be used as a library for Charm++, MPI or any other parallel program.

CHARM++ ARRAYS: A USEFUL ABSTRACTION FOR MAPPING

Parallelizing an application consists of two tasks: 1. decomposition of the problem into a large number of sub-problems to facilitate efficient parallelization to thousands of processors, 2. mapping of these sub-problems to physical processors to ensure load balance and minimum communication. Object-based decomposition separates the two tasks and gives independent control over both of them. In this paper, we use the Charm++ runtime which allows the application developer to decompose the problem into objects and the Charm++ runtime does a default mapping of objects to processors.

The basic unit of computation in Charm++ is called a *chare* (simply referred to as an "object" in this paper) which can be invoked through remote method invocations. The

application developer decomposes the problem into chares or objects and the CHARM++ runtime does a default mapping of objects to processors. Each processor can have multiple objects which facilitates overlap of computation and communication. This default mapping does not have any information about the machine topology. The user can override the default mapping with more intelligent schemes that take the topology of the machine into account.

Topology Manager API: Runtime Information

Mapping of communication graphs onto the processor graph requires information about the machine topology at runtime. The application should be able to query the runtime to get information like the dimensions of the allocated processor partition, mapping of ranks to physical nodes etc. However, the mapping interface should be simple and should hide machine-specific details from the application. The Topology Manager API in CHARM++ provides a uniform interface to the application developer and hence the application just knows that the job partition is a 3D torus or mesh topology. Application specific task mapping decisions require no architecture or machine specific knowledge (Blue Gene/L or XT3 for example).

The Topology Manager API in CHARM++ provides different functions which can be grouped into the following categories:

1. **Size and properties of the allocated partition:** At runtime, the application needs to know the dimensions of the allocated partition (`getDimNX, getDimNY, getDimNZ`), number of cores per node (`getDimNT`) and whether we have a torus or mesh in each dimension (`isTorusX, isTorusY, isTorusZ`).
2. **Properties of an individual node:** The interface also provides calls to convert from ranks to physical coordinates and vice-versa (`rankToCoordinates, coordinatesToRank`).
3. **Additional Functionality:** Mapping algorithms often need to calculate number of hops between two ranks or pick the closest rank to a given rank from a list. Hence, the API provides functions like `getHopsBetweenRanks, pickClosestRank` and `sortRanksByHops` to facilitate mapping algorithms.

We now discuss the process of extracting this information from the system at runtime and why is it useful to use the Topology Manager API on different machines:

IBM Blue Gene machines: On Blue Gene/L and Blue Gene/P [22], topology information is available through system calls to the "BGLPersonality" and "BGPPersonality" data structures, respectively. It is useful to use the Topology Manager API instead of the system calls for two reasons. First, these system calls can be expensive (especially on Blue Gene/L) and so it is advisable to avoid doing them a large number of times. The API does a few system calls to obtain enough information so that it can construct the topology information itself. It is useful to use the API instead of expensive system calls throughout the execution.

Cray XT machines: Cray machines have been designed with a significant overall bandwidth, and possibly for this reason, documentation for topology information was not readily available at the installations we used. We hope that the information provided here will be useful to

other application programmers. Obtaining topology information on XT machines is a two step process: 1. Getting the node ID (`nid`) corresponding to a given MPI rank (`pid`) which tells us which physical node a given MPI rank is on. 2. The second step is obtaining the physical coordinates for a given node ID. Once we have information about the physical coordinates for all ranks in the job, the API derives information such as the extent of the allocated partition by itself (this assumes that the machine has been reserved and we have a contiguous partition).

The API provides a uniform interface which works on all the above mentioned machines which hides architecture specific details from the application programmer. This API can be used as a library for CHARM++, MPI or any other parallel program. The next section describes the use of object-based decomposition and the Topology Manager API in a production code.

OPENATOM: A CASE STUDY

An accurate understanding of phenomena occurring at the quantum scale can be achieved by considering a model representing the electronic structure of the atoms involved. The CPAIMD method [23] is one such algorithm which has been widely used to study systems containing $10 - 10^3$ atoms. To achieve a fine-grained parallelization of CPAIMD, computation in OPENATOM [21] is divided into a large number of objects, enabling scaling to tens of thousands of processors. We will look at the parallel implementation of OPENATOM, explain the communication involved and then discuss the topology aware mapping of its objects.

In an *ab initio* approach, the system is driven by electrostatic interactions between the nuclei and electrons. Calculating the electrostatic energy involves computing several terms. Hence, CPAIMD computations involve a large number of phases with high inter-processor communication: (1) quantum mechanical kinetic energy of non-interacting electrons, (2) Coulomb interaction between electrons or the Hartree energy, (3) correction of the Hartree energy to account for the quantum nature of the electrons or the exchange-correlation energy, and (4) interaction of electrons with atoms in the system or the external energy. These phases are discretized into a large number of objects which generate a lot of communication, but ensures efficient interleaving of work. The entire computation is divided into ten phases which are parallelized by decomposing the physical system into fifteen *chare arrays*. For a detailed description of this algorithm please refer to [21].

Communication Dependencies

The ten phases referred to in the previous section are parallelized by decomposing the physical system into fifteen *chare arrays* of different dimensions (ranging between one and four). A simplified description of five of these arrays (those most relevant to the mapping) follows:

GSpace and RealSpace: These represent the g-space and real-space representations of each of the electronic states [23]. Each electronic state is represented by a 3D array of complex numbers. OPENATOM decomposes this data into a 2D *chare array* of objects. Each object holds a plane of one of the states (see Figure 1) . The *chare arrays* are represented by $G(s,p)$ $[n_s \times N_g]$ and $R(s,p)$ $[n_s \times N]$ respectively. GSpace and RealSpace interact through transpose operations

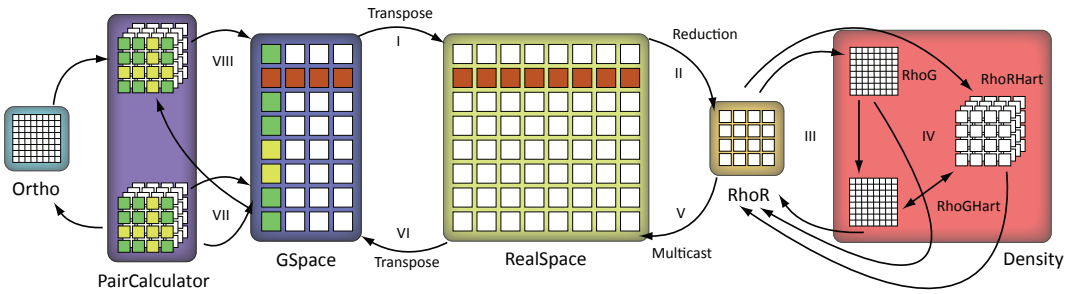


Figure 1. Decomposition of the physical system into chare arrays (only important ones shown for simplicity) in OpenAtom

(as part of a Fast Fourier Transform) in Phase I and hence all planes of one state of GSpace interact with all planes of the same state of RealSpace.

RhoG and RhoR: They are the g-space and real-space representations of electron density and are decomposed into 1D and 2D *chare arrays* respectively. They are represented as $G_\rho(p)$ and $R_\rho(p, p')$. RealSpace interacts with RhoR through reductions in Phase II. RhoG is obtained from RhoR in Phase III through two transposes.

PairCalculators: These 3D *chare arrays* are used in phase IV. They communicate with GSpace through multicasts and reductions. They are represented as $P_c(s, s', p)$ $[n_s \times n_s \times N_g]$. All elements of the GSpace array with a given state index interact with all elements of the PairCalculator array with the same state in one of their first two dimensions.

Mapping

OpenAtom provides us with a scenario where the load on each object is static (under the CPAIMD method) and the communication is regular and clearly understood. Hence, it should be possible to intelligently map the arrays in this application to minimize inter-processor communication and maintain load balance. OpenAtom has a default mapping scheme, but it should be noted that the default mapping is far from random. It is the mapping scheme used on standard fat-tree networks, wherein objects which communicate frequently are co-located on processors within the constraints of even distribution. This reduces the total communication volume. It only lacks a model for considering the relative distance between processors in its mapping considerations. We can do better than the default mapping by using the communication and topology information at runtime. We now describe how a complex interplay (of communication dependencies) between five of the *chare arrays* is handled by our mapping scheme.

GSpace and RealSpace are 2D *chare arrays* with states in one dimension and planes in the other. These arrays interact with each other through transpose operations where all planes of one state in GSpace, $G(s, *)$ talk to all planes of the same state, $R(s, *)$ in RealSpace (state-

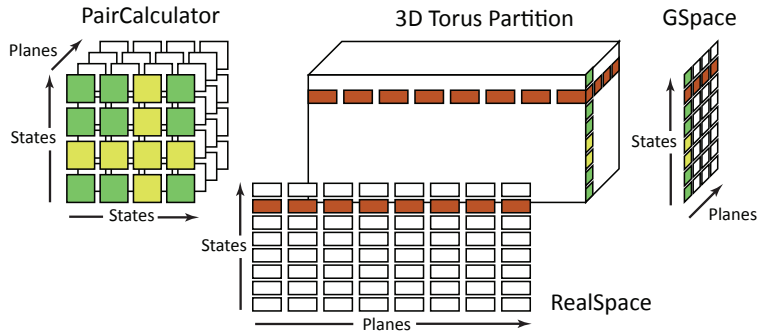


Figure 2. Mapping of different chare arrays to the 3D torus of the machine

wise communication). The number of planes in GSpace is different from that in RealSpace. GSpace also interacts with the PairCalculator arrays. Each plane of GSpace, $G(*, p)$ interacts with the corresponding plane, $P(*, *, p)$ of the PairCalculators (plane-wise communication) through multicasts and reductions. So, GSpace interacts state-wise with RealSpace and plane-wise with PairCalculators. If all planes of GSpace are placed together, then the transpose operation is favored, but if all states of GSpace are placed together, the multicasts/reductions are favored. To strike a balance between the two extremes, a hybrid map is built, where a subset of planes and states of these three arrays are placed on one processor.

Mapping GSpace and RealSpace Arrays: Initially, the GSpace array is placed on the torus and other objects are mapped relative to GSpace's mapping. The 3D torus is divided into rectangular boxes (which will be referred to as "prisms") such that the number of prisms is equal to the number of the planes in GSpace. The longest dimension of the prism is chosen to be same as one dimension of the torus. Each prism is used for all states of one plane of GSpace. Within each prism for a specific plane, the states in G(*, p) are laid out in increasing order along the long axis of the prism. Once GSpace is mapped, the RealSpace objects are placed. Prisms perpendicular to the GSpace prisms are created which are formed by including processors holding all planes for a particular state of GSpace, $G(s, *)$. These prisms are perpendicular to the GSpace prisms and the corresponding states of RealSpace, $R(s, *)$ are mapped on to these prisms. Figure 2 shows the GSpace objects (on the right) and the RealSpace objects (in the foreground) being mapped along the long dimension of the torus (box in the center).

Mapping of Density Arrays: RhoR objects communicate with RealSpace plane-wise and hence $R_\rho(p, *)$ have to be placed close to $R(*, p)$. To achieve this, we start with the centroid of the prism used by $R(*, p)$ and place RhoR objects in proximity to it. RhoG objects, $G_\rho(p)$ are mapped near RhoR objects, $R_\rho(p, *)$ but not on the same processors as RhoR to maximize overlap. The density computation is inherently smaller and hence occupies the center of the torus.

Table I. Time (in seconds) to obtain mapping solutions for RealSpace and RealParticlePlane objects on Blue Gene/P (System: WATER_256M_70Ry)

Cores	**1024**	**2048**	**4096**	**8192**
RealSpace	0.33	0.52	1.00	3.07
RealParticlePlane	1.48	3.19	4.90	17.89

Mapping PairCalculator Arrays: Since PairCalculator and GSpace objects interact plane-wise, the aim is to place $G(*,p)$ and $P(*,*,p)$ nearby. *Chares* with indices $P(s1,s2,p)$ are placed around the centroid of $G(s1,p),...,G(s1+block\_size,p)$ and $G(s2,p),....,G(s2+block\_size,p)$. This minimizes the hop-count for the multicast and reduction operations. The result of this mapping co-locates each plane of PairCalculators (on the left in Figure 2) with its corresponding plane of GSpace objects within the GSpace prisms.

The mapping schemes discussed above substantially reduce the hop-count for different phases. They also restrict different communication patterns to specific prisms within the torus, thereby reducing contention and ensuring balanced communication throughout the torus. State-wise and plane-wise communication is confined to different (orthogonal) prisms. This helps avoid scaling bottlenecks as we will see in the next section. These maps perform no better (and generally slightly worse) than the default maps on architectures which have more uniform network performance, such as Ethernet or Infiniband.

Time Complexity

Although maps are created only once during application start-up, they must still be efficient in terms of their space and time requirements. The memory cost of these maps grows linearly (4 integers per object) with the number of objects, which is a few megabytes in the largest system studied. The runtime cost of creating the most complex of these maps is $O(n^{3/2}log(n))$ where n is the number of objects. Despite this complexity, this time is sufficiently small that generating the maps for even the largest systems requires only a few minutes.

Table I shows the time it takes to construct two of the complex maps (*RealSpace* and *RealParticlePlane*) when running WATER_256_70Ry. Even on 8192 processors, it takes less than one-third of a minute to create a *RealParticlePlane* map. Algorithm 1 shows the pseudo code for creating the RealSpace map from the GSpace map. The creation of RealParticlePlane is similar but there are more sorting calls inside the first `for` loop which increases it running time. RealParticlePlane objects are not on the critical path of execution and hence the mapping of this array can be turned off it takes a long time for very large runs. This illustrates a common tradeoff: It is always important to evaluate if the time spent in computing a mapping is worth the performance benefit achieved from it.

RealSpace objects are extremely important because of the large communication with GSpace and density objects and their mapping cannot be ignored. The algorithm above shows that the

Algorithm 1: Mapping of RealSpace objects based on the the map for GSpace objects

```
begin
    Data: nstates (Number of states in the RealSpace chare array)
          nplanes (Number of planes in the RealSpace chare array)
          gsmap (Mapping of the GSpace chare array)
    Result: rsmap (Mapping of the RealSpace chare array)
    for state ← 1 to nstates do
        Create a processor list plist consisting of processors in gsmap[state, ∗]
        RSobjs_per_pe = maximum number of RSMap objects to be placed per processor
        for plane ← 1 to nplanes do
            Exclude processors which have RSobjs_per_pe from plist
            Sort plist by increasing hops from the first processor in the list
            Assign object rsmap_{state,plane} on the first element in plist
end
```

running time of the algorithm is $nstates \times nplanes \times nstates \times log(nstates)$. Approximating $nstates$ and $nplanes$ by $n^{1/2}$, the time complexity is $O(n^{3/2}log(n))$. There might be room for improvement if we can move the sorting out of the inner `for` loop. As an optimization, once created, maps can be stored and reloaded in subsequent runs to minimize restart time. Offline creation of maps using more sophisticated techniques and adapting these ideas to other topologies is an area of future work.

COMPARATIVE ANALYSIS OF OPENATOM

To analyze the effects of topology aware mapping in a production science code we studied the strong scaling (fixed problem size) performance of OpenAtom with and without topology aware mapping. Two benchmarks commonly used in the CPMD community: the minimization of WATER_32M_70Ry and WATER_256M_70Ry were used. The benchmarks simulate the electronic structure of 32 molecules and 256 molecules of water, respectively, with a standard g-space spherical cutoff radius of $|\mathbf{g}|^2_{cut} = 70$ Rydberg (Ry) on the states. To illustrate that the performance improvements extend beyond benchmarks to production science systems, we also present results for GST_BIG, which is a system being studied by our collaborator, Dr Glenn J. Martyna. GST_BIG consists of 64 molecules of Germanium, 128 molecules of Antimony and 256 molecules of Tellurium at cutoff radius of $|\mathbf{g}|^2_{cut} = 20$ Ry on the states.

Blue Gene/L (IBM T. J. Watson) runs are done in co-processor (CO) mode to use a single core per node. Blue Gene/P (Intrepid at ANL) runs were done in VN mode which uses all four cores per node. Cray XT3 (BigBen at PSC) runs are done in two modes: single core per node (SN) and two cores per node (VN). As shown in Table II, performance improvements from topology aware mapping for Blue Gene/L (BG/L) can be quite significant. As the number of cores and likewise, the diameter of the torus grows, the performance impact increases until it is a factor of two faster for WATER_32M_70Ry at 2048 and for WATER_256M_70Ry at 16384 cores. There is a maximum improvement of 40% for GST_BIG. The effect is not as strong in

Table II. Execution time per step (in seconds) of OpenAtom on Blue Gene/L (CO mode)

	WATER_32M_70Ry		WATER_256M_70Ry			GST_BIG
Cores	**Default**	**Topology**	**Default**	**Topology**	**Default**	**Topology**
512	0.274	0.259	-	-	-	-
1024	0.189	0.150	19.10	16.40	10.12	8.83
2048	0.219	0.112	13.88	8.14	7.14	6.18
4096	0.167	0.082	9.13	4.83	5.38	3.35
8192	0.129	0.063	4.83	2.75	3.13	1.89
16384	-	-	3.40	1.71	1.82	1.20

Table III. Execution time per step (in seconds) of OpenAtom on Blue Gene/P (VN mode)

	WATER_32M_70Ry		WATER_256M_70Ry			GST_BIG
Cores	**Default**	**Topology**	**Default**	**Topology**	**Default**	**Topology**
256	0.395	0.324	-	-	-	-
512	0.248	0.205	-	-	-	-
1024	0.188	0.127	10.78	6.70	6.24	5.16
2048	0.129	0.095	6.85	3.77	3.29	2.64
4096	0.114	0.067	4.21	2.17	3.63	2.53
8192	-	-	3.52	1.77	-	-

GST_BIG due to the fact that the time step in this system is dominated by a subset of the orthonormalization process which has not been optimized extensively, but a 40% improvement still represents a substantial improvement in time to solution.

Performance improvements on Blue Gene/P (BG/P) are similar to those observed on BG/L (Table III). The improvement for WATER_32M_70Ry is not as remarkable as on BG/L but for WATER_256M_70Ry, we see a factor of 2 improvement starting at 2048 cores. The absolute numbers on BG/P are much better than on BG/L partially because of the increase in processor speeds but more due to the better interconnect (higher bandwidth and an effective DMA engine). The performance for WATER_256M_70Ry at 1024 cores is 2.5 times better on BG/P than on BG/L. This is when comparing the VN mode on BG/P to the CO mode on BG/L. If we use only one core per node on BG/P, the performance difference is even greater, but the higher core per node count, combined with the DMA engine and faster network make single core per node use less interesting on BG/P.

The improvements from topology awareness on Cray XT3, presented in Table IV are comparable to those on BG/L and BG/P. The improvement of 27% and 21% on XT3 for WATER_256_70Ry and GST_BIG at $1,024$ cores is greater than the improvement of 14% and 13% respectively on BG/L at $1,024$ cores in spite of a much faster interconnect. However, on $2,048$ cores, performance improvements on the three machines are very close to one another.

Table IV. Execution time per step (in seconds) of OpenAtom on XT3 (SN and VN mode)

	WATER_32M_70Ry		WATER_256M_70Ry		GST_BIG	
Cores	**Default**	**Topology**	**Default**	**Topology**	**Default**	**Topology**
			Single core per node			
512	0.124	0.123	5.90	5.37	4.82	3.86
1024	0.095	0.078	4.08	3.24	2.49	2.02
			Two cores per node			
256	0.226	0.196	-	-	-	-
512	0.179	0.161	7.50	6.58	6.28	5.06
1024	0.144	0.114	5.70	4.14	3.51	2.76
2048	0.135	0.095	3.94	2.43	2.90	2.31

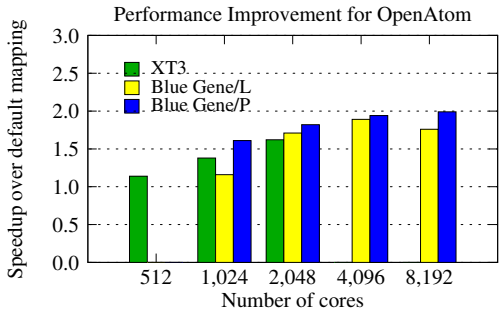


Figure 3. Comparison of benefit by topology aware mapping (System: WATER_256M_70Ry)

The improvement trends plotted in Figure 3 lead us to project that topology aware mapping should yield improvements proportional to torus size on larger Cray XT installations. The difference in processor speeds is approximately a factor of 4 (XT3 2.6 Ghz, BG/L 700 Mhz), which is reflected in the performance for the larger grained OpenAtom results on XT3 when comparing single core per node performance. The difference in network performance is approximately a factor of 7 (XT3 1.1 GB/s, BG/L 150 MB/s), when considering delivered bandwidth as measured by HPC Challenge [24] ping pong. This significant difference in absolute speed and computation/bandwidth ratios does not shield the XT3 from performance penalties from topology ignorant placement schemes. BG/P and BG/L show similar performance improvements which is expected since the BG/P architecture is similar to that of BG/L with slightly faster processors and increase network bandwidth.

OpenAtom is highly communication bound (as briefly discussed in the Introduction). Although Charm++ facilitates the exploitation of the available overlap and latency tolerance across phases, the amount of latency tolerance inevitably drops as the computation grain size

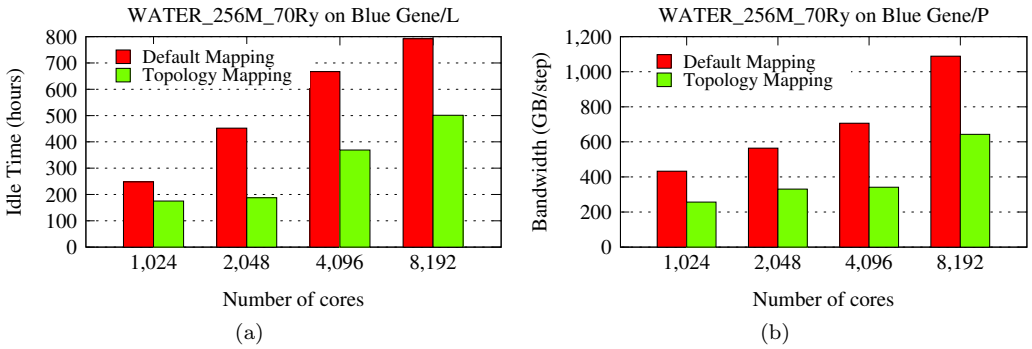


Figure 4. Effect of topology aware mapping (a) on idle time (time spent waiting for messages), and (b) on aggregate bandwidth consumption per step - smaller link bandwidth utilization suggests reduction in hops traversed by messages and hence reduction in contention

is decreased by the finer decomposition required for larger parallel runs. It is important to consider the reasons for these performance improvements in more detail. Figure 4(a) compares idle time as captured by the Projections profiling system in CHARM++ for OPENATOM on BG/L for the default mapping, versus the topology aware mapping. A processor is idle whenever it is waiting for messages to arrive. It is clear from Figure 4(a) that the factor of two speed increase from topology awareness is reflected directly in relative idle time and that the maximum speed increase which can be obtained from topology aware mapping is a reduction in the existing idle time.

It is illuminating to study the exact cause for this reduction in idle time. To that end, we ported IBM's High Performance Monitor library [25] for Blue Gene/P's Universal Performance Counters to Charm++, and enabled performance counters for a single time step in WATER_256M_70Ry in both topology aware and non-topology aware runs. We added the per node torus counters (BGP_TORUS_*_32BCHUNKS), to produce the aggregate link bandwidth consumed in one step across all nodes to obtain the results in Figure 4(b). This gives us an idea of the fraction of the total bandwidth across all links on the network used in one step. If messages travel fewer hops due a topology aware placement, it will lead to a smaller bandwidth consumption, thereby indicating less contention on the network. It is clear from the figure, that topology aware mapping results in a significant reduction, by up to a factor of two, in the total bandwidth consumed by the application. This more efficient use of the network is directly responsible for the reduction in latency due to contention and decreased idle time.

MULTIPLE APPLICATION INSTANCES

The discussion and results so far pertain to using OPENATOM for a single simulation of the evolution of the electronic states of a particular system. More information and/or improved accuracy can be obtained through the application of replica methods which combine the results

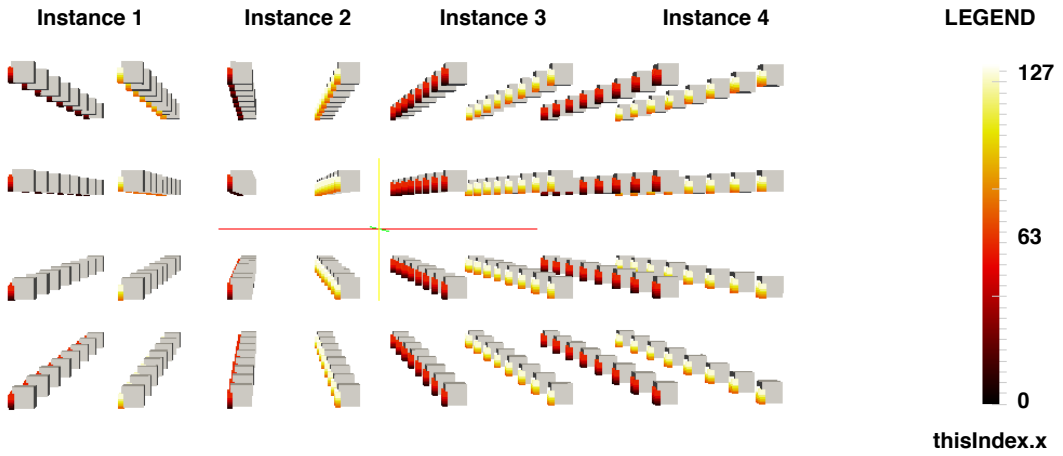


Figure 5. Mapping of four OpenAtom instances on a $8 \times 4 \times 8$ torus (System: WATER_32M_70Ry)

of multiple evolutions of a system. For example, applications of the path integral formulation combine multiple replicas to improve the accuracy of atomic positions. Similarly, when studying metals it is necessary to sample multiple K-points of the first Brillouin zone (the default scheme is $k = 0$) for convergence. Each of these methods and many similar applications not considered here, share the trait that most of the computation for each trajectory remains independent and their results are combined only within one phase of the computation. We can therefore treat these as largely independent instances of the simulation for most optimization considerations. In OpenAtom, we express each of these as separate instances of the Charm++ arrays.

Multiple instances of OpenAtom are mapped to different parts of the partition to prevent different instances from contending for the same resources. We divide the allocated partition along the longest dimension, into sub-partitions equal in number to the number of instances. We obtain the mappings for the first instance on the first sub-partition and then translate the maps along the longest dimension to obtain the maps for the other instances. Figure 5 shows the mapping of four OpenAtom instances on to a torus of dimensions $8 \times 4 \times 8$. The torus is split into four parts along the dimension (of size 8) orthogonal to the plane of the paper. Hence we see the color pattern repeated four times along this dimension for the identical mapping of the four instances.

There are other possible options which can be experimented with when mapping multiple instances. We can either split along the longest dimension or the shortest dimension. Let us take a concrete example where we have to map two instances of OpenAtom to be mapped on to a torus of dimensions $16x8x32$. If we split along the longest dimension, the diameter of the sub-partition is 28. If we split along the smallest dimension, the diameter of the sub-partition is still 28. Hence we do not hope to see major performance difference between the two schemes. Splitting along more than one dimension may degrade performance due to the creation of smaller meshes instead of tori.

CONCLUSION AND FUTURE WORK

In this paper we demonstrate that topology aware mapping can substantially improve performance for communication intensive applications on 3D mesh/torus networks. The performance improvement is achieved by minimizing bandwidth sharing between messages leading to reduced contention on the network. Significant improvements are obtained for the OpenAtom code and the effectiveness of topology aware mapping is shown for both IBM Blue Gene and Cray XT architectures. The Topology Manager API can be used to automatically obtain the topology for these machines at runtime. Mapping is also facilitated by object-based virtualization used in Charm++ which separates the processes of decomposition from mapping of computation to processors and allows a more flexible mapping based on communication patterns between objects.

Application developers can deploy the techniques presented in this paper to map their applications in a topology aware fashion. First and foremost, it is critical to ascertain that the application is latency sensitive and stands to benefit from topology aware mapping. This can be done by profiling the application using performance analysis tools and checking if computation waits unusually long for messages to be delivered. If this is the case, the Topology Manager API and mapping algorithms similar to those presented in the paper can be used to map an application. Overdecomposition, as in Charm++, can yield more efficient mappings because it provides additional degrees of freedom to place the communicating entities. The specific mapping algorithms to be used depend on the communication graph of the application.

In the future, we plan to improve the running time for computing maps for OpenAtom. A study of the effect of relative torus dimensions and routing protocols on communication performance will benefit the development of mapping algorithms. OpenAtom has complex but relatively regular or structured communication. We think that it is possible to develop general methodologies that deal with such structured communication. Unstructured static communication patterns, as represented by unstructured-mesh computations might need somewhat different mapping techniques. Work in the future will involve developing an automatic mapping framework which can work in tandem with the Topology Manager interface. This would also require a study of a more diverse set of applications with different communication patterns. Further study will be given to characterizing network resource usage patterns with respect to those which are most affected by topology aware task mapping.

ACKNOWLEDGEMENTS

This work was supported in part by DOE Grants B341494 funded by Center for Simulation of Advanced Rockets and DE-FG05-08OR23332 through ORNL LCF. This research was supported in part by NSF through TeraGrid [26] resources provided by NCSA and PSC through grants ASC050040N and MCA93S028. We also thank Fred Mintzer and Glenn Martyna from IBM for access and assistance in running on the Watson Blue Gene/L. We also used running time on the Blue Gene/P at Argonne National Laboratory, which is supported by DOE under contract DE-AC02-06CH11357.

REFERENCES

1. Greenberg RI, Oh HC. Universal wormhole routing. *IEEE Transactions on Parallel and Distributed Systems* 1997; **08**(3):254–262.
2. Ni LM, McKinley PK. A survey of wormhole routing techniques in direct networks. *Computer* 1993; **26**(2):62–76.
3. Bhatelé A, Kalé LV. Quantifying Network Contention on Large Parallel Machines. *Parallel Processing Letters (Special Issue on Large-Scale Parallel Processing)* 2009; **19**(4):553–572.
4. Kalé L, Krishnan S. CHARM++: A Portable Concurrent Object Oriented System Based on C++. *Proceedings of OOPSLA'93*, Paepcke A (ed.), ACM Press, 1993; 91–108.
5. A P, MS H, R C. Interface structure between silicon and its oxide by first-principles molecular dynamics. *Nature* 1998; **396**:58.
6. L DS, P C. Serine proteases: An ab initio molecular dynamics study. *Proteins* 1999; **37**:611.
7. Saitta AM, Soper PD, Wasserman E, Klein ML. Influence of a knot on the strength of a polymer strand. *Nature* 1999; **399**:46.
8. Bokhari SH. On the mapping problem. *IEEE Trans. Computers* 1981; **30**(3):207–214.
9. Lee SY, Aggarwal JK. A mapping strategy for parallel processing. *IEEE Trans. Computers* 1987; **36**(4):433–442.
10. Ercal F, Ramanujam J, Sadayappan P. Task allocation onto a hypercube by recursive mincut bipartitioning. *Proceedings of the 3rd conference on Hypercube concurrent computers and applications*, ACM Press, 1988; 210–221.
11. Berman F, Snyder L. On mapping parallel algorithms into parallel architectures. *Journal of Parallel and Distributed Computing* 1987; **4**(5):439–458.
12. Bollinger SW, Midkiff SF. Processor and link assignment in multicomputers using simulated annealing. *ICPP (1)*, 1988; 1–7.
13. Arunkumar S, Chockalingam T. Randomized heuristics for the mapping problem. *International Journal of High Speed Computing (IJHSC)* December 1992; **4**(4):289–300.
14. Bhanot G, Gara A, Heidelberger P, Lawless E, Sexton JC, Walkup R. Optimizing task layout on the Blue Gene/L supercomputer. *IBM Journal of Research and Development* 2005; **49**(2/3):489–500.
15. Gygi F, Draeger EW, Schulz M, Supinski BRD, Gunnels JA, Austel V, Sexton JC, Franchetti F, Kral S, Ueberhuber C, *et al.*. Large-Scale Electronic Structure Calculations of High-Z Metals on the Blue Gene/L Platform. *Proceedings of the International Conference in Supercomputing*, ACM Press, 2006.
16. Bhatelé A, Kalé LV, Kumar S. Dynamic Topology Aware Load Balancing Algorithms for Molecular Dynamics Applications. *23rd ACM International Conference on Supercomputing*, 2009.
17. Smith BE, Bode B. Performance Effects of Node Mappings on the IBM Blue Gene/L Machine. *Euro-Par*, 2005; 1005–1013.
18. Yu H, Chung IH, Moreira J. Topology mapping for Blue Gene/L supercomputer. *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, ACM: New York, NY, USA, 2006; 116, doi: http://doi.acm.org/10.1145/1188455.1188576.
19. Bhatelé A, Kalé LV. Benefits of Topology Aware Mapping for Mesh Interconnects. *Parallel Processing Letters (Special issue on Large-Scale Parallel Processing)* 2008; **18**(4):549–566.
20. Deborah Weisser, Nick Nystrom, Chad Vizino, Shawn T Brown, and John Urbanic. Optimizing Job Placement on the Cray XT3. *48th Cray User Group Proceedings* 2006; .
21. Bohm E, Bhatele A, Kale LV, Tuckerman ME, Kumar S, Gunnels JA, Martyna GJ. Fine Grained Parallelization of the Car-Parrinello ab initio MD Method on Blue Gene/L. *IBM Journal of Research and Development: Applications of Massively Parallel Systems* 2008; **52**(1/2):159–174.
22. IBM Blue Gene Team. Overview of the IBM Blue Gene/P project. *IBM Journal of Research and Development* 2008; **52**(1/2).
23. Tuckerman ME. Ab initio molecular dynamics: Basic concepts, current trends and novel applications. *J. Phys. Condensed Matter* 2002; **14**:R1297.
24. Dongarra J, Luszczek P. Introduction to the HPC Challenge Benchmark Suite. *Technical Report UT-CS-05-544*, University of Tennessee, Dept. of Computer Science 2005.
25. Salapura V, Ganesan K, Gara A, Gschwind M, Sexton J, Walkup R. Next-Generation Performance Counters: Towards Monitoring Over Thousand Concurrent Events. *IEEE International Symposium on Performance Analysis of Systems and Software*, 2008; 139 – 146.
26. Catlett C, et al. TeraGrid: Analysis of Organization, System Architecture, and Middleware Enabling New Types of Applications. *HPC and Grids in Action*, Grandinetti L (ed.), IOS Press: Amsterdam, 2007.