Kernel Assisted Collective Intra-node Communication Among Multicore and Manycore CPUs

Teng Ma*, George Bosilca*, Aurelien Bouteiller*, Brice Goglin†, Jeffrey M. Squyres‡, Jack J. Dongarra§
* *EECS, University of Tennessee*
1122 Volunteer Blvd., Knoxville, TN 37996-3450, USA
Email: {tma, bosilca, bouteill}@eecs.utk.edu
† *INRIA Bordeaux - Sud-Ouest, University of Bordeaux*
351 cours de la Libération, F-33405 Talence cedex, France
Email: Brice.Goglin@inria.fr
‡ *Cisco Systems, Inc.*
San Jose, CA USA
Email: jsquyres@cisco.com
§ *University of Tennessee*
Oak Ridge National Laboratory, Oak Ridge, TN, USA
University of Manchester, Manchester, UK
Email: dongarra@eecs.utk.edu

***Abstract*—Even with advances in materials science, fundamental limits in heat and power distribution are preventing higher CPU clock frequencies. Industry solutions for increasing computation speeds have concentrated on raising the number of computational cores available, leading to the wide-spread adoption of so-called "fat" nodes. However, keeping all the computation cores busy doing useful work is a challenge because typical high performance computing (HPC) workloads require reading and writing a steady stream of data from memory – contention for memory bandwidth becomes a bottleneck. Many commodity platforms have therefore embraced non-uniform memory access (NUMA) architectures that split up and distribute memory to be close to the cores.**

High-performance Message Passing Interface (MPI) implementations must exploit these architectures to provide reliable performance portability. NUMA architectures not only require specialized MPI point-to-point messaging protocols, they also require carefully designed and tuned algorithms for MPI collective operations. Multiple issues must be taken into account: 1) minimizing the number of copies required, 2) minimizing traffic to "remote" NUMA memory, and 3) carefully avoiding memory bottlenecks for "rooted" collective operations.

In this paper, we present a kernel assisted intra-node collective module addressing those three issues on many-core systems. A kernel level inter-process memory copy module, called KNEM, is used by a novel Open MPI collective module to implement several improved strategies based on decreasing the number of intermediate memory copies and improving locality to reduce both the pressure on the memory banks and the cache pollution. The collective topology is mapped onto the NUMA topology to minimize cross traffic on inter-socket links. Experiments illustrate that the KNEM enabled Open MPI collective module can achieve up to a threefold speedup on synthetic benchmarks, resulting in a 12% improvement for a parallel graph shortest path discovery application.

***Keywords*-MPI, multicore, shared memory, NUMA, kernel, collective communication**

I. Introduction

In the recent years, thermic issues have been preventing the straightforward performance improvement from an increase in the operating frequency of the processors. This has led to the wide adoption of parallelism in a chip, in the form of multicore processors, to continue fulfilling Moore's law expectations of exponential performance increase over time. This trend is even more pronounced when considering the systems of the TOP500 list of supercomputers [1], which are expected to feature, in a close future, *fat* manycore nodes composed of as many as a hundred of cores. While multiplying the number of cores increases the peak performance, to keep that many processing units busy, a significant amount of data needs to be pumped from the memory. The flat memory bus, as featured in many legacy Symmetric Multi Processors (SMP) *north-bridge* chipsets, would not have been able to sustain such a bandwidth and request throughput, practically limiting the achievable performance to a deplorable portion of the computing peak – a condition known as *the memory wall*. To dodge this problem, most recent multicore designs embrace Non Uniform Memory Access (NUMA) and hierarchical memory interconnects to enable core count scalability and an adequate bandwidth between the cores and the memory banks; but at the expense of an excruciating programming complexity. The Message Passing Interface (MPI)[2] has been the dominant programming model for High Performance Computing (HPC) applications for the last decade. However, MPI applications that are oblivious of the NUMA topologies and the associated performance traps are bound to suffer from unacceptable performance, because they generate a load pattern on the memory subsystem that

crashes into the memory wall. While the MPI programming model is expressive enough to enable a mapping between the underlying shared memory topology and the application communication pattern, such an approach calls for the modification of every code on every platform, defeating one key feature that have empowered the prevalence of MPI: performance portability. During the ending era of the distributed memory machines composed of single processor nodes, the complexity and performance difficulties posed by hierarchical network interconnects have been tackled inside the implementation of the collective communications, a set of standardized routines expressing the most common communication patterns. We believe that a similar approach should be undertaken to preserve performance portability in the coming era of the distributed system featuring manycore shared memory nodes.

Most shared-memory message passing implementations, such as the Nemesis [3] device in MPICH2 [4] and the SM component in Open MPI [5], depend on a double memory copy scheme. An extra shared memory buffer is pre-allocated as an exchange zone between processes. Every message is copied to this intermediate zone by the sender process, and then copied to the destination buffer by the receiver process. We have identified several critical issues with this approach on manycore systems. First, with many cores, the *root* process in one-to-all or all-to-one collective operations becomes a performance bottleneck. For one-to-all communications such as MPI_Bcast and MPI_Scatter, every other process needs to copy from the root process; actually, the algorithm based on double memory copy is sequentialized by the progression of the root process. The opposite case exists for all-to-one operations such as MPI_Gather. Therefore, the scalability is poor, especially for large messages. Second, communications based on double memory copy method induce more cache pollution due to each transferred byte being actually loaded and written twice. As one can expect, more cache pollution in turn leads to a plummeting memory bandwidth as more cache lines are reclaimed from the slow and contention-prone memory banks instead of the local cache. Last, most implementations ignore topological characteristics, such as NUMA and network-style processor interconnects. The blind application of the *one size fits all* collective algorithm leads to an unreasonable cross traffic between sockets and overwhelm some memory links while under-utilizing some others.

In this paper, we propose new MPI collective communication algorithms of MPI_BCAST, MPI_GATHER, MPI_GATHERV, MPI_SCATTER, MPI_SCATTERV, MPI_ALLTOALL, MPI_ALLTOALLV, MPI_ALLGATHER, and MPI_ALLGATHERV that take advantage of the NUMA memory subsystem, to avoid memory contention, and to maximize the overall sustained bandwidth. Our approach is based on the KNEM Linux module – a software mechanism that enables direct memory copying between processes. We investigate several different algorithms that maximize both parallelism and pipelining, and are NUMA topology-aware. A key point in the design is that multiple processes can access the same buffer – or different parts of the same buffer – simultaneously, without the need for more than a single memory copy between processes. Moreover, stream direction control enables the collective algorithm to select sender-writing or receiver-reading according to the communication pattern (all-to-one or one-to-all) to avoid a root process bottleneck. Last, our collective algorithms can detect distance between hardware units to build an optimized communication topology that reduces the cross-traffic on the weakest memory links. Each of these approaches are evaluated experimentally on a variety of different hardware setups, exhibiting substantial performance gains that increase with the number of cores.

The rest of this paper is organized as follows: Section II introduces related work on intra-node collective operations and kernel-assisted memory copy. Section III introduces some key concepts of the KNEM kernel copy framework. Then, Section IV describes the linear KNEM collective algorithms: one-to-all (Broadcast and Scatter), all-to-one (Gather), all-to-all (Alltoall and Allgather) and an implementation in a new collective module for a leading MPI implementation (Open MPI). Section IV presents a theoretical performance analysis outlining the differences between linear KNEM collectives and Open MPI's basic set of linear collective algorithms. Next, Section V discusses the simultaneous use of KNEM-based collective algorithms with a NUMA topology aware hierarchical layout. All those algorithms are compared experimentally with state of the art implementations of double-copy algorithms in Section VI. Finally, Section VII concludes the paper with a discussion of the results and future directions.

II. Related Work

Several optimizations have been used to maximize throughput of collective communications on shared memory nodes. Most of them have been based on adopting different communication topologies (linear, chain, split binary tree, binomial tree and etc.) [6] and by enabling parallel treatment of the message through pipelining. Both MPICH2 and Open MPI feature many of those algorithms and select among them with highly tuned and optimized switch-points based on the message size. Consequently, they deliver good performance on SMP nodes, even though all those algorithms rely on the double memory copy shared memory transport device. However, this last aspect is greatly challenged by the multiplication of the number of cores in currently deployed *fat* supercomputer nodes.

One of the most recent of those efforts is due to Richard Graham et al. [7], who proposed a shared memory-based fan-in/fan-out implementation for multi-core MPI collectives, implemented in the Open MPI SM collective component.

Their optimization focuses on lightweight synchronization, reducing memory copy times, increasing parallelism by copying messages in a pipeline way, and controlling working set size to fit into caches by building a logical fixed degree tree. This shared memory based method simply takes multi-core/many-core as a SMP system and ignores other architecture characteristic, such as NUMA, memory hierarchy and core distance. Especially for large messages, too many memory copies generated by the pipeline algorithm easily overflow the capacity of the memory controller, which decreases achieved memory throughput. Their fixed degree tree is built following logical ranks' layout, which cannot always reflect architecture characteristics. With more heterogeneity in modern NUMA multicore designs, it is hard to optimally tune a shared memory based implementations for different platforms.

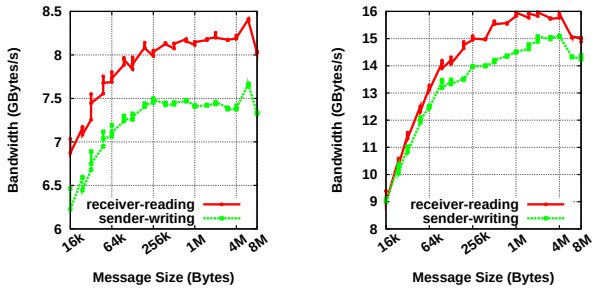
Open MPI also feature another interesting intra-node collective component named *tuned* collectives. The fundamental idea of the tuned collectives, described in the paper [8], is to make available several different algorithms and use a runtime decision to select the best algorithm according to message size, communicator size, and other parameters. As an example, for a Broadcast in the *tuned* collective module, a binomial algorithm is used to deliver small messages, a split binary tree algorithm is selected for intermediate messages, and large messages are transferred by a pipeline algorithm in which the pipeline size varies with the message and communicator size. However, it is still hard to tune for an unknown platform, even for expert developers. Indeed, there are too many parameters such as pipeline size, thresholds, etc, and any wrong selection might ruin the overall performance of the *tuned* collectives.

All the previous approaches are orthogonal works to the proposed ideas of this paper. They try to maximize the throughput of the collective operation by developing new collective topologies, or selecting among the available algorithms the most suitable one. For those approaches to reach their full potential, there is a need to cooperate with another approach to alleviate the penalty due to heterogeneous NUMA architectures, a feature that our kernel assisted approach is able to deliver.

Several platform-specific efforts offered single-copy large message communication. For instance BIP-SMP implemented such an optimization for Myrinet based clusters [9]. This idea has spread into most vendor specific HPC stacks, such as Myricom MX, Qlogic ipath and Bull MPI. Cray platforms enabled an even bigger rework of the model thanks to the ability of their custom lightweight operating system to make all process' address spaces accessible to any of them. This unusual feature enabled single-copy RMA-based communication (SMARTMAP [10]) which greatly reduces memory copies needed by intra-node message passing, especially for collectives. However, it's impossible to apply their SMARTMAP strategy into common Linux/Unix operating systems.

Lei Chai and etc. [11] introduced a kernel module interface called LiMIC. This kernel-based approach can reduce the number of necessary memory copies to one. KNEM [12] is another similar kernel module used in MPICH2 and Open MPI. KNEM offers additional features such as an asynchronous copy model, vectorial buffer support, and copy offload on dedicated hardware. In this paper, we will present new collective algorithms based on KNEM copy, which is a single-copy method. The details of KNEM copy will be introduced in the section III

III. KNEM



(a) on Tigerton (b) on Istanbul

Figure 1. Point-to-Point Communication Bandwidth of KNEM Copy on Intel Tigerton and AMD Istanbul Platforms.

KNEM [13] is a Linux kernel module enabling high-performance intra-node large message passing. It offers support for asynchronous and vectorial data transfers as well as offloading memory copies on to Intel I/O AT hardware. MPICH2 since release 1.1.1 uses KNEM in the DMA LMT to improve large message performance within a single node. Open MPI also includes KNEM support in its SM BTL component since release 1.5. These works focused on point-to-point communication within a single node. Early details of KNEM copy model are described in Darius Buntinas and etc's paper[12] and may be summarized as: (1) The sender process declares a send buffer to KNEM. The driver saves the list of virtual segments contained in the buffer and associates them with a unique cookie. (2) The sender passes the cookie to the process which is interested in this buffer by an out-of-band transfer. (3) The receiver gives this incoming cookie to KNEM along with a receive buffer. (4) KNEM device moves the data from the send buffer to the receive buffer within the kernel.

Brice Goglin and etc. [14], [12] gave out the performance analysis of using KNEM in point-to-point communication. Using KNEM copy for large messages can achieve comparable performance with MPICH2 default LMT between two processes sharing L2 cache, and beat default LMT between two processes not sharing any cache for the messages from 64KBytes to 2MBytes on their platform. Single-copy and

cache efficient advantage in KNEM makes KNEM-based point-to-point communication contributed more performance to NAS benchmarks, especially in IS.

Figure 1 gives out the KNEM point-to-point communication on our two platforms: Zoot (quad-socket quad-core Intel Tigerton) and IG (8-socket six-core AMD Istanbul, depicted on Figure 3). This experiment uses the KNEM module in NetPIPE release 3.7.2 [15] with the off-cache option enabled. By default the KNEM backend in NetPIPE uses a receiver-reading method, but KNEM also supports sender-writing (since release 0.7) The former consists in the target process CPU pulling data from the sender, while in the latter the source process CPU pushes data to the receiver. In this experiment, we modified the KNEM backend of NetPIPE to also support sender-writing. From Figure 1,we can see that receiver-reading is little better than sender-writing on both platforms. One reason for this difference lies in the actual copy implementation in the Linux kernel which is more optimized in the receiver-reading scheme.

KNEM 0.7 introduced a new API that lets users declare regions and reuse them or part of them multiple times. Combined with the ability to change the copy direction at runtime according to access pattern, these features are the key to implementing a KNEM-based intra-node collective component.

IV. Implementation of linear KNEM collectives

We implemented KNEM-based collectives as a new component named 'KNEM' collectives in Open MPI. The main speedup of this component comes from new collective algorithms, which make use of KNEM RMA-based API to achieve more parallelism than algorithms based on double memory copy. Firstly, we introduce the implementation of linear KNEM collective in KNEM collectives.

Contrary to the user-space two-copy model, the Linux kernel has knowledge of mapping between virtual memory and physical memory. So a real RMA-based memory copy is possible thanks to the kernel and it does not need pre-allocate any intermediate shared memory buffer. However, trapping into kernel mode has a non-negligible overhead (about 100 ns on modern processors) when delivering small messages. So we only consider using KNEM to optimize collectives for intermediate and large messages (larger than 16KBytes). Architecture characteristics are also considered so as to reduce inter-socket traffic in Section V.

Our linear KNEM collective implementation covers MPI_Bcast, MPI_Gather, MPI_Gatherv, MPI_Scatter, MPI_Scatterv, MPI_Alltoall, MPI_Alltoallv, MPI_Allgather, and MPI_Allgatherv. We now detail the most meaningful ones.

A. KNEM Broadcast

The implementation of KNEM Broadcast is a straightforward adaptation of the KNEM point-to-point model: (1) The root process declares a send buffer to KNEM and gets the corresponding cookie in return. (2) It passes the cookie to all non-root processes in the communicator through an out-of-band transfer (point-to-point send/receive with Open MPI small message double-copy implementation). (3) Each receiver process passes the incoming cookie to KNEM along with a receive buffer. (4) KNEM triggers a memory copy from the send buffer to receive buffer within the kernel.

B. KNEM Gather and Scatter

The KNEM Scatter implementation may be derived from the above Broadcast. Each non-root process however reads only part of the root buffer. Their starting offset is calculated from its rank and data type.

Then, the KNEM Gather consists in the opposite data transfer. The root process declares its buffer with *write* enabled and all non-root processes will write there instead of reading.

C. KNEM Alltoall and Allgather

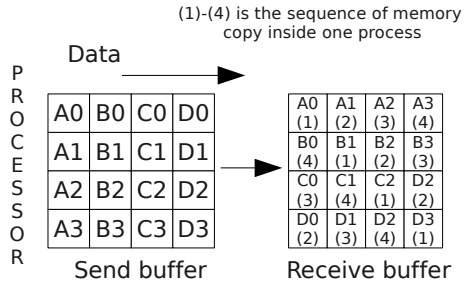


Figure 2. An Example of Copy Sequence for KNEM Alltoall on four processors.

The KNEM Alltoall implementation has to pre-allocate an integer array to store cookies from all other processes in the communicator. First, each process declares its send buffer to KNEM and gets back the corresponding cookie. Secondly, processes in the communicator do an Allgather operation to exchange their cookies. Each cookie is stored according to process' ranks. Finally, a loop of KNEM copies is performed to fetch the corresponding messages from other nodes (receiver-reading). The offset on the remote nodes is derived from local process' ranks. Additionally, Alltoall requires to copy the local sender buffer into the local receiver buffer. The corresponding offset which can be calculated by the product of its rank and data type size and count. To avoid having all processes concurrently access the same source buffer simultaneously, they actually start by doing their local copy and then copy messages according to rank ordering in a round-robin manner like Figure 2.

The KNEM Allgather adopts a simple method which combines Broadcast and Gather together. During the first step, all processes do a KNEM Gather operation with rank 0 as the root process. Then the second step consists in the root process doing a KNEM Broadcast to others in the communicator. This method is far away from a optimal method, because the memory controller belonged to the root process will always be in a big pressure, our KNEM Allgather still has room for improvement.

D. Theoretical Analysis

Table I shows a theoretical analysis of linear KNEM algorithms and basic algorithms in Open MPI. Here, T is execution time of one memory copy, N is the number of processes in the communicator and M is message size.

In linear KNEM algorithms, every non-root process can execute memory copy at the same time for Broadcast, Gather and Scatter. So their runtime complexity is O(T). For Alltoall, each process executes N memory copy, so the overall time complexity is O(NT). For Allgather, the linear KNEM algorithm combine Broadcast and Gather together which both have a O(T) time complexity.

In basic Open MPI algorithms, the root process will copy memory by $N-1$ times to/from shared memory buffer for Broadcast, Gather or Scatter. And then each non-root process can copy messages from shared memory buffers. So the time complexity of basic algorithms for these operations is O(NT). For basic Alltoall, each process executes $2N-1$ times memory copy including $N-1$ copy-out, $N-1$ copy-in, and 1 memory copy to itself. Although it is also O(NT) for Alltoall in basic algorithms, the constant for NT in the linear KNEM is smaller than in the basic algorithm. The basic Allgather also requires $2N-1$ memory copies, which means O(NT).

There is no intermediate buffers needed by linear KNEM algorithms except for Alltoall operations which need an integer buffer of size of $8N$ Bytes in each process. In basic algorithms, there is one shared memory buffer needed by each pair of processes. For Broadcast, Gather or Scatter, $N-1$ message buffers are needed. For Alltoall and Allgather, $(N-1)N$ message buffers are pre-allocated.

From theoretical analysis in Table I, linear KNEM algorithms may have speedup of N on Broadcast, Gather, Scatter, and Allgather. However the practical improvement of linear KNEM algorithms is still restricted by architecture characteristics such as memory controller, cache size, and communication patterns such as the amount of inter-socket traffic. For example, the real speedup of linear KNEM algorithm in Allgather cannot be N times because its implementation puts too much pressure on root node memory controller forming a single point in the algorithm.

	Duration		Memory Usage	
Operations	linear KNEM	basic	linear KNEM	basic
Broadcast	O(T)	O(NT)	0	O(NM)
Gather	O(T)	O(NT)	0	O(NM)
Scatter	O(T)	O(NT)	0	O(NM)
Alltoall	O(NT)	O(NT)	O(N^2)	O(N^2M)
Allgather	O(T)	O(NT)	0	O(N^2M)

Table I
THEORETICAL ANALYSIS OF LINEAR KNEM ALGORITHMS AND BASIC ALGORITHMS.

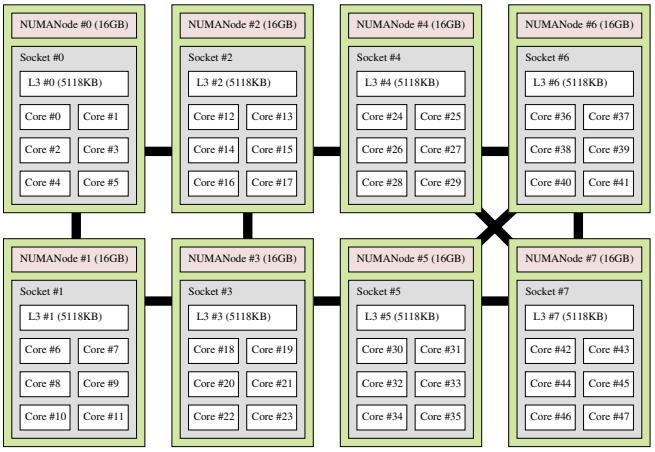


Figure 3. Architecture of IG. 8 NUMA nodes, containing 6 cores and 16GB each, are interconnected through HyperTransport links. Each core also has its own 64kB L1 and 512kB L2 cache.

V. TOPOLOGY AND NUMA AWARE KNEM COLLECTIVES WITH PIPELINE

We now present our work towards making KNEM collectives more efficient on large NUMA and manycore platforms thanks to pipelined strategies and topology and NUMA awareness. Figure 4 shows an example of hierarchical pipelined KNEM broadcast on IG. As shown on Figure 3, this machine consists in 8 NUMA nodes, each of them containing a Six-core AMD Opteron processor and 16GB of local memory. Processes in the communicator are split into 8 sets according to their NUMA localities, which means two processes within the same socket and NUMA node are in the same set. A two levels' tree is then built accordingly. One process of each non-root set is selected as intermediate peers (green nodes in Figure 4). The other processes in each set are leaf nodes (blue nodes in Figure 4). This tree structure reflects the architecture topology such as core distances and relationships which can be retrieved thanks to the hwloc software (*Hardware Locality*[16]).

We now give an example of the hierarchical topology and NUMA-aware algorithm for the KNEM Broadcast. (1) As in the first step of linear KNEM Broadcast, the root process declares a send buffer to KNEM device and gets back a cookie. (2) The root process passes the cookie to its children

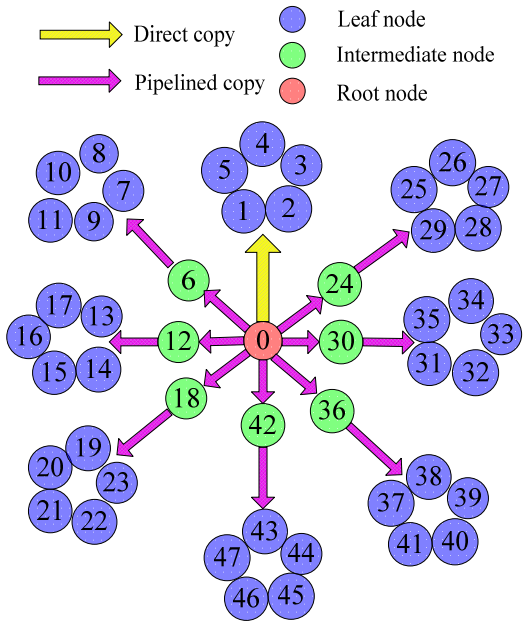


Figure 4. Working Progression of Hierarchical Pipeline KNEM Broadcast on IG

(intermediate nodes, and leaf nodes in the root set). (3) These nodes pass a receive buffer along with the cookie to the KNEM module so as to trigger a KNEM copy that fetches data from the root process. (4) Intermediate nodes then pass the just receive data to their children (leaf nodes in non-root sets) by repeating the same steps.

This topology and NUMA-aware KNEM Broadcast has the advantage of reducing inter-socket traffic since a single data transfer is performed towards each NUMA node. Moreover, since a cache is shared between all processes inside the same NUMA node, multiple copies between processes in the same set benefit from cache hits. The disadvantage of this algorithm lies in reducing the degree of parallelism, compared with linear KNEM collectives, since less processes are copying simultaneously. Also, leaf nodes behind intermediate nodes must wait for their father to receive from the root before starting their data transfer.

To reduce this unnecessary waiting on the leaf nodes, we then implemented a pipelined KNEM copy along the path from root to intermediate nodes and then from intermediate nodes to leaf nodes (pink arrows). Messages are split into several segments. Once one segment in an intermediate peer is ready, its availability is notified to leaf nodes so that the data is forwarded to them without waiting for the remaining data to arrive. So the messages will be delivered in a *pipelined* manner across all nodes of both levels. There is no pipeline for the KNEM copy between the root node and its children in the same working set (yellow arrow) since these children are actually leaf nodes in the tree. And there is no other layers' nodes waiting for them to forward the data.

The hierarchical pipelined algorithm is a more balanced implementation in which all nodes in different layers can execute KNEM copy as early as possible. This algorithm can only be applied in delivering the same message across different layers such as Broadcast. We did not add any hierarchical or pipelined scheme to Gather or Scatter operations since processes do not manipulate the same buffer chunks.

Finding a suitable pipeline size is critical to the performance of hierarchical pipelined algorithms. This parameter is decided by several factors: working set size, cache size, memory hierarchy, and memory bandwidth. We plan to add a module to the OTPO[17] project to tune pipeline size dynamically.

VI. Experimental Evaluation

A. Experimental Platforms

Our experimental platform is composed of four multicore/manycore machines ranging from Intel and AMD processors and representing a wide variety of setups from SMP to massively NUMA machines.

Zoot is a 16 core machine with 32GB of memory. The system has four sockets with a quad-core 2.40 GHz Intel Xeon Tigerton E7340 featuring 4 MB L2 caches shared between pairs of cores. A single SMP memory controller in the north-bridge chipset connects all the sockets with the global shared memory.

Dancer is an 8 core machine with 4GB of memory. The system is composed of two sockets populated with a quad-core 2.27 GHz Intel Xeon Nehalem-EP E5520 with 8 MBytes L3 caches and 2 GB of memory on each NUMA socket. Hyper-threading is disabled in the configuration.

IG is a 48 cores machine with 128GB of memory. The system is composed of 8 sockets with a six-core 2.8 GHz AMD Opteron 8439 SE, 5 Mbytes L3 caches and 16 Gbytes memory per NUMA node. The sockets are further divided as two sets of 4 sockets on two separate boards connected by a low performance interlink. The architecture is depicted on Figure 3.

Saturn is a 16 core machine with 64GB of memory. It is composed of two sockets with an octo-core 2.00 GHz Intel Xeon Nehalem-EX X7550, 18 MBytes L3 caches and 32 GBytes memory on each NUMA socket. Hyper-threading is enabled but not used.

The KNEM copy component uses the KNEM 0.9.1 [13] kernel module. The Intel MPI benchmark suite IMB-3.2 [18] was used to assert the difference between the collective modules, with the *offcache* option in enabled to avoid cache reuse. Open MPI trunk version 1.7a1 was configured to include the different collective components to be compared to.

Our proposed KNEM copy-based component (KNEM coll) is pitted against several other collective components: the Open MPI's default send/recv-based basic algorithms (basic coll), the shared memory-based component from Richard Graham et al. (SM coll), and the runtime selection of the collective algorithm according to the communicator and message sizes (tuned coll). Due to the large number of combinations, we restrict the discussion to the meaningful comparisons only. We present the performance of the KNEM collectives for most algorithms: Broadcast, Gather, Scatter, Alltoall and Allgather. Only the Broadcast and Allgather operations are presented for the hierarchical pipelined algorithm of the KNEM collective component, because those are the only one implemented as for now. Additionally, this algorithm is studied only on the IG platform, as it is specifically tailored for large NUMA nodes. The SM collective component is evaluated for the Broadcast operation only, as no other collective differs from the original in this module.

To ease the performance comparison, all execution time are normalized against the time of the Open MPI basic algorithm, which is the default algorithm without optimizations. The smaller these normalized values, the better the performance of the corresponding collective component.

B. Tuning the pipeline size

Selecting a suitable pipeline size can be a challenging problem for the hierarchical pipelined strategies. A too small pipeline size induces more synchronization between each segment, while a too large pipeline size leads to a long initialization time for the pipeline algorithm to take effect. Figure 5 presents the effect of the pipeline size on the hierarchical pipelined KNEM Broadcast on the large NUMA machine IG. The execution time of different pipeline sizes is normalized to the execution time of the basic Broadcast module. One can see that 64 KB is the best pipeline size for intermediate messages, smaller than 2 MBytes. For larger message sizes the best pipeline size varies between 128KB and 512KB. However, the difference between all the pipeline size is small and selecting the wrong setting is never catastrophic. In the rest of this paper, we settled for the pipeline size of 256KB, that performs adequately on the entire range of message sizes. In the future, parameters should enable to control the pipeline length depending on the message size.

C. Broadcast

Figure 6 shows the performance comparison of the Broadcast implementations on all platforms. The tuned collective module shows dramatic performance difference with the basic module. While on most platform it fails to select the adequate algorithm for small messages, it is able to exhibit significant speedup for large size messages. On the IG platform, which is the largest and most complex NUMA setup of our testbed, the tuned module is confused

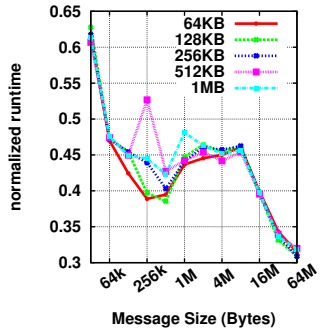
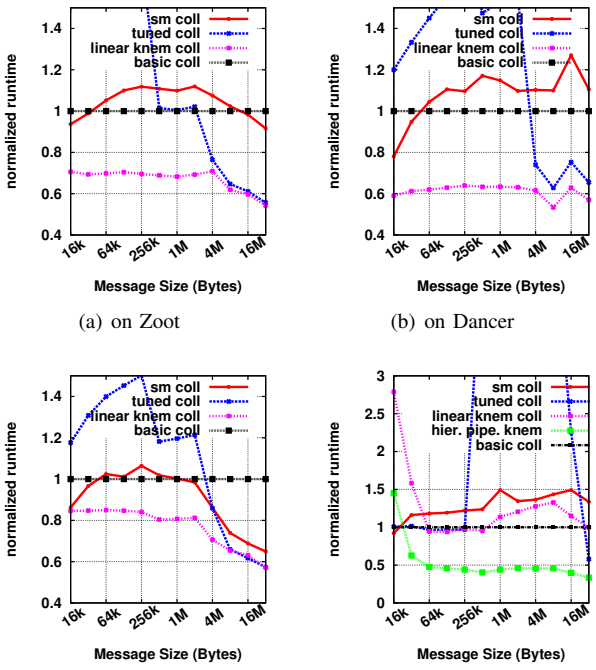


Figure 5. Performance comparison between different pipeline sizes in the hierarchical pipelined KNEM Broadcast on the IG platform. Results are normalized to the runtime of the basic module's Broadcast (lower is better).



(a) on Zoot (b) on Dancer

(c) on Saturn (d) on IG

Figure 6. Performance comparison of Broadcast operations between Basic, SM, Tuned and KNEM collective modules, normalized to the Basic module runtime (lower is better).

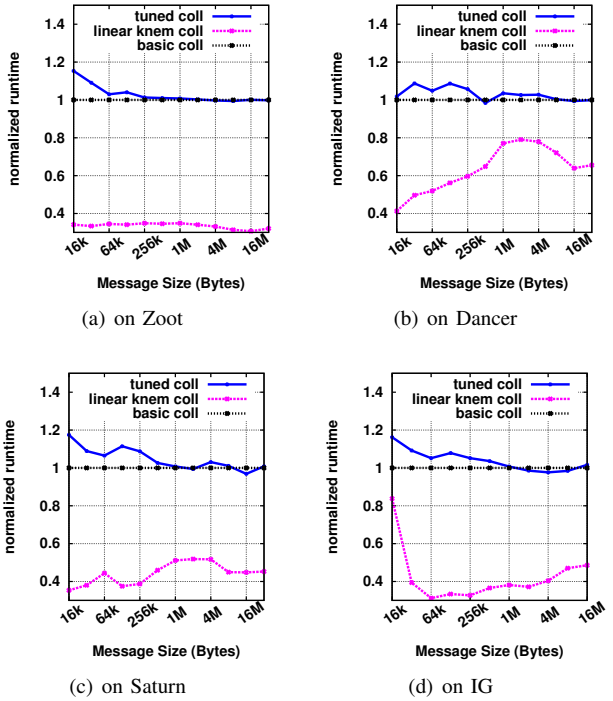
and exhibit a completely inverse behavior. It selects the basic algorithm for small messages, but fails to select the correct algorithm for message sizes between 256KB and 16MB, translating in an order of magnitude increase of the collective runtime. The selected collective topology does not match the hardware topology, which induces an unbearable pressure on the inter-socket links. The overall performance improvement of the SM collective module is more balanced, as on every machine it performs adequately, but conversely it does not yield extreme performance improvements. Overall, it is almost always outperformed by our KNEM module.

The linear KNEM Broadcast beats all other components on the SMP platform (Zoot) and on our two small size NUMA setups (Dancer and Saturn). On the large NUMA node IG, the linear algorithm does not yield significant improvement, but introducing the hierarchical pipelined optimization allows the KNEM module to outperform all other components for messages bigger than 16 KBytes. Clearly, our KNEM approach, with the option to use the hierarchical pipelined optimization, can yield larger speedup than the SM algorithm, with a simpler selection logic of the algorithm that is not subject to confusion like the Tuned module is when confronted with NUMA nodes.

Compared with basic algorithms, the maximum improvement of KNEM collectives is about 46% on Zoot, 48% on Dancer, 43% on Saturn with linear algorithms, and 67% on IG with hierarchical pipelined algorithms.

D. Gather and Scatter

Figure 7 presents the performance comparison of the Gather operation. The tuned collective module is rarely yielding improvement when compared to the basic implementation. On the complete opposite, the linear KNEM Gather tremendously outperforms all other components in all cases. Compared with the basic Gather, the maximum improvement thanks to KNEM is 69% on Zoot, 58% on Dancer, 45% on Saturn and 70% on IG, thanks to unleashing parallel access to the buffer of the root process.



(a) on Zoot (b) on Dancer

(c) on Saturn (d) on IG

Figure 7. Performance comparison of Gather operations between Basic, Tuned and KNEM collective modules. All results are normalized to the basic Gather (lower is better).

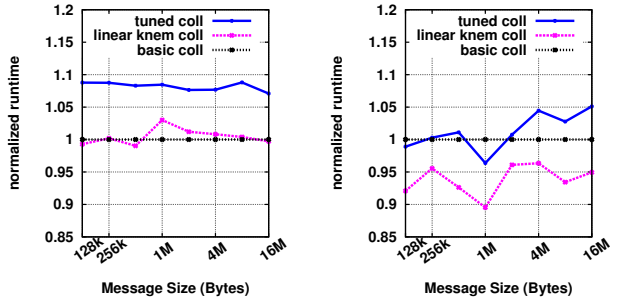
KNEM Gather and Scatter operations are very similar, except from the different copy direction: sender-writing for Gather and receiver-reading for Scatter. Consequently, those two algorithms exhibit very similar performance profiles and the Scatter results are not presented here. The KNEM Scatter speedup is nonetheless larger than for Gather because receiver-reading is faster than sender-writing in KNEM (see Figure 1). Compared with the basic Scatter implementation, the maximum improvement of KNEM Scatter is 70% on Zoot, 62% on Dancer, 72% on Saturn and 74% on IG.

E. Alltoall and Allgather

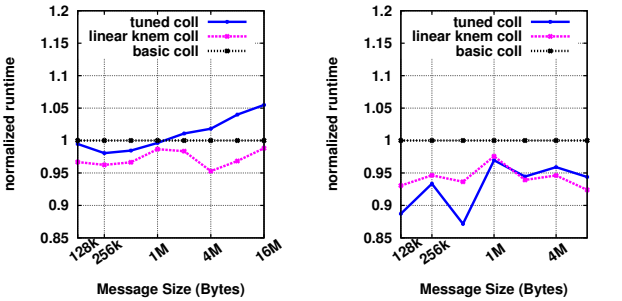
Figure 8 shows the performance comparison for the AlltoAll collective operation. On this non-rooted operation, there are less opportunity to gain from the extra parallelism when accessing the root buffer. As a consequence, the relative performance benefits are smaller when compared to the basic algorithm than for the one-to-all or all-to-one rooted operations presented in the previous paragraph. On Zoot, the KNEM collective module is a little worse than the basic collective module for message size from 1 MBytes to 8 MBytes. The reason is that too many simultaneous KNEM copies overwhelms the single SMP memory controller, which cannot sustain such a high rate of memory accesses. We expect that by tailoring the size of the working sets to control the degree of parallelism, it can be decreased to match architectures with few NUMA sockets. The performance profile between KNEM and Tuned collective modules are very similar, as they share the same topology and core mapping, and differ only by the fact that the Tuned AlltoAll algorithm benefits from pipelined point to point SendRecv communication as the underlying transport, while the KNEM AlltoAll algorithm does not feature yet pipeline, but only benefits from 1-copy transfer. On those three first machines, the Tuned module is suffering and often decreases performance. Contrasting is the KNEM module, which gets better performance in all cases on the small size NUMA nodes, and secures a maximum of 11% improvement on Dancer and 5% on Saturn.

On the larger NUMA machine (IG), because of the large number of cores, the available memory limits the maximum dataset to be exchanged by the AlltoAll operation. Therefore, the maximum message size is restricted to 8MB. Although the KNEM module increases performance on this platform when compared to the standard module for the entire range of message sizes, it dominates the Tuned module only for messages larger than 2MB. It appears the pipelining is extremely beneficial to the performance of intermediate size messages, and only for the largest message size the raw bandwidth advantage of the KNEM module is able to show. This indicates that, like for the Broadcast algorithm, the AlltoAll KNEM module should use at the same time pipelining and the single copy transport.

Finally, figure 9 presents the performance of the collective

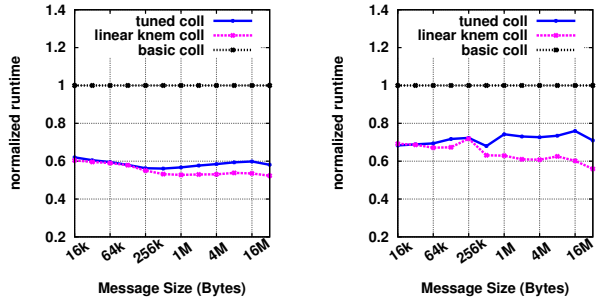


(a) on Zoot (b) on Dancer

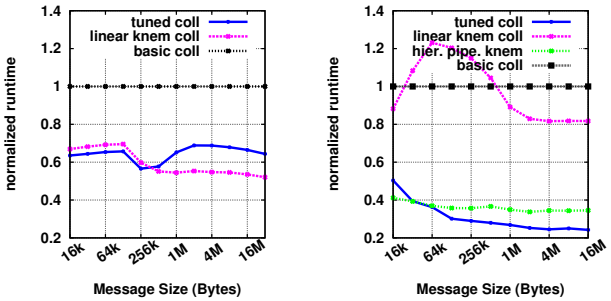


(c) on Saturn (d) on IG

Figure 8. Performance comparison of Alltoall Operations between Basic, Tuned, KNEM collectives. All results are normalized to the basic Alltoall runtime (lower is better).



(a) on Zoot (b) on Dancer



(c) on Saturn (d) on IG

Figure 9. Performance comparison of Allgather operations between the Basic, Tuned, KNEM collective modules. All curves are normalized to the basic Allgather runtime (lower is better).

modules on the AllGather operation. On the SMP machine (Zoot) and the medium size NUMA machine (Dancer, Saturn), the Tuned and KNEM modules perform similarly for small messages, and the KNEM module surpasses all other approaches for messages larger than 512KB. The performance improvement when compared with the basic module is substantial with more than 40% improvement for large messages. On the large NUMA node (IG), the linear KNEM algorithm is worse than tuned collectives and sometimes even worse than the basic algorithm. We came to the conclusion that the bigger the NUMA factor, the worse the linear KNEM Allgather behaves for all-to-all operations. The reason is that too much traffic is generated over the *slow* inter-socket bus. Indeed, the architecture topology is not considered in the linear KNEM algorithms which is based on the concatenation of the linear KNEM Gather and Broadcast algorithms. By using the hierarchical pipelined KNEM algorithm for the Broadcast, the performance gets close to the Tuned module, which does not use the simple concatenation of a gather and a broadcast but uses a more elaborate algorithm.

F. Application Performance

To evaluate the impact of the improvement due to using KNEM collective operations on real application performance, we use the ASP [19] application, a parallel implementation of the Floyd-Warshall algorithm used to solve the all-pairs-shortest-path problem. The main MPI collective operation used in this application is MPI_Bcast. We tested this application on two machines: Zoot and IG, the two extreme platforms regarding the degree of complexity of the core hierarchy. The problem is scaled to match the available memory; the matrix size is 16384^2 on Zoot and 32768^2 on IG (32bits integers). Matrices are distributed by rows across all the available cores. The MPI_Bcast operation is called 16384 times (64 KB message) on Zoot and 32768 times (128 KB message) on IG. The KNEM collective component uses the linear KNEM algorithm on Zoot and the hierarchical pipelined algorithm on IG. All tests use the same mapping between cores and processes.

	Zoot		IG	
Coll Algo.	Bcast time	App. time	Bcast time	App. time
tuned	405.7s	2891.2s	692.5s	6798.6s
SM	359.4s	2846.7s	256.9s	6344.8s
linear KNEM	26.84s	2508.43s	–	–
hier.pipe.KNEM	–	–	198s	6288.1s
Improvement	92.5%	12%	23%	0.9%

Table II
APPLICATION SPEEDUP FROM USING KNEM COLLECTIVES

Table II presents the application execution time of ASP when using different collective components. The improvement is relative difference between the best performing

collective module and our KNEM collective module. By using the KNEM collective modules, the application can see a significant improvement in the time it spends doing Broadcast operations. One can notice that the performance improvement of the Broadcast only on the SMP machine is even more pronounced than for the synthetic benchmark, because unlike the synthetic benchmark, the application does not systematically invalidate the cache before performing the operation. As a consequence of the shorter time spent in the collective operations, the overall application runtime is improved when compared to other optimized collective modules, no matter on SMP or large NUMA platforms.

VII. Conclusion and Future Work

The current trend in HPC is toward a large increase in the non-uniformity inside the node, both from the number of cores and the number of memory hierarchies. In this paper we have showed that an MPI implementation can successfully take advantage of these features to deliver more performance to the applications by exploiting the NUMA characteristics for point-to-point as well as collective communications. Well planned collective communication can automatically map themselves onto the architectural features of today's "fat" nodes, minimizing the number of memory copies, decreasing the pressure on the "remote" NUMA memory and carefully avoiding memory bottlenecks. With the help of specialized inter-process memory copy module (KNEM), root process can offload memory copies to non-root process in order to evenly distribute the memory accesses. In addition, taking advantage of cache reuse by pipelining the individual transfers can bust the performance even further. As a result collective algorithms with a complexity $O(T)$ instead of $O(NT)$ can drastically improve the performance of collective communication implementations in MPI. We believe that architecture-aware collectives will play a critical role in the future *fatter* nodes with more sockets/cores and deeper memory hierarchies.

Future Work: Continuing on the work presented in this paper, we will continue to improve the collective algorithms already implemented. In order to deal with the current issues for small sizes messages an integration with the shared memory collective module in Open MPI will be done in the near future. The resulting module will automatically select the best implementation based on the architecture, the process placement inside the node, the internal architecture of the node and the message size.

Moreover, we will use hwloc to optimize the hierarchical implementation and find a more automatic way to decide at which level of the topology, the hierarchical pipelined KNEM collectives will be selected. Hwloc project can provide, at runtime, architectural metadata such as NUMA factor and cache size, which can help KNEM collectives find suitable parameters. In addition, we plan to investigate the integration of upcoming additions to the MPI standard (expected in MPI 3.0) related to collective communications, such as non-blocking collectives.

References

[1] H. W. Meuer, E. Strohmaier, J. J. Dongarra, and H. D. Simon, "Top500 supercomputing sites (2010)," http://top500.org.

[2] MPI-Forum, "MPI: A message passing interface standard," 1994, http://www.mpi-forum.org/.

[3] D. Buntinas, G. Mercier, and W. Gropp, "Design and evaluation of Nemesis, a scalable, low-latency, message-passing communication subsystem," *Cluster Computing and the Grid, 2006. Sixth IEEE International Symposium on*, vol. 1, pp. 10–20, May 2006.

[4] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Computing*, vol. 22, no. 6, pp. 789–828, Sep. 1996.

[5] E. Gabriel, G. E. Fagg, G. Bosilca, T. Angskun, J. J. Dongarra, J. M. Squyres, V. Sahay, P. Kambadur, B. Barrett, A. Lumsdaine, R. H. Castain, D. J. Daniel, R. L. Graham, and T. S. Woodall, "Open MPI: Goals, concept, and design of a next generation MPI implementation," in *Proceedings, 11th European PVM/MPI Users' Group Meeting*, Budapest, Hungary, September 2004, pp. 97–104.

[6] L. Huse, "Collective communication on dedicated clusters of workstations," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, J. Dongarra, E. Luque, and T. Margalef, Eds. Springer-Verlag, 1999, vol. 1697, pp. 682–682, 10.1007/3-540-48158-3_58. [Online]. Available: http://dx.doi.org/10.1007/3-540-48158-3_58

[7] R. Graham and G. Shipman, "Mpi support for multi-core architectures: Optimized shared memory collectives," in *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, ser. Lecture Notes in Computer Science, A. Lastovetsky, T. Kechadi, and J. Dongarra, Eds. Springer Berlin / Heidelberg, 2008, vol. 5205, pp. 130–140. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-87475-1_21

[8] G. E. Fagg, G. Bosilca, J. Pješivac-Grbović, T. Angskun, and J. Dongarra, "Tuned: A flexible high performance collective communication component developed for open mpi," in *Proccedings of DAPSYS'06*. Innsbruck, Austria: Springer-Verlag, September 2006, pp. 65–72.

[9] P. Geoffray, L. Prylli, and B. Tourancheau, "BIP-SMP: high performance message passing over a cluster of commodity SMPs," in *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, Portland, OR, Nov. 1999.

[10] R. Brightwell, "Exploiting Direct Access Shared Memory for MPI On Multi-Core Processors," *International Journal of High Performance Computing Applications*, vol. 24, no. 1, pp. 69–77, 2010. [Online]. Available: http://hpc.sagepub.com/content/24/1/69.abstract

[11] H.-W. Jin, S. Sur, L. Chai, and D. K. Panda, "Lightweight kernel-level primitives for high-performance MPI intra-node communication over multi-core systems," *Cluster Computing, IEEE International Conference on*, vol. 0, pp. 446–451, 2007.

[12] D. Buntinas, B. Goglin, D. Goodell, G. Mercier, and S. Moreaud, "Cache-Efficient, Intranode Large-Message MPI Communication with MPICH2-Nemesis," in *Proceedings of the 38th International Conference on Parallel Processing (ICPP-2009)*. Vienna, Austria: IEEE Computer Society Press, Sep. 2009, pp. 462–469. [Online]. Available: http://hal.inria.fr/inria-00390064

[13] "KNEM: High-Performance Intra-Node MPI Communication," http://runtime.bordeaux.inria.fr/knem/.

[14] S. Moreaud, B. Goglin, D. Goodell, and R. Namyst, "Optimizing MPI Communication within large Multicore nodes with Kernel assistance," in *CAC 2010: The 10th Workshop on Communication Architecture for Clusters, held in conjunction with IPDPS 2010*. Atlanta, GA: IEEE Computer Society Press, Apr. 2010.

[15] "Netpipe 3.7.2," http://bitspjoule.org/NetPIPE/.

[16] F. Broquedis, J. Clet Ortega, S. Moreaud, N. Furmento, B. Goglin, G. Mercier, S. Thibault, and R. Namyst, "hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications," in *The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, 02 2010. [Online]. Available: http://hal.inria.fr/inria-00429889/en/

[17] M. Chaarawi, J. M.Squyres, E. Gabriel, and S. Feki, "A tool for optimizing runtime parameters of Open MPI," in *Proceedings, 15th European PVM/MPI Users' Group Meeting*, ser. Lecture Notes in Computer Science, no. 5205. Springer Verlag, 2008, pp. 210–217.

[18] "Intel MPI benchmarks 3.2," http://software.intel.com/en-us/articles/intel-mpi-benchmarks/.

[19] A. Plaat, H. E. Bal, R. F. H. Hofman, and T. Kielmann, "Sensitivity of parallel applications to large differences in bandwidth and latency in two-layer interconnects," *Future Generation Computer Systems*, vol. 17, no. 6, pp. 769 – 782, 2001. [Online]. Available: http://www.sciencedirect.com/science/article/B6V06-42M1FNP-D/2/e525ec950024b8b0667d16f5b50f58b4