Analysis of Inter-Chip Communication Patterns on Multi-Core Distributed Shared-Memory Computers

Manfred Mücke, Wilfried N. Gansterer
University of Vienna
Research Lab Computational Technologies and Applications

Abstract

Multi-core multi-socket distributed shared-memory computers (DSM computers, for short) have become an important node architecture in scientific computing as they provide substantial computational capacity with relatively low space and power requirements. Compared to conventional computer networks, inter-chip networks used in DSM computers feature higher bandwidth, lower latency and tighter integration with the CPU.

The inter-chip network is a shared resource among the user application and many other services, which can lead to considerable variation of execution times of identical communication tasks.

In this work, we explore traffic patterns resulting from MPI collective communication primitives and investigate the question whether inter-chip link load is a reliable indicator and predictor for the execution time of collective communication primitives on a DSM computer. Our experiments on a Sun Fire X4600 M2 DSM computer with 32 cores (eight quad-core CPUs) indicate that specific single link loads are positively correlated with the execution time of MPI_ALLREDUCE. Observing patterns over multiple links allows refinement of the single-link observation.

1. Motivation

Multi-core multi-socket distributed shared-memory (DSM) computers are a viable option to consolidate cluster infrastructure and to improve communication performance by reducing inter-node communication. One can think of a DSM computer as a small cluster with very high bandwith and low latency point-to-point interconnect.

In a cluster environment (many interconnected independent nodes), the overall performance is usually limited by the inter-node communication which is typically slow compared to local computation. Yet, recent work has shown that an unexpectedly high percentage of communication time is spent *within* multi-core nodes [3]. As a result, the node-internal communication performance – although faster than inter-node communication – is becoming more important for distributed applications' performance in a conventional cluster setting.

With current DSM computers integrating up to 48 cores in a single chassis, there is an increasing set of distributed applications which can run efficiently on a single DSM computer, thereby removing the need for a conventional cluster environment. To improve such an application's performance usually requires optimising the intra-node communication performance.

The situation for distributed applications executed on a single DSM computer changes considerably compared to a cluster environment as dedicated communication times among CPUs and memory access times become potentially identical. Additionally – and also in contrast to clusters – both computation (via memory access and/or cache coherency) and communication access the inter-chip communication network, which makes it a shared resource. Consequently, execution times of communication and computation can no longer be considered independent of each other but potentially heavily influence each other.

MPI collective communication functions [5] are powerful communication primitives whose optimisation is key to maximising performance of many parallel scientific computing applications. Collective communication can be seen as a parametriseable collection of point-to-point communications with only a few defined synchronisation points and the specific schedule being left to the implementation. We believe that a static schedule (or a set of several static schedules) is inadequate to efficiently exploit the available bandwidth in a contemporary multi-core DSM computer. Dynamic schedules might guarantee a more consistent performance over a wide range of network traffic scenarios. Dynamic schedules require, however, a cheap, yet reliable performance predictor, which is the motivation of our work.

MPI blocking communication provides function calls which return only when communication has finished

(i.e., communication and computation is mutually exclusive for a single MPI process). There is an ongoing discussion on integrating non-blocking collective communication primitives into future versions of MPI. Non-blocking communication allows for overlapping communication and computation. However, when communication and computation overlap on DSM computers, usage patterns of shared resources become highly dynamic. In the worst case, this could lead to lower performance compared to blocking communication. Non-blocking collective communication implementations can, however, devise an efficient dynamic communication strategy, subject to available performance indicators. Therefore delivering the performance promises of non-blocking collective communication on DSM computers requires reliable communication performance predictors.

While DSM computers have existed for a long time, only recent developments have made them an almost ubiquitous computing platform. First, AMD integrated high-bandwidth low-latency inter-chip network interfaces (HyperTransport) into its mainstream server CPU family (Opteron), thereby removing the need for dedicated inter-chip communication circuits and simplifying the design of multi-socket computers considerably. Second, the integration of memory interfaces into CPUs enabled low-latency access to memory via inter-chip network thereby allowing very-low-latency non-uniform memory access (NUMA) computers. Third, multi-core CPUs have mitigated the scaling limitations of integrated inter-chip networks (for example AMD Opterons only support up to eight-socket configurations) by providing more cores per socket. Currently, systems with 48 cores (eight quad-core CPUs) are available. Finally, the evolution of communication technology has led to inter-chip point-to-point interface specifications matching typical internal bandwidths of CPUs (HyperTransport 3.1: 16 bit@3.2 GHz, max. 16 bit bi-directional bandwidth of 25.6 GB/s), leading to a communication performance which is *at par* with computation performance.

2. Problem Formulation

Inter-chip networks of contemporary DSM computers are typically used by multiple system services, they are a shared resource. Most prominently, remote memory access, the cache coherency protocol and system I/O usually use the same inter-chip network as dedicated communication between CPUs. Consequently, identical user-triggered communication can meet very different resource usage scenarios leading to variations in execution times.

Dynamic communication schedules can mitigate this effect. To choose the most efficient schedule for a communication operation at any given time, a performance model is required, taking the load on all relevant shared resources into account. The fastest schedule is then derived from the model by extrapolating current usage on all relevant shared resources.

Our aim in this paper is to identify the relevant observables necessary to implement dynamic schedules for MPI collective communication functions on DSM computers at the lowest possible cost (i.e., observation should be feasible on standard hardware and should cause only little overhead).

We hypothesise that on DSM computers the respective bandwidth available on each link of the inter-chip network is the single most relevant parameter influencing the execution time of a collective communication function. If this hypothesis can be verified, observing the inter-chip network bandwidth would provide sufficient information for optimizing dynamic communication schedules. Contemporary CPUs feature hardware performance counters which provide detailed information on the link traffic with high accuracy and at low cost, therefore on existing CPU architectures, monitoring inter-chip network bandwidth is possible for user applications at basically no extra cost.

3. Related Work

Scogland et al. [12] describe in a more general setting than our MPI-centric one that although multi-core hardware is mostly symmetric (i.e. cores have equivalent raw performance and bandwidth available), resulting workload per core is highly asymmetric due to the interaction of communication and computation.

Kayi et al. [7] report performance figures for large-scale simulations on a hybrid cluster consisting of nodes with 2 sockets (4 cores) and 8 sockets (16 cores), respectively. They found that application performance was *poorer* on the more powerful nodes. Only when applications employed some kind of node-internal load balancing, improvements could be observed. Core binding was found to improve the situation, too.

Porterfield et al. [11] conducted a detailed performance study of a variety of AMD quad-core multi-socket systems over a set of memory benchmarks. They found that performance models characterising memory by maximum bandwidth and average latency parameters are not sufficient to model the deep memory hierarchies found in modern ccNUMA architectures. Specifically, they found performance variability for memory-bound benchmarks to be a serious obstacle to load balancing and performance tuning [10]. Binding threads and data

to specific sockets and carefully selecting the sockets they are bound to both reduced variability and improved overall performance of the benchmarks.

Underwood [17] discussed the mismatch between frequently used MPI microbenchmarks and the setting which MPI functions encounter in real-world applications, reporting an execution time difference up to a factor of four in extreme cases.

Mamidala et al. [9] investigated performance of MPI collectives on contemporary multi-core architectures. They concentrate on exploiting features of modern multi-core architectures (e.g. shared caches) for improving *average* performance of selected collectives. Their work does not consider execution time deviations of identical function calls. Mamidala et al. show more efficient ways to implement collectives while our work demonstrates the behaviour of a given implementation in the dynamic setting inherent to multi-core distributed shared-memory computers. Our work is complementary, as Mamidala et al. try to understand and reduce average execution time while we try to understand and improve execution time variability.

Hoefler and Lumsdaine investigated the performance of non-blocking MPI collectives on Infiniband and suggested measures for improving overlap of communication and computation [6]. They showed that performance can be improved considerably. They do, however, not consider inter-chip networks but only inter-node networks (Infiniband).

AMD provides a technical report "Performance Guidelines for AMD Athlon and AMD Opteron cc-NUMA Multiprocessor Systems" [2] which summarises detailed measurements performed on a system with four dual-core AMD Opteron CPUs. A synthetic benchmark is used which comprises two tasks reading/writing data from/to independent memory locations. Execution times for all possible combinations of task and data placement are measured. Additional tasks read data from local memory to simulate background activity. The benchmark chosen explores how (remote) memory access translates into HyperTransport activity under varying task and data placement scenarios. In contrast to the data presented by AMD, we consider collective communication instead of point-to-point communication. Furthermore, while AMD creates a synthetic background activity, our goal is to infer unknown background activity patterns and its impact on execution time on a known collective communication.

In summary, existing work concentrates on cluster settings when evaluating overall application performance. In contrast, we argue that the performance of existing DSM computers is sufficient to run distributed applications entirely on a single DSM computer. Detailed performance analysis of DSM computers exist in literature but mostly focuses on the relative placement of tasks and data. Where communication functions are investigated, the aim is at reducing the *average* performance. To the best of our knowledge, our work is the first one investigating the execution time variance of collective communication due to background activity on a DSM computer's HyperTransport inter-chip network.

4. Experimental Setup

Our aim is to better understand execution times of MPI collective communication primitives on DSM computer inter-chip networks. To make insights attractive to as many distributed applications as possible, we have chosen a DSM computer with many cores and a complex inter-chip network.

The Sun Fire X4600 M2 server [15] fulfils these requirements by supporting up to eight quad-core CPUs, which results in an inter-chip network of the maximum size currently supported by the AMD Opteron architecture (8 sockets) and a maximum worst-case traffic pressure per link (up to four cores sharing a single link). The X4600's inter-chip network fully relies on functionality (cache coherency protocol, ...) and interfaces (HyperTransport) integrated in the AMD Opteron architecture. However, Opteron-internal tables specifying routing and hardware buffer sizes can be set at system start-up potentially leading to physically identical DSM systems executing identical applications yet exhibiting varying appliation performance.

The general findings of our experiments will therefore apply to a wide range of servers with similar architecture while the exact results of our measurement are obviously specific to the system used.

4.1 Hardware

Our prototypical DSM computer is a Sun Fire X4600 M2 server [15] by Sun Microsystems, which we will refer to as "X4600" in the following. The X4600 is designed to accept up to eight CPU/memory modules and can therefore exploit the maximum number of CPUs currently supported by AMD's Opteron 8000 CPU family [1]. The motherboard itself provides no memory or computing facilities but only module interconnect, power and I/O.

Each CPU/memory module carries local memory. The total of all local memory present on all modules is mapped by the operating system into a uniform address space (8×4 GB = 32 GB for our system).

Every CPU/memory module features a single CPU socket, which can be fitted with a single-core, dual-

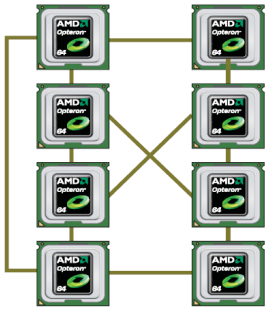

Figure 1. HyperTransport socket interconnect topology of a Sun Fire X4600 M2 server equipped with eight CPU modules.

core or quad-core AMD Opteron. The CPU used in our X4600 configuration is an AMD quad-core Opteron 8356. Each 8356 core features 2 MB private L2 cache while a 2 MB L3 cache is shared among all four cores. The cache-coherency protocol guarantees that all existing cache copies of data in memory are refreshed when data is modified anywhere in the system.

The AMD Opteron architecture integrates all memory controller functionality and three HyperTransport interfaces on-die [4]. The latter makes it possible to build servers with a very dense inter-socket communication network [8]. The AMD Opteron 8356 HyperTransport interfaces comply to HyperTransport 1.0, specifying 16 bit wide links with a clock frequency of 1 GHz. The links work in double-data-rate mode which results in a total bandwidth of 4 GB/s per direction.

Our system is equipped with eight CPU/memory modules. Sockets 0 and 7 dedicate one of their three links to connect the inter-socket network to system I/O. Figure 1 shows the X4600's inter-socket network topology ("twisted ladder"). Our X4600's inter-chip network therefore consists of 22 unidirectional HyperTransport links, while the two remaining links connect the network to system I/O facilities (hard disk drive, network, ..).

4.2 Operating System, Middleware

The used operating system is OpenSolaris 10 5/09. OpenSolaris features memory placement optimisation (MPO) which attempts to allocate memory as near to a process as possible [13, 14]. While the Solaris scheduler is able to move threads between all available cores (and therefore also between sockets), data remains by default on the CPU/memory module where it was first allocated.

The MPI distribution used is OpenMPI 1.3. OpenMPI provides support for core binding, i.e. manually assigning an MPI process to a core. We always bind all processes to distinct cores with the root process being assigned to core 4 (i.e. the first core on the second socket, thereby avoiding socket 0 through which I/O access is routed).

The AMD Opteron architecture provides hardware event counters to measure link load [1]. We have used the Solaris `lcpc(3CPC)` library for setting up and reading out hardware event counter values.

5 Experiments

We have chosen the `MPI_Allreduce` function as a prototypical MPI collective communication function. In this operation, all processes send arrays of identical size and type to the root process. There, entries of the same index are reduced using a specified arithmetic function.

In terms of communication performance, it would suffice to consider `MPI_Allgather`, as `MPI_Allreduce` can be assembled from an allgather operation followed by some local computation. `MPI_Allreduce`, however, natively integrates this computation following communication and therefore provides better workload characteristics in terms of possible interference between communication and computation.

Each process is bound to a specific core. No specific measures are taken to guarantee placement of data in local memory.

There is no explicit waiting between consecutive calls of `MPI_Allreduce`. While this might be unrealistic in most application settings, it maximises stress on the inter-chip network and therefore allows observation of effects which might only be visible sporadically otherwise.

Using hardware counters accessible via libcpc, we measure the link load (i.e. sent/received data words on the observed link in the given time interval) in both directions on all links during execution of a given communication function (48 measurements). Specifically, we monitor the Opteron's "Link Event" registers (0F6h, 0F7h, 0F8h, 1F9h, "HyperTransport Link x Transmit Bandwidth", see [1] for full details).

5.1 `MPI_Allreduce` with 8x4 processes

We measure the execution time of an `MPI_Allreduce` function call (using `hrtimer()`) collecting and processing messages of 16kB each from 32 MPI processes. Additionally, we monitor the traffic on all HyperTransport links during execution of `MPI_Allreduce`. The measurements are repeated for consecutive 2000 calls of `MPI_Allreduce`.

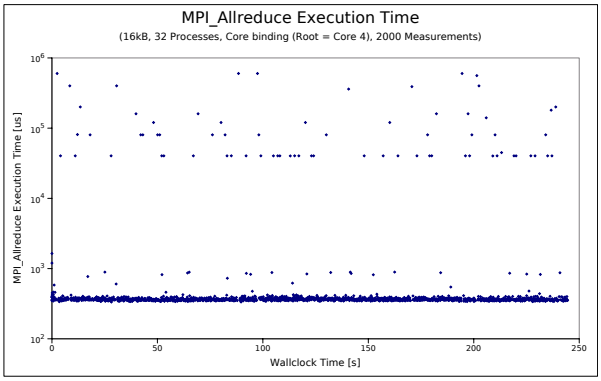

Figure 2. Execution times of 2000 consecutive `MPI_Allreduce` calls.

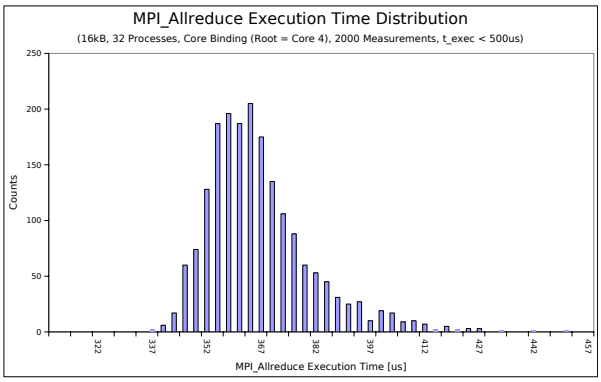

Figure 3. Execution time distribution of `MPI_Allreduce` calls executed in less than 500μs .

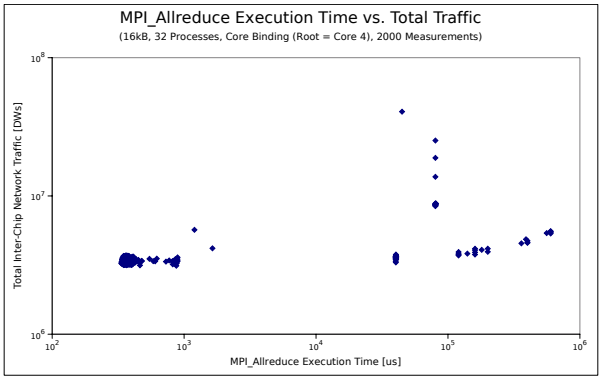

Figure 4. Execution time of `MPI_Allreduce` versus total inter-chip network traffic.

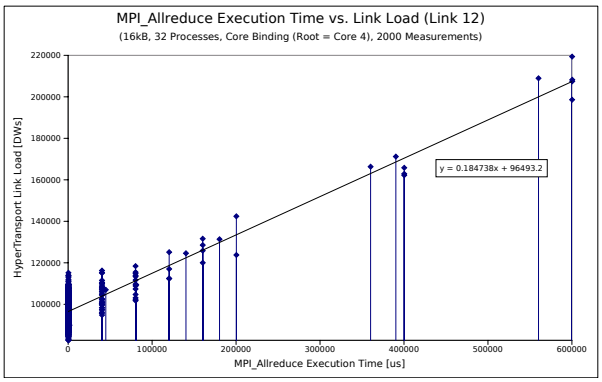

Figure 5. Execution time of `MPI_Allreduce` versus traffic on HyperTransport link 12.

Figure 2 shows the execution time of each `MPI_Allreduce` call over wallclock time (i.e. the x-axis corresponds to time progress during experiment). Most calls take less than 1000 μs (the median of all measurements is 363 μs). However, some execution times deviate considerably with maximum execution times up to 600 ms!

More than 95% of all measurements result in an execution time smaller than 500 μs. Figure 3 shows the distribution of these measurements.

While the majority of calls is very fast, the remaining calls consume an disproportional amount of time. The accumulated execution time of the 100 slowest calls (5%) consumes 93% of the overall sum of all execution times.

We hypothesise that the longer execution times can be explained by activity on the inter-chip network resulting in reduced available bandwidth on some HyperTransport links. In the following, we focus on relating `MPI_Allreduce` execution time with HyperTransport link load.

A first naive approach could be to relate execution time to the overall traffic on the inter-chip network during execution of each call as shown in Figure 4. No obvious correlation can be identified.

We use GGobi [16] to interactively explore the 23-dimensional space spanned by our measurements (22x HyperTransport outgoing link traffic, 1x MPI_Allreduce execution time) and find that some link traffic data is positively correlated with the `MPI_Allreduce` execution time. Figure 5 shows the traffic on link 12 over the execution time of `MPI_Allreduce`. The positive correlation is obvious. Similar correlation exists for data from several links.

The correlation observed is sufficient to distinguish short, medium and long execution times by single link load observations.

The observed link load stems from at least one collective communication (initiated by our foreground task) and multiple additional (point-to-point and maybe col-

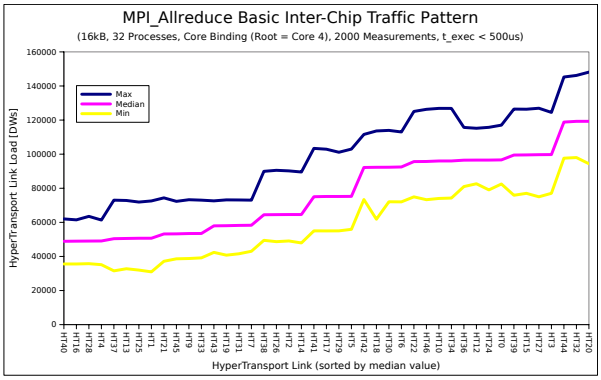

Figure 6. Min/max/median HyperTransport link load for calls of `MPI_Allreduce` with an execution time smaller than 500μs.

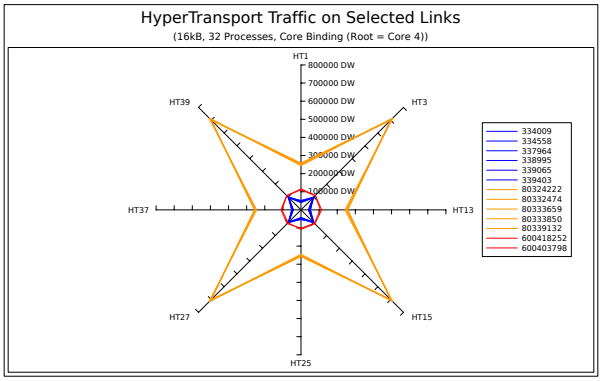

Figure 7. HyperTransport link load for selected calls of `MPI_Allreduce` on Links 1, 3, 13, 15, 25, 27, 37 and 39.

lective) communication triggered by background tasks (scheduler, cache coherency protocol, I/O activity, ...). According to Figure 1, messages exchanged between cores on different sockets can lead to routing of the message through up to three HyperTransport links. We therefore try to identify traffic patterns rather than simple link load to explain execution times.

Figure 6 shows the observed minimum, maximum and median HyperTransport link load for all links when inspecting data for all calls of `MPI_Allreduce` which result in an execution time smaller than 500μs. The links are ordered by their median load.

To identify distinct traffic patterns being related with specific execution time levels, we use GGobi's automatic brushing tool which allows colouring of data in all plots according to an additional given parameter (execution time in our case). Inspection reveals that increased activity on Links 1, 3, 13 and 15 corresponds to an execution time of about 80ms (the third cluster from left in Figure 5) . Figure 7 shows the activity on each of the links for some measurements resulting in high (red), low (blue) and moderate (about 80ms, orange) execution times respectively. The large star pattern is formed by measurements resulting in execution times of around 80ms exclusively.

5.2 Discussion of measurements

Our measurements of 2000 consecutive `MPI_Allreduce` calls reveal that while most calls (95%) finish within a very short time (less than 500 μs, median is 362 μs), the remaining 5% consume 93% of the experiment's run time.

It is possible to identify HyperTransport links whose load is positively correlated with `MPI_Allreduce` execution time. While not a very accurate indicator, single link load data of selected links seems sufficient to separate typical execution times from pathological cases.

The distribution of execution times is not continuous but shows strong clustering. Mining the measurements for correlations between clusters of similar execution time has revealed that increased traffic on a set of links directly corresponds to execution times within the cluster. This insight can be used to improve accuracy of the above indicator.

6 Conclusions and Outlook

We have shown that `MPI_Allreduce` execution time is correlated with HyperTransport link load. This is an important observation as a multitude of root causes might originally be involved, leading ultimately to the varying execution times observed. Relying on the correlation identified, we can focus on a much smaller set of observables. Current CPU architectures provide on-chip hardware performance counters for monitoring of inter-chip network traffic which allows link loads to be observed easily from a user application at run-time.

Which links need to be observed is a function of the full set of communication triggered by both foreground and background tasks. Our method of identifying relevant links relies on visual inspection of data which implies a big overhead in case substantial changes to the set of tasks are made. It would therefore be most desirable to partially automate the process of identifying relevant links.

Different implementations of `MPI_Allreduce` lead to different communication patterns. Therefore our findings only apply to the specific implementation of`MPI_Allreduce` in the used OpenMPI version.

We have considered a single foreground traffic pattern (`MPI_Allreduce`). Further work will investigate other MPI collective communication functions and the effects they will encounter when being executed on an inter-chip network with varying load.

We have not actively triggered any background communication activity. The varying execution times observed show that symmetric multi-core architectures in use today sporadically exhibit extremely asymmetric performance behaviour. This is due to the asymmetry of the communication infrastructure (see Figure 1) as well as conflicting resource usage by competing user and system tasks and communication stack deficiencies (see [12]).

There are two ways how our findings could be applied: First, it could be used to construct a predictor for execution times of selected communication functions under dynamic load situations. Second, it could also be used as a bottom-up analysis tool for system activity affecting the execution time of communication.

We plan to extend our work by identifying relevant background tasks and reducing their activity if possible. We will as well equip our benchmark with the cheap predictor proposed in this work. A simple measure to show the viability of our predictor in a conventional MPI setting would be to postpone execution of communication calls if the predictor suggests very long execution times.

During preparation of this work, the maximum number of cores available on an AMD Opteron CPU has increased from four to twelve. As a consequence, larger distributed applications can be run on a single server, increasing the complexity of traffic patterns on the inter-chip network while relying heavily on its performance.

7 Acknowledgements

We thank the X4600's system administrator Martin Paul for his support and Richard Smith of Sun Microsystems for his helpful comments and pointers on Opteron architecture and hardware event counters under Solaris.

This work was partially supported by the CPAMMS project of the University of Vienna (FS397001) and by the NFN S106 (SISE) of the Austrian Science Fund FWF.

References

[1] AMD. *BIOS and Kernel Developer's Guide for AMD Athlon and AMD Opteron Processors*, February 2006.

[2] AMD. *Performance Guidelines for AMD Athlon and AMD Opteron ccNUMA Multiprocessor Systems*. Advanced Micro Devices, Inc., 3.00 edition, June 2006.

[3] L. Chai, Q. Gao, and D. K. Panda. Understanding the impact of multi-core architecture in cluster computing: A case study with intel dual-core system. In *Cluster Computing and the Grid, 2007. CCGRID 2007. Seventh IEEE International Symposium on*, pages 471–478, 2007.

[4] P. Conway and B. Hughes. The AMD Opteron northbridge architecture. *Micro, IEEE*, 27(2):10–21, 2007.

[5] W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, B. Nitzberg, W. Saphir, and M. Snir. *MPI: The Complete Reference*. The MIT Press, September 1998.

[6] T. Hoefler, P. Gottschling, and A. Lumsdaine. Leveraging non-blocking collective communication in high-performance applications. In *SPAA '08: Proceedings of the twentieth annual symposium on Parallelism in algorithms and architectures*, pages 113–115, New York, NY, USA, 2008. ACM.

[7] A. Kayi, E. Kornkven, T. E. Ghazawi, and G. Newby. Application performance tuning for clusters with cc-NUMA nodes. In *CSE '08: Proceedings of the 2008 11th IEEE International Conference on Computational Science and Engineering*, pages 245–252, Washington, DC, USA, 2008. IEEE Computer Society.

[8] C. N. Keltcher, K. J. Mcgrath, A. Ahmed, and P. Conway. The AMD Opteron processor for multiprocessor servers. *Micro, IEEE*, 23(2):66–76, 2003.

[9] A. R. Mamidala, R. Kumar, D. De, and D. K. Panda. MPI collectives on modern multicore clusters: Performance optimizations and communication characteristics. In *Cluster Computing and the Grid, 2008. CCGRID '08. 8th IEEE International Symposium on*, pages 130–137, 2008.

[10] A. Mandal, A. Porterfield, R. J. Fowler, and M. Y. Lim. Performance consistency on multi-socket AMD Opteron systems. Technical Report TR-08-07, RENCI, North Carolina, 2008.

[11] A. Porterfield, R. Fowler, A. Mandal, and M. Y. Lim. Empirical evaluation of multi-core memory concurrency. Technical Report TR-09-01, RENCI, North Carolina, January 2009.

[12] T. Scogland, P. Balaji, W. Feng, and G. Narayanaswamy. Asymmetric interactions in symmetric multi-core systems: analysis, enhancements and evaluation. In *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, pages 1–12, Piscataway, NJ, USA, 2008. IEEE Press.

[13] Sun Microsystems. Solaris memory placement optimization and Sun Fire servers. Technical report, Sun Microsystems, March 2003.

[14] Sun Microsystems. *Memory and Thread Placement Optimization Developer's Guide*, June 2007.

[15] Sun Microsystems. Sun Fire X4600 M2 server architecture. Technical report, Sun Microsystems, June 2008.

[16] D. F. Swayne, D. Temple Lang, A. Buja, and D. Cook. GGobi: evolving from XGobi into an extensible framework for interactive data visualization. *Computational Statistics & Data Analysis*, 43:423–444, 2003.

[17] K. Underwood. Challenges and issues in benchmarking MPI. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 339–346. Springer, 2006.