SANDIA REPORT

SAND2010-6262

Unlimited Release

Printed September, 2010

LDRD Final Report: Managing Shared Memory Data Distribution in Hybrid HPC Applications

Kevin T. Pedretti, Alexander M. Merritt

Prepared by

Sandia National Laboratories

Albuquerque, New Mexico 87185 and Livermore, California 94550

Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

Approved for public release; further dissemination unlimited.

Sandia National Laboratories

Issued by Sandia National Laboratories, operated for the United States Department of Energy by Sandia Corporation.

NOTICE: This report was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government, nor any agency thereof, nor any of their employees, nor any of their contractors, subcontractors, or their employees, make any warranty, express or implied, or assume any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represent that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government, any agency thereof, or any of their contractors or subcontractors. The views and opinions expressed herein do not necessarily state or reflect those of the United States Government, any agency thereof, or any of their contractors.

Printed in the United States of America. This report has been reproduced directly from the best available copy.

Available to DOE and DOE contractors from
U.S. Department of Energy
Office of Scientific and Technical Information
P.O. Box 62
Oak Ridge, TN 37831

Telephone: (865) 576-8401
Facsimile: (865) 576-5728
E-Mail: reports@adonis.osti.gov
Online ordering: <http://www.osti.gov/bridge>

Available to the public from
U.S. Department of Commerce
National Technical Information Service
5285 Port Royal Rd
Springfield, VA 22161

Telephone: (800) 553-6847
Facsimile: (703) 605-6900
E-Mail: orders@ntis.fedworld.gov
Online ordering: <http://www.ntis.gov/help/ordermethods.asp?loc=7-4-0#online>

LDRD Final Report: Managing Shared Memory Data Distribution in Hybrid HPC Applications

Kevin T. Pedretti (PI)
Scalable System Software Department (1423)
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185-1319
ktpedre@sandia.gov

Alexander M. Merritt
Graduate Student, 2010 CSRI Summer Student
Georgia Institute of Technology
Atlanta, GA 30332
merritt.alex@gatech.edu

Abstract

MPI is the dominant programming model for distributed memory parallel computers, and is often used as the intra-node programming model on multi-core compute nodes. However, application developers are increasingly turning to hybrid models that use threading within a node and MPI between nodes. In contrast to MPI, most current threaded models do not require application developers to deal explicitly with data locality. With increasing core counts and deeper NUMA hierarchies seen in the upcoming LANL/SNL “Cielo” capability supercomputer, data distribution poses an upper boundary on intra-node scalability within threaded applications. Data locality therefore has to be identified at runtime using static memory allocation policies such as first-touch or next-touch, or specified by the application user at launch time. We evaluate several existing techniques for managing data distribution

using micro-benchmarks on an AMD “Magny-Cours” system with 24 cores among 4 NUMA domains and argue for the adoption of a dynamic runtime system implemented at the kernel level, employing a novel page table replication scheme to gather per-NUMA domain memory access traces.

Acknowledgment

We wish to acknowledge Michael Heroux (SNL, 1416) and Michael Wolf (SNL, 1416) for their valuable discussions and for providing the Mantevo benchmark suite, which was used in our evaluations.

We also thank Douglas Doerfler (SNL, 1422) and James Laros (SNL, 1422) for providing information on the Cielo architecture, and for their advice throughout the project.

We thank Brice Goglin (INRIA, France) for providing us with the “migrate on next touch” patch that he and others developed for the Linux kernel, and helping us to get it working on our test systems.

We thank Rolf Riesen for contributing the L^AT_EX template for this report.

Contents

Summary	9
1 Techniques for Managing Data Distribution in NUMA Systems	11
1.1 Introduction	12
1.1.1 Parallel Programming Models	13
1.1.2 Related Work	14
1.2 Evaluation	14
1.2.1 Background	14
1.2.2 Benchmarks	15
Evaluation System	16
STREAM	16
HPCCG	16
1.2.3 Techniques	18
First-Touch	18
Memory Interleaving	18
Next-Touch	18
1.2.4 Results	19
1.3 Future Work	21
1.4 Conclusion	23
References	24

List of Figures

1.1	Two AMD Opteron 6174 processors.....	12
1.2	Virtual memory mapping.	14
1.3	UMA vs NUMA: memory latencies relative to domain 0.	15
1.4	STREAM micro-kernels.	16
1.5	STREAM per-thread data locality (entire execution).....	17
1.6	HPCCG per-thread data locality.	17
1.7	Two levels and granularities of memory interleaving.	19
1.8	Performance of STREAM applying current data distribution techniques.	20
1.9	Performance of HPCCG applying current data distribution techniques.	21
1.10	Proposed dynamic runtime system for page migration within the kernel.	22

Summary

This report summarizes the R&D activities of the FY10 late-start LDRD project “Managing Shared Memory Data Distribution in Hybrid HPC Applications”, which was funded at a level of approximately 0.15 FTE for the year. The project was carried out by Kevin Pedretti (1423) and Alexander Merritt, a 2010 summer student from Georgia Tech.

The primary objectives of this project were to examine existing intra-node data distribution approaches on Non-Uniform Memory Access (NUMA) platforms and to develop new approaches that address their observed weaknesses. In particular, we were interested in pursuing a dynamic runtime approach that observes thread–data affinity at runtime and migrates memory pages accordingly. The report in Chapter 1 contains our analysis of existing mechanisms on an AMD “Magny-Cours” 24-core, 4 NUMA node test system that is similar to the upcoming LANL/SNL Cielo supercomputer. The results provide valuable insights into the issues that can arise when running shared-memory applications on such a system. The results also provide motivation for a dynamic runtime approach, and we have begun implementing one. This work was accepted as a poster at the ACM/IEEE 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC’10), and we plan to expand it to a full paper in the future.

Chapter 1

Techniques for Managing Data Distribution in NUMA Systems

Authors: Alexander M. Merritt (Georgia Institute of Technology) and Kevin T. Pedretti (Sandia)

The content of this chapter was originally published in the 2010 CSRI (Computer Science Research Institute) Summer Proceedings. Every CSRI summer student is required to write a report for the proceedings, and each report is peer-reviewed by one or more CSRI staff members. The proceedings are publicly available on the CSRI web page, <http://www.cs.sandia.gov/CSRI/>. The results in this paper will be included in a poster at the ACM/IEEE 2010 International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10), and we plan to expand it to a full paper in the future.

Abstract

MPI is the dominant programming model for distributed memory parallel computers, and is often used as the intra-node programming model on multi-core compute nodes. However, application developers are increasingly turning to hybrid models that use threading within a node and MPI between nodes. In contrast to MPI, most current threaded models do not require application developers to deal explicitly with data locality. With increasing core counts and deeper NUMA hierarchies seen in the upcoming LANL/SNL “Cielo” capability supercomputer, data distribution poses an upper boundary on intra-node scalability within threaded applications. Data locality therefore has to be identified at runtime using static memory allocation policies such as first-touch or next-touch, or specified by the application user at launch time. We evaluate several existing techniques for managing data distribution using micro-benchmarks on an AMD “Magny-Cours” system with 24 cores among 4 NUMA domains and argue for the adoption of a dynamic runtime system implemented at the kernel level, employing a novel page table replication scheme to gather per-NUMA domain memory access traces.

1.1 Introduction

Scalability plays an important role in high-performance computing. Large-scale simulations of nuclear reactions, nuclear decay, climate modeling, fluid dynamics and combustion all require greater precision and reduced time-to-solution to achieve accurate and timely predictions. In order to achieve this goal hardware and software must scale well together. With the onset of exascale computing current parallel programming models approach their limits in scalability as supercomputing hardware transitions from distributed-memory single-core processor architectures to hybrids of distributed-memory and shared-memory, heterogeneous and accelerator-supported systems.

Our work is focused on the architecture of the upcoming LANL/SNL “Cielo” capability supercomputer, which consists of AMD Opteron “Magny-Cours” non-uniform memory access (NUMA) processors. In contrast to previous SNL systems such as ASCI Red and Red Storm, Cielo’s hardware exhibits greater complexity within the node and processor itself, adding more cores and greater variation in memory access latencies. Non-local memory accesses comprise multiple levels of non-uniformity, incurring different penalties depending on the path traveled. Figure 1.1 illustrates our 24-core 4 NUMA domain dual-socket Magny-Cours experimental system. Off-chip diagonal HyperTransport links exhibit the largest latencies and are 8-bit wide, which is half as wide as all other processor interconnects in the system. Each Cielo compute node is similar to our test system except that Cielo uses 8-core processors running at 2.2 GHz on a 64-bit architecture. The 2.2 GHz on a 64-bit architecture threaded processor degradation factor is 1.6.

Figure 1.1: Two AMD Opteron 6174 processors.

In this paper we discuss the hybridization of parallel programming models and how they perform in light of this new architecture, focusing on the evaluation of current intra-node approaches to data distribution that attempt to minimize the influence of NUMA latencies

in multithreaded applications. We demonstrate that these approaches are not adequate to address losses in performance and additionally require developers to modify their application code. We argue for the use of a dynamic runtime system within the operating system to identify data access patterns of an application and use this information to redistribute data, avoiding code changes and user intervention current approaches require.

1.1.1 Parallel Programming Models

Programming languages have evolved along with the hardware on which they run. MPI [1] is a library-based message-passing extension to existing languages, such as C and Fortran. This model is a good fit for the distributed-memory architectures of supercomputers. Parallel units of execution called *ranks* exist in their own address space, each occupying one CPU core per compute node. In this model, communication and data sharing are explicit and must be known to the application developer¹ enabling him or her to finely tune algorithms for minimizing communication overheads. This advantage of MPI is, however, also its drawback: scaling applications to the extreme of exascale computing with hundreds of millions of processors [12] detracts from programmer productivity in addition to making debugging difficult. More limitations on scalability arise with the introduction of modern supercomputing hardware: higher core counts—more ranks—within a node increase memory consumed by global state replication and increase communication, causing contention on processor interconnects. Furthermore, a recent research study [14] compared the performances of message-passing and threaded programming implementations of sparse matrix-vector multiplication code, demonstrating that threaded models perform better on SMP hardware. These are all motivations for relaxing the trend of running “MPI everywhere”.

Thread-based parallel programming models such as OpenMP [2] and Pthreads operate in a single address space, avoiding the overheads of MPI. One address space allows global access to shared state and communication to be achieved through shared variables. Compared to other models, OpenMP is an automated parallelization tool designed to move the burden of explicit parallelization from the programmer to the compiler using simple `#pragma` directives in code. While combining threaded and message-passing programming models for intra-node and inter-node parallelization may show improvements [11], support for threading models on SMP NUMA systems is still immature. OpenMP was designed assuming homogenous processing and uniform memory access architectures (UMA), but this is no longer true as commodity processor technologies are becoming increasingly NUMA. Without data replication, distribution is proving to be an inhibitor on software scalability with the advancement of parallel hardware present in the Cielo supercomputer.

¹*Programmer* and *developer* are used interchangeably in this paper.

1.1.2 Related Work

Current approaches take advantage of the virtual memory subsystem that the operating system kernel provides [5], use hardware counters provided by the CPU [13] as well as combine techniques at the user-level [3, 4].

User-level support for data distribution [3] combines knowledge from both a NUMA-aware memory manager and custom thread scheduler, provided as an extension to OpenMP to best minimize remote memory references. As with our work, this research advocates a *dynamic* runtime system that can adapt to changes in thread-data affinities throughout an application’s execution. In contrast to our proposed technique for managing data distribution, this work still requires applications to use a supplementary API where ours aims for an application-independent implementation.

1.2 Evaluation

In this section we discuss recent research in literature and their effects on relevant micro-benchmarks at Sandia as our motivation for runtime analysis.

Figure 1.2: Virtual memory mapping.

Hardware and operating system support for virtual memory allow for flexibility when designing models for data distribution. Each process is allowed to believe it has complete control over the entire system—full access to all of memory and the processor. In figure 1.2 we see the state of two processes at a given point in time. Memory (both virtual and physical) are divided into equal-sized regions called *pages*. When a process performs a memory operation such as a load or a store on data not present in memory, the hardware traps the faulting process and automatically transfers control to the operating system. The

operating system then loads an entire page of data from an external source into an empty page in memory and establishes a translation. Each translation’s state is represented by a structure in memory maintained by the kernel—called a *page table*—containing among other information, protection and access bits. Applications run as a combination of processes and threads, unaware of this mechanism that controls the distribution of its data in real hardware.

Within a NUMA system, all processors and memory regions are divided into *domains*. A domain is defined as a set of processors or cores and a region of memory between which is the lowest latency. Figure 1.3 depicts normalized latency costs of accessing all regions of memory for domain 0 in both an UMA and four-domain NUMA environment. Should a process be scheduled to execute in domain 0, domain 1 would incur the highest latency.

Figure 1.3: UMA vs NUMA: memory latencies relative to domain 0.

In the following subsections we review micro-benchmarks and their memory access patterns, current techniques examined, finally concluding with a discussion on the impact of these techniques as motivation for a dynamic runtime memory migration system.

1.2.2 Benchmarks

In this subsection we discuss two micro-benchmarks used to evaluate current approaches to data distribution, describe them in terms of *phases* and how current approaches affect these phases. Both micro-benchmarks model common characteristics seen in scientific code at Sandia and are used as the foundation for further study. Benchmarks designed around MPI rewritten to use OpenMP have all explicit data movement removed.

We define a phase in a multithreaded application to mean an interval of time within which data access patterns remain deterministic or constant for each thread, within some threshold. A change in the number of threads also constitutes a phase change as data access characteristics must either be established for newly-spawned threads, or forgotten with fewer threads.

The use of a dynamic binary instrumentation tool called Pin [9] allows us to capture all instructions of an application that access memory. With this information we are able to visualize the data access patterns for both micro-benchmarks.

Evaluation System

Our evaluation system is a single shared-memory system with two AMD Opteron 6174 “Magny-Cours” processors. Each is a multichip module (MCM) containing two processor dies each with six symmetric cores running at 2.2 GHz, two memory controllers rated at 10.6 GiBps and a local subset of system memory. All four dies are connected with 10.4 GiBps HyperTransport (HT) links in each direction, forming a complete graph. All HT links are 16-bit wide except for two 8-bit wide diagonal crosslinks connecting domains 0-3 and 1-2. Figure 1.1 illustrates this design. Four memory and processor domains are available, each with 8 GiB of memory. Our study of this processor is significant as it forms the basis of the upcoming Los Alamos/Sandia National Labs “Cielo” capability supercomputer.

Results from current approaches were obtained on this system running RHEL 5.4 with a vanilla 2.6.27 kernel patched with support for the next-touch [5] memory policy.

STREAM

STREAM [10] is a memory bandwidth micro-benchmark parallelized with OpenMP. Each thread executes multiple kernels over its subset of data within three global arrays. Two kernels are represented in our data, “copy” and “triad” illustrated in figure 1.4. STREAM was configured to use 333 MiB-sized arrays to minimize caching effects (our test system has 20 MiB of effective last-level cache ²). Additionally, array elements are only accessed within parallel regions to create high affinities between threads and their data.

Kernel	Code
<i>Copy</i>	<code>c[j] = a[j]</code>
<i>Triad</i>	<code>a[j] = b[j] + scalar * c[j]</code>

Figure 1.4: STREAM micro-kernels.

Each of the three arrays are visible in figure 1.5 clearly indicating the regular access patterns of all threads. A fourth line towards the end of its virtual memory space represents the result vector storing timing measurements. STREAM exhibits this behavior throughout its entire execution.

HPCCG

HPCCG [6] is a sparse-matrix vector multiply application with parallel implementations in both MPI and OpenMP. HPCCG allocates multiple arrays of data before entering repeated iterations of its parallel sections where, like STREAM, each thread performs work on a subset

²While the system has 24 MiB of L3 cache, the HT Assist optimization uses 4 MiB of L3 to store directory information.

Figure 1.5: STREAM per-thread data locality (entire execution).

of data. Two phases have been identified in this micro-benchmark, both visualized in figure 1.6 for the OpenMP implementation.

(a) Phase 1.

(b) Phase 2.

Figure 1.6: HPCCG per-thread data locality.

Figure 1.6a depicts the master thread allocating and initializing all data. Figure 1.6b depicts the second phase of HPCCG's execution, wherein multiple parallel sections perform work on subsets of the overall data. The majority of work performed by this micro-benchmark is within this phase. Affinities between threads and data are strong but different in the two phases.

1.2.3 Techniques

In this subsection we discuss current approaches to data distribution at or below the operating system level and investigate their advantages and disadvantages as they apply to application phases seen in STREAM and HPCCG.

First-Touch

This is the default policy in the Linux kernel. It is not a solution to the NUMA problem per se, but rather a means to enable memory allocated by applications to be local to the domain in which they are executing. The Linux kernel memory allocator maintains a pool of memory for each domain in the system. This policy has no means to assist applications exhibiting strong phase changes, as data is never moved once allocated. Applications must be designed accordingly to minimize remote memory accesses, requiring that all data be touched first by the thread using it the most.

Memory Interleaving

Memory interleaving allows for an even distribution of memory over NUMA domains at the page and cache line granularity. Two configurations were available to us for experimentation and were simple to configure; these are presented in figure 1.7. Figure 1.7a illustrates operating system control over an application's virtual memory pages. The kernel memory allocator maintains pools of memory for each domain, cycling among them when creating translations from the virtual memory space of an application. Accessing virtual memory linearly physically iterates over all available NUMA domains at a page granularity. Figure 1.7b illustrates a second method of interleaving. Here the memory controllers are configured by the BIOS to modify its mapping of physical addresses to machine memory, interleaving them among the NUMA domains. In contrast, this method operates at the granularity of a cache line, is transparent to the operating system and allows for a finer distribution of memory among the domains (interleaving the kernel address space as well as all application address spaces). Both methods distribute memory such that the chance of accessing either a page or cache line locally is $\frac{1}{D}$ for D domains.

Page interleaving support is provided by the `numactl` command line tool in Linux. It allows for various NUMA-related operations on processes, such as memory and domain execution pinning, CPU core pinning and memory interleaving.

Next-Touch

Next-touch is a memory policy implemented in kernel space. In this scheme, an application flags regions of memory that should be migrated to the NUMA domain of the next

Figure 1.7: Two levels and granularities of memory interleaving.

thread that accesses the region. By manipulating protection bits in the page table we force the hardware to intercept memory accesses, migrating their pages before resuming process execution. Recent work [5] provides this implementation as a patch for the 2.6.27 Linux kernel. The patch adds additional flags to the `madvise()` system call enabling user-space applications to inform the kernel’s memory subsystem to modify the appropriate protection bits on a given range of pages. On subsequent touches the hardware traps the executing process and migrates pages to whichever domain the process is executing on.

Static code analysis enables the programmer to identify when page migrations are needed. One difficulty with this approach is ensuring that no thread other than the one intended touches the pages it will most heavily use. This assumes the appropriate place to call `madvise()` can be determined through static analysis. Furthermore, the data access pattern among threads may change throughout an application’s lifetime. For this approach to be effective each application phase must be identified and a call to `madvise()` inserted.

Support for this feature exists in the Oracle Solaris 9 operating system [13], but currently has not been accepted into the mainstream Linux kernel codebase.

1.2.4 Results

In this subsection we present and compare performance results for the next-touch, first-touch and memory interleaving strategies with respect to execution phases seen in STREAM and HPCCG. Results are shown in figures 1.9 and 1.8. Points plotted in each of the three graphs comprise an average of 20 executions with error bars illustrating one standard deviation. We used two strategies for pinning threads, round-robin (“Pin RR”) and ascending (“Pin Asc”), both tied to the Linux-logical core IDs.

Figure 1.8: Performance of STREAM applying current data distribution techniques.

STREAM performance data was collected from a combination of operating system thread scheduling and manual thread pinning in addition to the first-touch and page interleave strategies. Red curves in figures 1.8a and 1.8b show the performance of STREAM with no modifications and no runtime policies. Variability is high due to the non-deterministic behavior of the 2.6.27 Linux scheduler and its inability to identify affinities between data and threads. It may repeatedly schedule multiple threads for execution on the same domain before attempting to reduce oversubscription. This behavior causes threads to access their data from various domains, causing scattered first-touch allocations to occur. Should the scheduler know to not relocate threads, first-touch would show the best performance. Forcing thread pinnings reproduces this behavior and indeed shows the best performance, represented by black curves in both plots.³ Interleaving pages reduces performance considerably, as the probability of accessing local data is reduced to 25% from near 100% when compared with thread pinning and first-touch.⁴ Pinning threads in ascending order by Linux-logical core IDs shows a plateau (gray curve) as the first twelve core assignments alternate between domains 0 and 3, the latter twelve between domains 1 and 2. This demonstrates that assumptions cannot be made regarding the logical-to-physical mapping of cores by the operating system. Next-touch results were not included as first-touch and thread pinning give the same result. Curves in figure 1.8b are more linear due to a larger percentage of work coming from computational overhead. Overall trends are however still visible.

HPCCG performance data was collected from the same policy combinations used for STREAM, with the additional cache line interleaving technique and results from the MPI implementation. The red curve in figure 1.9 is the same policy as the red curve in figure 1.8, with no code modifications or runtime policies. This behavior is attributed to the changes

³We do not yet know the cause of the saw-tooth shape in the curve.

⁴A bug was discovered in the PGI OpenMP runtime library, preventing thread pinning while interleaving pages. Unfortunately this prevents accurate comparisons as we cannot eliminate the scheduler's nondeterministic behavior. This bug was reported to PGI and promptly fixed; it will be available in the next release of the PGI compiler suite.

Figure 1.9: Performance of HPCCG applying current data distribution techniques.

in the data access patterns mentioned in section 1.2.2: the existence of phase 1 forces the first-touch policy to allocate all memory pages on single memory domain. Transitioning to phase 2 shows an increase in threads, all of which are relocated by the scheduler to CPU cores in other domains, to minimize oversubscription. In doing so memory references for 75% of all threads become almost entirely non-local. In contrast, an MPI implementation scales much better due to data duplication, yet performance tapers off with many threads showing large variations most likely attributed again to the nondeterministic behavior of the Linux scheduler. Without data distribution in a threaded environment first-touch prevents applications with phases similar to HPCCG from scaling on NUMA hardware.

Having identified both phases of HPCCG, we were able to effectively use next-touch by placing appropriate calls to `madvise()` at the end of phase 1. Combined with thread pinning this method showed the best overall performance as memory pages were migrated to domains in which the accessing threads were executing, enabling nearly all memory references to be local. Page and cache line interleaving also improve performance as expected, with cache line interleaving showing slightly higher performance due to the smaller granularity.

1.3 Future Work

An application’s data access patterns change throughout execution. Approaches discussed in prior sections function in a one-shot manner, continuously apply the same rule or require user interaction. We therefore propose a dynamic runtime system for monitoring an application’s memory access patterns from within the kernel. Active monitoring allows the operating system to observe affinities between an application’s threads and pages in memory, migrating pages to reduce remote memory references.

An operating system typically allocates one page table per process. By using one page

Figure 1.10: Proposed dynamic runtime system for page migration within the kernel.

table per domain per process we will be able to capture where accesses originate and what regions of memory they reference, as seen in figure 1.10a. The appropriate page table is installed on a context switch, and access bits within page table entries updated by the hardware will be read to monitor an application’s data access patterns. A kernel thread will periodically scan these entries to observe page access patterns, identify frequent accesses of non-local memory and migrate pages accordingly. Frequent scanning will be required as only one access bit is available per page, potentially increasing profiling overhead. To prove itself advantageous, our approach must ensure the savings in execution time of the application are greater than the combined cost of data migration and active profiling—figure 1.10b. We argue that this approach will allow for a more flexible and customizable solution to the problem of data distribution on NUMA hardware.

Overheads introduced by this mechanism can be reduced if implemented in a light-weight kernel such as Kitten [7]. Kitten creates a complete linear mapping of all virtual memory addresses on process creation, in other words, pre-populating all page tables. Knowing the locations of all last-level page table entries can reduce the execution overhead introduced from frequent access bit scanning by avoiding full page table traversals. Kitten furthermore supports the use of large pages to map virtual memory. Enabling this will reduce the height of each page table and lower the number of page table entries needed. Using our proposed approach, each process will consume four times the amount of memory needed to store their address translations; large pages will reduce this footprint.

Our approach is a mechanism that will require various policies to show any benefit. Varying the profiling frequency and defining intervals for page migration and the meaning of “heavy” page access will have to be adjusted to determine which combinations give the best results across a range of application phases. Research in the early nineties examined this [8], but on different hardware and with a different application domain. It was determined that no single policy proved beneficial across all kernels in their benchmark suite. Our goal is to re-examine this on modern hardware across a more appropriate set of kernels.

Future support in hardware may include widening the access bit field in each page table entry and modifying the processor to treat this field as a saturating counter instead of a bit flip. While beyond the scope of this research, it would reduce the execution overhead from profiling, by lowering the frequency at which page table entries need to be scanned.

1.4 Conclusion

In this paper we examined several existing techniques for managing data distribution in a multicore NUMA environment, the basis for the upcoming “Cielo” capability supercomputer. As scientific applications are becoming increasingly hybrid, incorporating threaded programming models within nodes, support for data distribution becomes a limit on intra-node scalability. We demonstrated the static nature of current techniques, requiring time-consuming code modifications or user intervention. With tools developed to visualize application data access patterns we are better able to identify phase changes in thread-data affinities, enabling a more complete understanding of our application domain. Combined with an evaluation of our proposed kernel-level dynamic runtime technique we demonstrate the need for an invisible and adaptive data distribution model, empowering systems software to continue to scale as we approach exascale computing.

References

- [1] Mpi. <http://www.mpi-forum.org>.
- [2] Openmp. <http://www.openmp.org>.
- [3] F. Broquedis, O. Aumage, B. Goglin, S. Thibault, P.-A. Wacrenier, and R. Namyst. Structuring the execution of openmp applications for multicore architectures. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1 –10, 19-23 2010.
- [4] François Broquedis, Nathalie Furmento, Brice Goglin, Raymond Namyst, and Pierre-André Wacrenier. Dynamic Task and Data Placement over NUMA Architectures: an OpenMP Runtime Perspective. In *International Workshop on OpenMP (IWOMP)*, Dresden Allemagne, 2009.
- [5] B. Goglin and N. Furmento. Enabling high-performance memory migration for multi-threaded applications on linux. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1 –9, 23-29 2009.
- [6] Michael Heroux. Hpccg. <http://bec.syr.edu/hpccg.html>.
- [7] J. Lange, K. Pedretti, T. Hudson, P. Dinda, Zheng Cui, Lei Xia, P. Bridges, A. Gocke, S. Jaconette, M. Levenhagen, and R. Brightwell. Palacios and kitten: New high performance operating systems for scalable virtualized and native supercomputing. pages 1 –12, apr. 2010.
- [8] Richard P. Larowe, Jr. and Carla Schlatter Ellis. Experimental comparison of memory management policies for numa multiprocessors. *ACM Trans. Comput. Syst.*, 9(4):319–363, 1991.
- [9] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 190–200, New York, NY, USA, 2005. ACM.
- [10] John D. McCalpin. Memory bandwidth and machine balance in current high performance computers. *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25, December 1995.
- [11] Lorna Smith and Mark Bull. Development of mixed mode mpi / openmp applications. *Sci. Program.*, 9(2,3):83–98, 2001.

- [12] Patrick Thibodeau. Scientists, it community await exascale computers, December 2009.
http://www.computerworld.com/s/article/345800/Scientists_IT_Community_Await_Exascale.0
- [13] Mustafa M. Tikir and Jeffrey K. Hollingsworth. Using hardware counters to automatically improve memory performance. In *SC '04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*, page 46, Washington, DC, USA, 2004. IEEE Computer Society.
- [14] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. In *SC '07: Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, pages 1–12, New York, NY, USA, 2007. ACM.

DISTRIBUTION:

MS ,
,
1 MS 0899 Technical Library, 9536 (electronic copy)