Runtime Optimization of Vector Operations on Large Scale SMP Clusters

Costin Iancu and Steven Hofmeyr
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
cciancu@lbl.gov, shofmeyr@lbl.gov

ABSTRACT

"Vector" style communication operations transfer multiple disjoint memory regions within one logical step. These operations are widely used in applications, they do improve application performance, and their behavior has been studied and optimized using different implementation techniques across a large variety of systems. In this paper we present a methodology for the selection of the best performing implementation of a *vector* operation from multiple alternative implementations. Our approach is designed to work for systems with wide SMP nodes where we believe that most published studies fail to correctly predict performance. Due to the emergence of multi-core processors we believe that techniques similar to ours will be incorporated for performance reasons in communication libraries or language runtimes.

The methodology relies on the exploration of the application space and a classification of the regions within this space where a particular implementation method performs best. We use micro-benchmarks to measure the performance of an implementation for a given point in the application space and then compose profiles that compare the performance of two given implementations. These profiles capture an empirical upper bound for the performance degradation of a given protocol under heavy node load. At runtime, the application selects the implementation according to these performance profiles. Our approach provides performance portability and using our dynamic multi-protocol selection we have been able to improve the performance of a NAS Parallel Benchmarks workload by 22% on an IBM large scale cluster. Very positive results have also been obtained on large scale InfiniBand and Cray XT systems. This work indicates that perhaps the most important factor for application performance on wide SMP systems is the successful management of load on the Network Interface Cards.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: [Design studies, Modeling techniques]; D.2.4 [**Software Engineering**]: Metrics—*Performance measures*; D.1.3 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; D.3 [**Programming Languages**]: [Parallel, Compilers]; I.6.4 [**Computing Methodologies**]: Simulation and Modeling—*Model Validation and Analysis*

General Terms

Performance, Measurement, Languages, Design

Keywords

Parallel Programming, Program Transformations, Performance Portability, Communication Code Generation, Latency Hiding

PACT'08, October 25–29, 2008, Toronto, Ontario, Canada.
ACM 978-1-60558-282-5/08/10.

1. INTRODUCTION

Contemporary high performance networks provide hardware support for Remote Direct Memory Access (RDMA) and efficient non-blocking communication. Exploitation of these features [4, 14, 15, 25] either at the application level or inside communication libraries has shown considerable performance benefits.

Multiple studies show how to improve the performance of applications using *Put/Get* primitives by decomposing transfers and hiding communication latency through overlap. Several classes of applications require the transfer of multiple disjoint memory regions in a single logical operation e.g., boundary data in finite difference calculations, particle-mesh structures or sparse matrices. To address the needs of these applications, communication layers provide higher level primitives for such transfers. These operations are usually referred to in literature as *vector* or *strided* and they have been shown to improve application performance [19, 23, 7]. Most of the native communication layers, such as Elan, InfiniBand Verbs, IBM LAPI provide an API for these operations. There are also third party one-sided communication libraries, such as ARMCI [17] or GASNet [5] that provide an API and efficient implementations.

The third party communication libraries are very portable, run on a large variety of modern networks and have been used as targets for code generation for parallel programming languages: Titanium [31], Co-Array Fortran [20], Unified Parallel C [28] target GASNet, Global Arrays [18] and CAF target ARMCI. They usually provide alternate implementations for the *vector* operations and match or improve the performance of the native libraries. In general, *vector* style communication primitives are important enough that they are considered for language level extensions [6] and have

been used for code generation by compilers for parallel programming languages [8, 31, 28].

Regardless of their provenance, implementations for transfers of multiple disjoint memory regions usually use a single approach: the simple ones pipeline individual messages while the more sophisticated use data packing with offloading or Active Messages [30]. However, the performance is highly dependent on system architecture, load and application characteristics and it is the case that a "single protocol" implementation [19, 23] will not offer best performance in all situations.

In this paper we present a methodology for the selection of the best performing implementation of a *vector* operation from multiple alternative implementations. Our approach is designed to work for systems with wide SMP nodes and to our knowledge this is the first published study that presents a successful methodology for dynamically tuning *vector* operations on such systems. Due to the emergence of multi-core processors we believe that techniques similar to ours will be incorporated for good performance in future communication libraries or language runtimes.

Our approach relies on the exploration of the application space and a classification of the regions within this space where a particular implementation method performs best. We use micro-benchmarks to measure the performance of an implementation for a given point in the application space and then compose profiles that compare the performance of two given implementations. These profiles capture an empirical upper bound for the performance degradation of a given protocol under heavy node load. At runtime, the application selects the implementation according to these performance profiles. We have validated this approach on a large variety of contemporary networks: InfiniBand, Quadrics, IBM LAPI and Cray XT3; using the Berkeley UPC compiler and the GASNet communication layer.

At first glance, the feasibility of our approach seems to be doomed by the sheer volume of the application space that needs to be explored: data volume, number of messages, system size and architecture, communication topology, load balance are all factors that might affect performance. We perform a large set of experiments to prune the space of the performance parameters. Our results indicate that the determining factor is the load on the Network Interface Card within an SMP node. Using our techniques we eliminate the performance "noise" caused by congestion and we have obtained good performance results on several NAS Parallel Benchmarks application kernels. We have observed performance improvements as high as 22% across the whole application workload and a maximum of $\approx 250\%$ improvements in some instance.

2. MOTIVATION

This research has been motivated by our work in developing a high performance portable compiler for the Unified Parallel C language. UPC is a Partitioned Global Address Space programming language that assumes a one-sided communication model. The current UPC implementation uses a source-to-source translation approach and generates code that runs on top of the GASNet communication layer. GASNet is a portable high performance one-sided communication library with efficient implementations on a large variety of contemporary networks: Quadrics, InfiniBand, Myrinet, IBM LAPI and the Cray XT family.

One of the main goals of our research has been to provide performance portability. GASNet provides a wide communication interface and for some applications there are multiple choices for the code generation strategy. Ideally, the UPC compiler or runtime should be able to choose the best performing implementation strategy for an application on a given system. We believe that performance model based code generation strategies are capable of greatly improving development productivity and application performance on large scale parallel systems. From the beginning of this work, we strove to provide a simple and practical approach and we tried to prune the performance parameter space in order to achieve both a lightweight[1] system tuning step and provide a methodology that is intuitive to application and library developers. We do believe that the more intricate the knowledge about application characteristics required for performance tuning, the less the potential for adoption by developers.

We have already shown [12] good results when using performance models to guide strip-mining and overlap transformations for programs that use *Put/Get* primitives. These models are able to choose a good optimization strategy based on application characteristics and network load and improve application scalability to a degree hard to match when using manual transformations.

In this paper we present a performance tuning methodology for applications that transfer multiple data regions in one logical step. Multiple code generation strategies are available at the application level: blocking communication, pipelining of non-blocking communication calls or direct calls to the GASNet *vector* interface. Multiple implementations might be also available for the *vector* interface at the communication library level. Previous studies [19, 23] clearly indicate the value of using a multi-protocol approach for the code generation in applications. We are not aware of any published approach that can accurately select the best performing implementation in all cases, especially in the small to mid-size transfer ranges that are widely used [29] in scientific applications. One of the limitations of current approaches is ignoring the behavior at the SMP node level, when multiple processors within a node are active. A recent performance study [27] indicates that dual-core processors issuing word size transfers can use up to 80% of modern Network Interface Cards capacity, while quad-core processors can fully saturate the network. With the advent of multi-core processors we believe that this problem will have to be explicitly addressed at the communication library level or at the runtime level.

In the rest of this paper we consider as "wide" SMP nodes any system where the network is underprovisioned with respect to the processor hierarchy. Section 3 discusses the state of the art. Section 4 describes the existing interfaces and the impact of implementations on performance. In Section 5 we describe the methodology and finally we present application results and conclusions.

3. RELATED WORK

The performance of *vector* communication primitives on RDMA capable networks has been studied at the implementation as well as at the application level.

Tipparaju et al [26] present the implementation of host-

[1]Small number of short running experiments.

assisted zero-copy remote memory access communication operations on InfiniBand networks and "thin" SMP nodes. They present various implementation alternatives for transfers of non-contiguous memory regions and use microbench–marks to show the performance potential of RDMA operations. One of the conclusions of their study is the importance of using multiple protocols in achieving good sustained performance. Nieplocha et al [19] study the performance of strided RDMA operations on the Quadrics QsNetII network. They describe host-assisted implementations and implementations that offload data processing to the Network Interface Card and discuss the performance trade-offs of each approach. They do not provide a methodology for choosing the best implementation and in particular, their application performance results indicate that while offloading always performs best when running with one processor per SMP node, there is no clear winner when utilizing the full node.

Vector operations have been also studied from the point of view of their usage in higher level programming abstractions. Santhanaraman et al [22] discuss implementation alternatives for MPI derived data types over InfiniBand. They evaluate multiple implementations and show good performance improvements when using the existing hardware *scatter/gather* support. Their data indicates that the SGRS proposed scheme outperforms the other evaluated schemes when the length of contiguous data regions is relatively large or there is a large number of regions transferred. It is not clear what implementation alternative performs best in the non-asymptotic case. Coarfa et al [10] discuss using *vector* operations in the code generation for CAF programs. They show promising performance results for the NAS Parallel Benchmarks but also comment on problems with the noisy behavior of these primitives. Su et al [23] discuss code generation strategies for applications that use irregular communication in the context of the Titanium programming language. They propose a performance model based approach for multi-protocol inspector/executor implementations. Their results show the performance advantages of the multi-protocol approach at large system scale, but the approach has not been validated when running with multiple processors per SMP node. Cameron et al [9] present a performance model that takes into account the influence of middle-ware on application performance. One of their case studies is the implementation of strided memory transfers and their approach is very close in spirit to the approach in [23]. For these operations their results have been validated on a thin node (2 processors) SMP cluster in a point-to-point experiment (2 nodes).

Another relevant research direction is employing machine learning techniques for automatic algorithm selection as illustrated in STAPL [24]. We believe that similar techniques are perfectly capable of solving the problem presented here. However, in this case machine learning will require a large training overhead, a larger number of parameters and will provide no performance intuition to application developers.

4. VECTOR OPERATIONS

There are several supported interfaces for vector operations. The most common form takes as arguments a list of source and destination addresses and a list of region lengths. Most native and third party communication libraries support this interface: Quadrics (e.g. `elan_putv`), IBM LAPI and GASNet support this form. InfiniBand supports a restricted form of this interface, Send Gather/Recv Scatter, where the buffer at one endpoint has to be contiguous. In order to reduce meta-data overhead, libraries such as IBM LAPI and GASNet provide a strided version of the interface which takes as arguments a source address, a destination address, a length, a stride and a message count. GASNet supports a generic N-stride to N-stride interface. IBM LAPI also supports generic Data Gather Scatter Programs (DGSP) and provide a meta-compiler for them. The proposed [6] UPC level interface includes both *vector* and strided operations.

At the implementation level, the simplest optimization approach is to use non-blocking communication and pipeline individual messages. More sophisticated implementations perform packing/unpacking in order to reduce the overhead of message injection and better utilize bandwidth. Library providers employ different approaches: 1) a helper thread is used to perform the packing in ARMCI and IBM LAPI; 2) the packing is offloaded to the NIC in the ARMCI Quadrics implementation; 3) the packing is implemented using Active Messages and polling in GASNet.

The first obvious performance trade-off is whether the bandwidth and message injection time improvements are enough to offset the cost of packing when compared to the pipelined implementations. All of the studies cited in this paper take into account only this aspect.

Some more subtle problems are posed by the packing implementations, namely fairness and scalability. Offloading diverts NIC resources to packing and has the potential to make it unresponsive to other communication requests. In this case the NIC is oversubscribed. Helper threads divert CPU resources and have to potential to increase latency due to context switching and CPU scheduling and also raise the issue of fairness within an SMP node. In this case the SMP node is oversubscribed. The GASNet implementation uses Active Messages and polling and besides fairness problems it might suffer from attentiveness problems. Since AMs are served only when threads enter the communication library, the latency of AM based operations is potentially unbound. Fairness issues are raised whenever multiple AM based operations are outstanding at a SMP node. Regardless of the implementation strategy, interrupt based or polling, it might happen that certain threads will serve a large fraction of the asynchronous events.

Dwelling deeper upon library internals, packing implementations usually pipeline the packing / transmission / unpacking process and they have an internal parameter for the transmission unit. In the GASNet implementation this is referred to as AMSize. Implementations pack until filling the threshold size for a buffer and then transmit. When the contiguous piece of a vector request is over the threshold, implementations usually use non-blocking communication primitives directly.

Another orthogonal aspect that determines performance is the execution model used by application runtimes. The choice is running an application either with full fledged UNIX processes or with threads (pthreads). Threads usually share a "software" connection while processes have their own and therefore might observe different QoS (fairness) behavior.

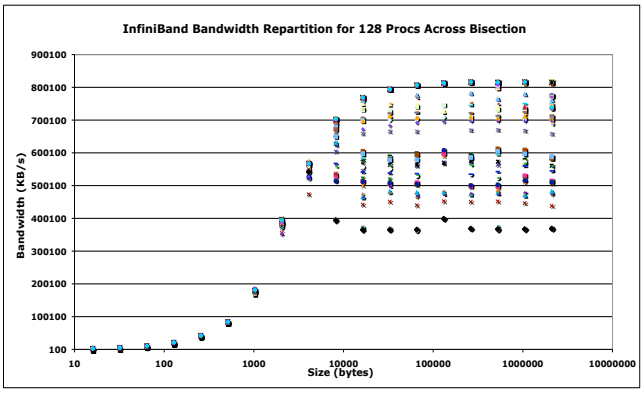


Figure 1: Variation of per-pair sustained bandwidth for 128 processors communication across network bisection on an InfiniBand network.

5. PERFORMANCE CHARACTERIZATION

The performance of a given implementation is determined by a combination of application characteristics and systems characteristics. For our case, the important application metrics are: number and length of messages, communication topology, degree of parallelism and load balance. The important system parameters are [1]: o - overhead of initiating communication, G - bandwidth and overall response to congestion.

Our previous work [12] indicates that on the system side, *unless* congestion is present, the variation of performance parameters is "continuous", e.g. the variation of overhead and bandwidth with message size. Measurements of these parameters show an evolution without large discontinuities and they can be reasonably well approximated with analy–tical models. Our first assumption is that when comparing two implementations, varying parameter values will reveal large continuous ranges on the variation axis where relative performance does not change.

Problems appear when a parameter setting causes congestion, i.e. some internal resource is over-committed. For example pipelining a large number of messages will exhaust card resources (message queue) and flow control mechanisms are activated. Examining the fairness [12] of bandwidth allocation between pairs of communicating processors shows a large variance in the per-pair sustained bandwidth. Settings with a small number of active processors show little variance, continuously increasing the system size shows an asymptotic difference of 100% as illustrated in Figure 1. The logical communication topology affects this variance: *nearest neighbor* communication exhibits very little variance while *cross* network communication exhibits a large variance. In all these cases, performance response becomes noisy and "discrete" and we are not aware of any performance modeling work that can characterize accurately congestion response on these networks.

The second underlying assumption of our approach is that when one setting for a particular parameter causes congestion, "increasing" that value will still cause congestion. We assume that when comparing two implementations, the scheme that avoids congestion will perform faster. Because increasing the value of a parameter can only increase congestion, we again expect to see large contiguous ranges for parameters where relative performance is stable, i.e. one particular implementation is always faster.

```
foreach(S)
  start_time()
  for (iters)
    foreach(N)
      get(S)
  end_time()

foreach(S)
  start_time()
  for(iters)
    foreach(N)
      get_nb(S)
    sync_all
  end_time()

foreach(S)
  start_time()
  for(iters)
    foreach(N)
      vector_get(N,S)
  end_time()
```

Figure 2: Micro-benchmark code, N is number of messages, S is message size.

Our characterization approach is based on these observations. We consider three candidate implementations of *vector* operations: 1) blocking *Put/Get* communication; 2) pipelined *Put/Get*; and 3) Active Messages based packing. We consider the problem parameter space and design micro-benchmarks to determine how the parameter variation affects relative performance. Our micro-benchmarks are designed for the pessimistic case: they measure an upper bound for performance degradation under heavy system load. Contemporary wide node SMP systems are likely to operate with heavy loads on the Network Interface Cards. Underwood et al [27] indicate that dual-core processors issuing word size transfers can use up to 80% of modern Network Interface Cards capacity, while quad-core processors can fully saturate the network. Some of the recently deployed large scale systems evaluated here contain eight or sixteen cores per SMP node.

Based on the micro-benchmark results we produce an ordering of the performance of various implementations in a pruned parameter space. This ordering is used at application runtime to choose the best performing implementation given the parameter setting.

5.1 Classification

We use for the classification a simple micro-benchmark that transfers N messages of size S using three different implementations: blocking, pipelining and AM based. We vary the values of N and S, the number of tasks, tasks per node and communication topology. The number of tasks per node captures how NICs and SMP nodes respond to congestion. The total number of tasks captures the network response at scale. The communication topology captures how the network responds to congested paths. We also consider implementations running with either pthreads or processes. The micro-benchmark code is presented in Figure 2.

In the experiments, the value of the numerical parameters (N, S, total tasks) is varied in powers of two. The number of tasks per node is varied from one to the number of cores per node. We consider two communication topologies: *nearest neighbor* ($P_i <-> P_{i+1}$) and *cross network* ($P_i <-> P_{(i+\frac{P}{2})\%P}$). Our choice of logical topologies is motivated by the fact that most likely neighboring logical indexes will map to physical neighbors on the networks considered. For each parameter setting we determine the **maximum** of the running time across all active entities. We think this approach best captures the effect of congestion on appli-

System	Network	CPUs X Nodes
AMD cluster [13]	InfiniBand 4x	2 x 320 2.2Ghz Opteron
IBM p575 [2]	Federation	8 x 111 1.9Ghz POWER5
Cray XT3[3]	Custom	2 x 2068 2.6Ghz Opteron
Sun AMD cluster[21]	InfiniBand	16 x 3936 1.9Ghz Barcelona

Table 1: Systems Used for Benchmarks

cation performance and we classify implementations based on this value.

The structure of the micro-benchmark is simple but deceiving. We have extracted predictors using permutations of the loops inside the micro-benchmark, predictors where there are barriers present at the end of the `foreach(S)` loop iteration or predictors based on the fastest execution (**minimum** execution time) for each parameter setting. Our application results show that these predictors did not perform as well in practice.

6. EXPERIMENTAL RESULTS

We ran the micro-benchmarks on the systems described in Table 1. The InfiniBand and IBM Federation systems are connected in a fat-tree topology. The Sun Constellation system is connected by a full-CLOS network. The Cray XT system is connected using a 3-D torus topology. Two systems have thin SMP nodes (two processors per node) and two systems have wide nodes (eight and sixteen processors per node).

All of our experiments have been performed using UNIX processes. In this case the current GASNet and UPC implementation uses the Network Interface Card for intra-node communication and we make the distinction between intra-node and inter-node traffic. In a thread based implementation our models consider only inter-node traffic.

6.1 Micro-Benchmark Results

Figures 3 and 4 present selected micro-benchmark results. Figures 3(a)(b) and 4(c)(d) present the relative performance of two implementations for a *vector* operation that transfers NMSG contiguous regions of length SIZE. The three implementations are: 1) AM based packing is labeled VIS; 2) PIPE denotes an implementation that pipelines the original transfers; and 3) BLOCK refers to the implementation using blocking communication. For example a chart containing VIS/PIPE in the title bar plots $\frac{T_{VIS}}{T_{PIPE}}$ for the given transfer. A value less than one will indicate that VIS is faster than PIPE. For presentation reasons, the magnitude of the charts has been bounded at five in all cases.

Most of the performance modeling efforts that we are aware of, measure network performance parameters using an experimental setting with one processor per SMP node, usually with uni-directional traffic. All of our results were obtained with bi-directional traffic.

6.1.1 Node Response

These experiments are designed to evaluate the impact of the steady state load on the Networking Interface Card on the performance of *vector* operations. The first set of experiments we conduct measures the relative performance of implementations in a two node setting where only one processor per node is active. Figures 3 (a)(b) present the relative performance results for the Sun Constellation system and Figure 3(c) shows the best performing protocol. The results are qualitatively similar on all other systems considered: the blocking implementation under-performs in all cases on all systems and there is an observable difference in the behavior of inter-node and intra-node traffic.

The second set of experiments we conducted measured the relative performance of implementations where two nodes are active and all the processors within a node are active. On the systems with thin SMP nodes, the results are very similar to the previous case and the blocking implementation still under-performs in all cases. On these systems it is the case that processors can not saturate the NIC.

Figure 3(d) shows the best performing protocol for the inter-node experiment on the Sun system. In this case the processors can saturate the Network Interface Card and the pipelined implementation performs worse than the blocking and the packing implementations. Similar results are observed on the IBM system. On this system, the presence of a helper thread increases the system noise and the magnitude of the performance differences.

A comparison of Figures 3(c) and (d) shows that obtaining good performance when increasing the number of active processors within a SMP node requires a protocol change. Since both VIS and BLOCK implementations attain a lighter node load (fewer messages and longer inter-arrival time respectively), this difference illustrates that steady state congestion on the Network Interface Card adversely affects performance.

6.1.2 Network Scaling Response

To assess the impact of network scaling on the implementation of *vector* operations, we consider both the impact of communication topology and the total number of active nodes in the system. We consider the *nearest neighbor* and *cross network* communication topologies presented in 5.1 which we examine at increasing concurrency.

Scaling the benchmark with the number of processors affects directly the sustained load on the Network Interface Card. The fewer the number of participating nodes and the closer they are situated in the physical network, the higher the attainable load on the cards. A comparison of Figures 4(a) and (b) illustrates this. In chart (b) which presents the relative VIS/PIPE performance on the Cray XT system for a 32 node *cross network* experiment, the region where VIS performs best is larger than the corresponding region in chart (a) which shows the two node experiment at heavy communication load. Profiles similar to chart (b) are observable in the two node case only when running a modified version of the benchmark that generates a lighter load on the cards.

For lack of space we do not present a more complete set of results. On all systems examined, a *nearest neighbor* experiment is able to generate a higher load on the cards than a *cross network* experiment. This manifests as a smaller area where the VIS implementation performs better for *nearest neighbor*. The results for *nearest neighbor* experiments are very similar to the results observed for the two node experiments: hence we can discard this topology as a determining performance parameter.

The question that remains to be answered is whether we can completely discard communication topology as a determining performance parameter for *vector* operations. We have chosen the node[2] predictor as a "baseline" predictor

[2]Based on communication between two nodes at full utiliza-

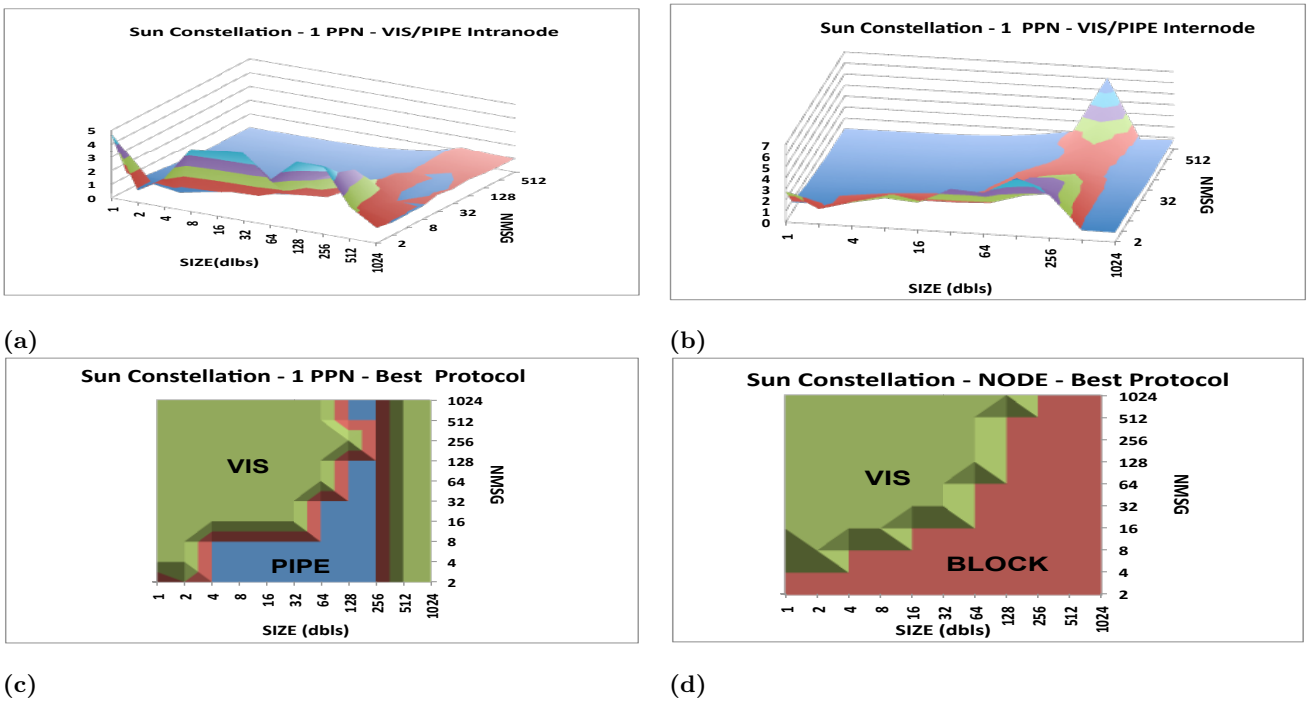


Figure 3: Relative performance of implementations reported as the ratio of the duration to perform a certain transfer. SIZE = length of contiguous region in doubles. NMSG = number of regions.VIS = AM based packing. PIPE = message pipelining. BLOCK = blocking communication, PPN = processors per node, NODE = all processors on a node. For example a VIS/PIPE chart plots $\frac{T_{VIS}}{T_{PIPE}}$. Values less than 1 indicate that VIS is faster than PIPE for that given transfer. The magnitude of the performance differences has been limited to 5 for presentation reasons.

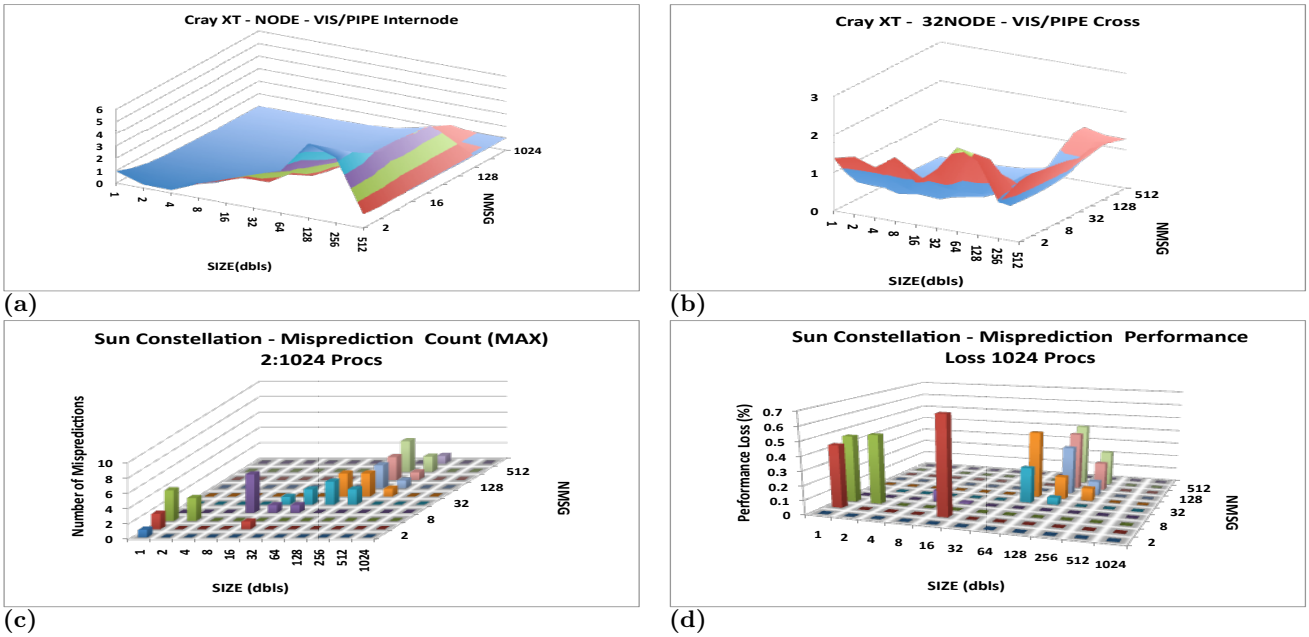
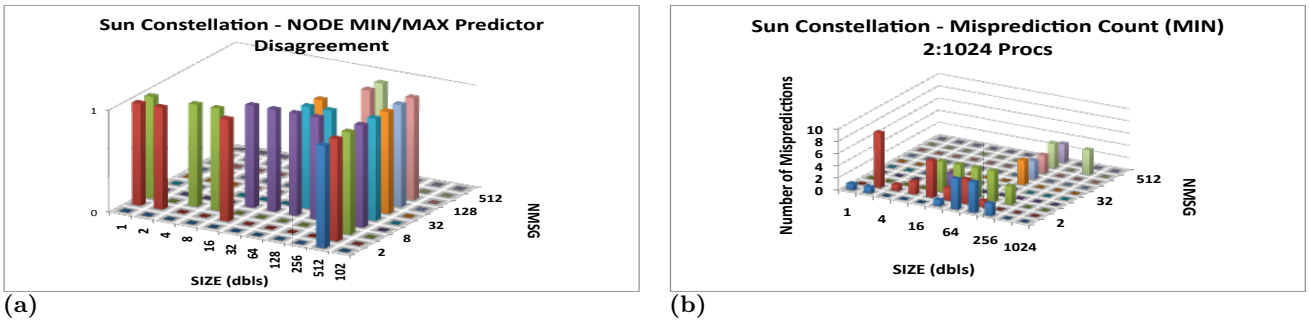


Figure 4: Impact of concurrency on observed performance and accuracy of predictions. Figure (c) presents the distribution of mispredictions when using the node predictor as a baseline and increasing the concurrency from 2 to 1024 processors. Height of bars shows the number of concurrency levels where mispredictions occur.



(a) (b)

Figure 5: Behavior of "optimistic" predictors (minimum based). (a) Disagreement in prediction between optimistic and pessimistic predictors. (b) Lack of accuracy at scale of optimistic predictors.

and extracted from *cross* predictors the points in the (SIZE, NMSG, Procs) space where the protocol choices are different. The concurrency is varied depending on the system from two processors to 512 or 1024 processors.

For each point in this intersection space we examine the performance penalty of using the wrong protocol. The results indicate that using a predictor that underestimates load, such as a *cross* predictor, will result in most cases, when mis-predicting, in a choice with a greater performance penalty impact than using a predictor obtained under heavier load, such as the node predictor. For example, in Figure 4(a) and (b) for the point (256, 16), the (b) *cross* predictor will choose the VIS implementation as the best candidate. Based on the node predictor (a), this implementation is four times slower than the PIPE implementation. Doing the reverse exercise will result in choice that exhibits only a 30% slowdown. Similar trends are observed when comparing predictors at increasing concurrency. While we have not explored the whole (topology, concurrency) parameter space we believe that based on the trends observed for samples in this space we can safely eliminate topology as a determining performance factor.

The other parameter of interest is the concurrency level. Optimized *vector* operations perform well for the cases where the length of the contiguous pieces of memory transferred is small to medium (KB). For these cases, it is unlikely[3] that the total volume of transferred data will reach the threshold where the scale heavily affects the total throughput (achievable bisection bandwidth) and the fairness of the bandwidth allocation between communicating node pairs. This argument is valid on all systems with a fat-tree network topology. On the Cray XT system, preliminary results are encouraging, but we are still lacking some experimental data due to system or communication software problems.

Figure 4(c) presents the total number of mispredictions across all concurrencies examined on the Sun Constellation system (up to 1024 processors), when using the node predictor as a baseline. For each point in the (SIZE, NMSG) space we select the best performing node level protocol and compare it to the best performing protocol at the given concurrency. The magnitude of the bars shows the number of concurrency levels where experimental data does not match the baseline predictor. For example, on the Sun system we have run experiments varying the concurrency from 32 processors to 1024 processors in powers of two, accounting for 770 experiments. For this data set, the baseline node predictor is able to choose the best performing protocol in 93.7% of the cases. Similar results are observed for the *nearest neighbor* topology and on all systems. For all systems and communication topologies the protocol differences are clustered on the protocol switch boundary for the baseline node predictor. Figure 4(d) illustrates the magnitude of performance loss when using the baseline predictor at 1024 processors concurrency.

The internal packing and pipelining threshold within GASNet is static with $AMSize = 1500$ *bytes* and of particular interest are the differences clustered at the lines SIZE=128 (1024 *bytes*) and SIZE=256 (2048 *bytes*). The mispredictions occurring at these values across all systems indicate that performance benefits can be achieved by further tuning the packing strategy of contiguous messages around this threshold size.

Based on the experimental data, we believe that we can safely discard both communication topology and concurrency level as determining performance factors and use only node based predictors. We conclude that instantaneous Network Interface Card load is the determining performance factor for *vector* operation implementations.

6.2 Building a Predictor

Our predictors are based on the relative performance charts obtained for a two processor micro-benchmark run and for a two node micro-benchmark run. We use different predictors for intra-node and inter-node communication. For a given vector operation we extract the total number (N) of disjoint memory regions and the size (S) of the contiguous region. We then select the fastest implementation for the $(2^{\lceil \log(S) \rceil}, 2^{\lceil \log(N) \rceil})$ point. All the results in this paper are obtained using this classification scheme.

We consider several predictors that combine intra-node and inter-node processor (P) and node (N) profiles. In the rest of this paper we label predictors as *intra-inter*, based on the profile used. For example a predictor labeled *P-N* will use the processor profile for intra-node communication and the node profile for inter-node communication.

All the experimental results for the application workload have been obtained using pessimistic predictors obtained by examining the performance the slowest communicating processor pair (**maximum** time). We have examined also sev-

tion.

[3]Previous studies and our own experience indicate that scale effects are observable only for large transfers (MB). Bandwidth saturation occurs at 64KB messages on the InfiniBand networks and at ≈ 200KB on the Federation network.

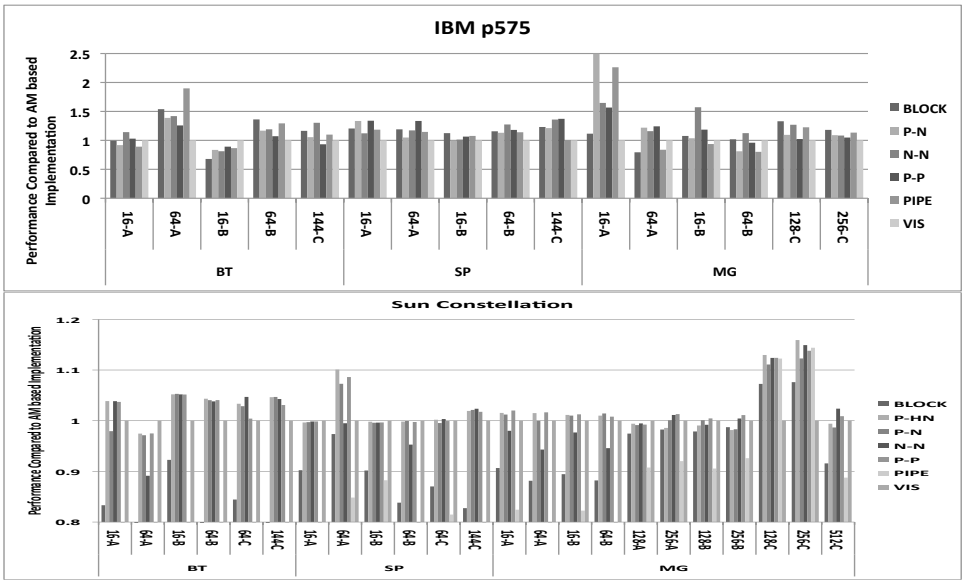


Figure 6: Performance of benchmarks using dynamic optimization techniques compared to the performance of AM based implementations. Values higher than 1 indicate performance improvements. Both systems contain wide SMP nodes.

eral "optimistic" predictors obtained under lighter system load or using the fastest execution (**minimum** time) observed across all pairs.

Figure 5(a) shows the points in the (SIZE, NMSG) space where the pessimistic predictor disagrees with the optimistic (**minimum**) predictor on the best performing protocol on the Sun system for a two nodes experiment. While the optimistic node based predictor has a structure similar to Figure 3(c), the pessimistic predictor, the protocol choices are different and the former does not choose the blocking implementation in any case. Figure 5(b) shows the distribution of mispredictions with increasing concurrency when using the optimistic node predictor as a baseline. Note that the differences are not clustered on boundaries any longer and appear in the region where the optimistic predictor chooses the pipelined implementation. Similar behavior is observed on the wide node IBM p575 cluster.

We believe that previous performance modeling efforts [9, 23] are able to predict well only on parts of the whole parameter space and have a behavior similar to the optimistic predictors. Intuitively, the misprediction space of previous approaches is captured by the "union" of Figures 5(a) and (b).

6.3 Application Results

We validate our optimization framework using the MG, SP and BT application kernels from the NAS [16] Parallel Benchmarks suite. We have a complete set of results for all systems except the Cray XT system where we have encountered problems with the software stack.

We compare the performance of blocking (BLOCK), pipe–lined (PIPE), AM based packing (VIS) and dynamically optimized implementations using our predictors. All versions are based on the officially released UPC implementation [28] of the NAS benchmarks, which we use as a performance baseline. All baseline implementations use blocking communication. All the performance models and heuristics described in this paper have been integrated in the Berkeley UPC compiler and runtime and are fully automated. For a detailed description of the optimization infrastructure and application level optimizations applied see [11].

The benchmarks exhibit different characteristics. In MG, the communication granularity varies dynamically at each call site. SP issues requests (*Put*) to transfer a variable number of mid-size contiguous regions. The requests in BT (*Put* and *Get*) vary from small to medium sizes. For all benchmarks, the count and granularity of messages varies with problem class and system size. Table 2 illustrates these differences for a class B problem running on 64 processors. Vetter and Mueller [29] indicate that large scientific applications show a significant amount of small to mid-size transfers and all the benchmark instances considered in this paper exhibit this characteristic.

Figure 6 presents the performance results obtained on the Sun Constellation and the IBM systems which contain the wide SMP nodes. We report P_{impl}/P_{VIS}, where P_{VIS} represents the performance of the AM based implementation. We use for the comparison the performance in *operations per second* as reported by the benchmarks. Values larger than one signify performance improvements. Each benchmark has been run at least for ten times on each system and we report the average performance across all runs. Since our technique tries to eliminate the performance degradation and noise due to congestion, we believe that the average performance reflects best its benefits. In general our technique improves both on the best performance of a benchmark setting and the variability of execution times across different runs. The impact is more pronounced on the systems with "wide" SMP nodes. Of particular interest is the IBM p575 cluster which exhibits a very noisy performance behavior and the impact of our techniques is most pronounced.

On the Sun Constellation system, best behavior is produced by the static *P-P* and *P-HN*[4] predictors which improve overall performance by 2.5% across the whole workload, with maximum improvements of 15% for some benchmark settings. Relatively similar behavior is observed on the

[4]HN stands for half-node.

thin node InfiniBand cluster. On the IBM p575 cluster we have performed experiments where notification of communication operations is performed using either interrupts or a polling implementation. The interrupt based implementation produces a higher node load and lower end-to-end performance for all benchmarks. The best behavior is produced by the static *N-N* predictor which improves performance by 22% across the whole workload. For the polling based implementation, our approach improves performance by 17% across the whole workload. Some benchmarks ran two to three times faster on this system.

On the IBM p575 system, node *N-N* based predictors offer best performance in all cases. The software stack instantiates hidden helper threads and nodes are over-committed at any time during the execution. On this system, our implementation will use a static *N-N* predictor in all cases. On the Sun Constellation system, node based (*N-N, P-N*) predictors offer better performance at lower concurrency. At high concurrency, predictors obtained for lighter node load (*P-P, P-HN*) offer better performance. At high concurrency, heavy node load is harder to sustain since load imbalance is likelier to occur (e.g. threads leave barriers at different times) and protocol traffic is likelier to be delayed inside the network. In our implementation we statically use the *P-HN* predictor. We believe that on this system choosing a predictor dynamically based on instantaneous node load estimation will not improve performance significantly. We plan to examine dynamic predictors in the near future.

Figure 7 shows the performance impact of using an "optimistic" predictor obtained for a benchmark setting with lighter node load. In this case *barrier* operations have been inserted after the measurements for each setting of the (S,N) parameters. On the Sun system, our dynamic optimization technique provides performance across the whole workload that is indistinguishable from the static VIS implementation. Similar results are observed on the IBM system. Using predictors obtained for even more optimistic behavior such as a *minimum* based predictor results in workload performance inferior to that of a static VIS protocol. We have also experimented with predictors obtained at different concurrency. Our results indicate that predictors tailored for the concurrency level do not improve the performance of the considered workload beyond the level obtained with node predictors.

The overall benchmark results illustrate the pitfalls of achieving performance portability across different system architectures or even on the same system at different scales and demonstrate the benefits of model based multi-protocol implementations for *vector* operation and in general for communication generation. For example, on the IBM system, each static implementation (blocking, pipelined or AM based) performs best in some case. On the Sun Constellation system, pipelining messages does not perform well at all when whole nodes are used. On the thin node systems, blocking communication does not perform well. Table 2 illustrates these trends by showing the protocol choices on the Sun Constellation system for selected (class B, 64 processors) instances of the benchmarks. Our approach is capable of improving the performance when compared to any single static *vector* implementation and our results indicate that each available protocol has been chosen for some application setting.

(N,S)	P-P	N-N	P-HN	P-N	Count	Max
(26, 65)	V-P	B-B	V-B	V-B	23684	3.4 V/B
(13, 65)	P-P	B-B	P-B	P-B	4408	6.7 V/B
(169, 65)	V-V	B-V	V-V	V-V	1407	0.86 V/B
(169, 5)	V-V	V-V	V-V	V-V	2703	1
(169, 25)	V-V	V-V	V-V	V-V	2923	1
(5, 578)	B-V	B-B	B-B	B-B	5628	12 V/B
(65, 34)	V-V	V-V	V-V	V-V	5979	1
(5, 338)	P-P	B-B	P-B	P-B	2807	8.5 V/B
(120, 13)	V-V	B-V	V-V	V-V	3246	1
(65, 66)	V-V	B-B	V-V	V-B	134	1.05 V/B
(33, 34)	V-V	B-V	V-V	V-V	126	1
(17, 18)	V-V	B-B	V-B	V-B	126	1.65 V/B
(9, 10)	V-P	B-B	V-B	V-B	126	1.42 V/B
(5, 6)	B-P	B-B	B-B	B-B	126	1.44 V/B
(3, 4)	B-P	B-B	B-B	B-B	126	2.36 V/B
(2, 3)	B-B	B-B	B-B	B-B	126	2.07 V/B

Table 2: Sample protocol choice for BT (top), SP (mid) and MG (bottom) on the Sun Constellation system. Class B, 64 processors. N = number of messages, S = message size in doubles. Count = approximate number of calls per process. Max = the maximum performance impact as reported by our node predictor for inter-node communication. We report the time for the VIS implementation compared to the time of the dominant protocol on the line.

7. DISCUSSION

There are several directions where our work can be extended. One direction is to provide more accurate process and node predictors by increasing the sampling in the (*size, number of messages*) space and providing neighbor based interpolation for points in this space. We have eliminated communication topology and concurrency level as important parameters of our performance models and the current experimental results seem to validate this choice. The experimental data indicates that mispredictions infrequently occur with the increasing concurrency level at the protocol boundaries in the (S, N) space. Increasing concurrency beyond the level validated in this paper (1024 processors) might require predictor refinement. We believe that we can still synthesize only a very limited number of predictors using two possible approaches. One approach is to choose the profile with the least potential performance impact in case of misdiagnosing the instantaneous setting. For example, choosing for each point in the (S,N) space the prediction with the least performance impact at any concurrency level might result in a good overall predictor. Another simpler approach suggested by the existing data is to determine ranges of concurrency and refine the predictors along protocol boundaries.

The applications we examined are written in a relatively bulk synchronous manner and our static predictors were able to provide good performance results. For more asynchronous and load unbalanced applications dynamic load estimation might be required. However, note that the currently accepted notion of load balance refers to the amount of work per processor and most applications currently deemed as load imbalanced still tend to exchange data in lock-steps. Our current implementation allows developers to select the

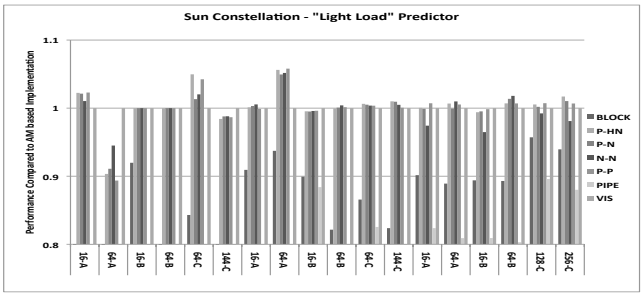


Figure 7: Impact of predictor choice on overall performance: predictor obtained for a "lighter" node load.The micro-benchmark used measures performance for each parameter setting (S,N) in isolation, i.e. contains *barriers* after each setting. Similar trends are observed with MIN based predictors.

predictors and the principles we present are simple and intuitive.

Most applications that use dense data structures tend to perform *vector* operations that transfer same sized data regions. Even more "dynamic" applications, such as an UPC Computational Fluid Dynamics benchmark which has been extracted from an Adaptive Mesh Refinement framework exhibits this characteristic. As future work, we will extend our approach for applications where communication is irregular, e.g. sparse data structures.

Our current implementation is driven by a compiler and implemented as a wrapper around communication library interfaces. An interesting question is where do mechanisms similar to ours belong: at the communication library level or at the Network Interface Card driver level. We believe that these mechanisms should be incorporated at driver level for several reasons. Instantaneous load estimation is easiest to perform and changes in system or communication library (e.g. MPI or GASNet) software do not usually permeate to that level. For example, the IBM p575 system is a dual-rail system and software changes in the communication striping algorithm might require re-tuning of the predictors used. Incorporating the mechanisms at driver level will alleviate the need for re-tuning when software changes happen.

Although our methodology has been validated only for a one-sided communication paradigm, we believe that the approach is directly applicable to two-sided communication paradigms such as MPI and the implementation of scatter-gather operations and MPI Derived Data Types could benefit from it.

8. CONCLUSION

Previous work on optimizing *vector* style operations has concentrated on clusters composed of thin SMP nodes. Several performance models have been proposed to select the best performing implementation for a given application setting. A common trait of previous studies is that the models seem to fail in the presence of wide SMP nodes. In wide SMP systems applications can generate a heavier load on the Network Interface Cards and can possibly generate steady state operation in a congested regimen.

In this paper we study the performance aspects of the implementation of *vector* operations and propose a simple and intuitive methodology for the tuning of these operations in a multi-protocol setting. Our results show good performance improvements across a variety of large scale systems.

Our results indicate that the determining performance aspect in clusters composed of wide SMP nodes is handling of the instantaneous load on the Network Interface Cards. Our approach is based on the assumption that given multiple implementation choices there are situations where the implementation that avoids congestion will perform best even if it will under-perform in an un-congested setting.

We believe that our results are of interest to both communication library and application developers.

Acknowledgments: The authors would like to thank James Browne at TACC for the access to the Sun Constellation system (ranger).

9. REFERENCES

[1] A. Alexandrov, M. F. Ionescu, K. E. Schauser, and C. Scheiman. LogGP: Incorporating Long Messages into the LogP Model for Parallel Computation. *Journal of Parallel and Distributed Computing*, 44(1):71–79, 1997.

[2] Bassi IBM p575 POWER5. LBNL National Energy Research Supercomputing Center.

[3] Bigben Cray XT3 MPP. Pittsburgh Supercomputing Center.

[4] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick. Optimizing Bandwidth Limited Problems Using One-Sided Communication and Overlap. *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS)*, 2006.

[5] D. Bonachea. GASNet Specification, v1.1. Technical Report CSD-02-1207, University of California at Berkeley, October 2002.

[6] D. Bonachea. Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, v1.0. Technical Report LBNL-56495, Lawrence Berkeley National Laboratory, 2004.

[7] S. Byna, W. D. Gropp, X.-H. Sun, and R. Thakur. Improving the Performance of MPI Derived Datatypes by Optimizing Memory-Access Cost. In *IEEE International Conference on Cluster Computing*, 2003.

[8] Co-Array Fortran - Technical Specification. Available at http://www.co-array.org/caf_def.htm.

[9] K. Cameron, R. Ge, and X.-H. Sun. lognP and log3P: Accurate Analytical Models of Point-to-Point Communication in Distributed Systems. *IEEE Transactions on Computers*, 2007.

[10] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey. A Multiplatform Co-array Fortran Compiler. In *Proceedings of the IEEE Parallel Architecture and Compilation Techniques Conference (PACT)*, Antibes Juan-les-Pins, France, 2004.

[11] C. Iancu, W. Chen, and K. Yelick. Performance Portable Optimizations for Loop Containing Communication Operations. In *Proceedings of the 2008 ACM International Conference on Supercomputing (ICS'08)*, 2008.

[12] C. Iancu and E. Strohmaier. Optimizing Communication Overlap for High-Speed Networks. *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[13] Jacquard AMD Opteron cluster. LBNL National Energy Research Supercomputing Center.

[14] N. Koziris, A. Sotiropoulos, and G. I. Goumas. A Pipelined Schedule to Minimize Completion Time for Loop Tiling with Computation and Communication Overlapping. *Journal of Parallel and Distributed Computing*, 63(11):1138–1151, 2003.

[15] M. Krishnan and J. Nieplocha. Optimizing Performance on Linux Clusters Using Advanced Communication Protocols: Achieving Over 10 Teraflops on a 8.6 Teraflops Linpack-Rated Linux Cluster. *Proceedings of the 6th International Conference on Linux clusters: The HPC Revolution*, 2005.

[16] The NAS Parallel Benchmarks. Available at http://www.nas.nasa.gov/Software/NPB.

[17] J. Nieplocha and B. Carpenter. ARMCI: A Portable Remote Memory Copy Library for Distributed Array Libraries and Compiler Run-Time Systems. *Lecture Notes in Computer Science*, 1586:533–??, 1999.

[18] J. Nieplocha, R. Harrison, and R. Littlefield. Global Arrays: A Non-Uniform Memory Access Programming Model for High-Performance Computers. In *The Journal of Supercomputing*, volume 10, 1996.

[19] J. Nieplocha, V. Tipparaju, and M. Krishnan. Optimizing Strided Remote Memory Access Operations on the Quadrics QsNetII Network Interconnect. In *HPCASIA '05: Proceedings of the Eighth International Conference on High-Performance Computing in Asia-Pacific Region*, page 28, Washington, DC, USA, 2005. IEEE Computer Society.

[20] R. Numrich and J. Reid. Co-Array Fortran for Parallel Programming. Technical Report RAL-TR-1998-060, Rutherford Appleton Laboratory, 1998.

[21] Ranger SUN Constellation linux Cluster. Texas Advanced Computing Center, University of Texas at Austin.

[22] G. Santhanaraman, D. Wu, and D. K. Panda. Zero-Copy MPI Derived Datatype Communication over InfiniBand. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface, 11th European PVM/MPI Users' Group Meeting*, 2004.

[23] J. Su and K. Yelick. Automatic Support for Irregular Computations in a High-Level Language. In *Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05)*, 2005.

[24] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *Proceedings of the tenth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming, (PPoPP'05)*, 2005.

[25] V. Tipparaju, M. Krishnan, J. Nieplocha, and D. P. G. Santhanaraman. Exploiting Non-blocking Remote Memory Access Communication in Scientific Benchmarks. *Proceedings of the 2003 International Conference on High Performance Computng, HiPC'2003*, 2003.

[26] V. Tipparaju, G. Santhanaraman, J. Nieplocha, and D. K. Panda. Host-Assisted Zero-Copy Remote Memory Access Communication on InfiniBand. In *18th International Parallel and Distributed Processing Symposium*, 2004.

[27] K. Underwood, M. Levenhagen, and R. Brigthwell. Evaluating NIC Hardware Requirements to Achieve High Message Rate PGAS Support on Multi-Core Processors . In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC07)*, 2007.

[28] UPC Language Specification, Version 1.0. Available at http://upc.gwu.edu.

[29] J. Vetter and F. Mueller. Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures. *Proceedings of the 2002 International Parallel and Distributed Processing Symposium (IPDPS)*, 2002.

[30] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *19th International Symposium on Computer Architecture*, pages 256–266, Gold Coast, Australia, 1992.

[31] K. Yelick, L. Semenzato, G. Pike, C. Miyamoto, B. Liblit, A. Krishnamurthy, P. Hilfinger, S. Graham, D. Gay, P. Colella, and A. Aiken. Titanium: A High-Performance Java Dialect. In *Proceedings of the ACM 1998 Workshop on Java for High-Performance Network Computing*. ACM Press, 1998.