ACSIJ Advances in Computer Science: an International Journal, Vol. 3, Issue 3, No.9 , May 2014
ISSN : 2322-5157
www.ACSIJ.org

ACSIJ
WWW.ACSIJ.ORG

Task Parallel Models Based on Dynamic Data Placement to Reduce NUMA Effects

Yan Wang[1] and Brian Vinter[2]

[1] Niels Bohr Institute, University of Copenhagen
Copenhagen, DK-2100, Denmark
yanwang@nbi.ku.dk

[2] Niels Bohr Institute, University of Copenhagen
Copenhagen, DK-2100, Denmark
vinter@nbi.ku.dk

Abstract

NUMA (Non-Uniform Memory Access) multicore computers become popular in scientific and industrial fields due to its scalable memory performance. However, large-scale intensive data computing on NUMA architecture are facing up to the challenges in data locality problems called NUMA effects that are caused by the overhead accesses of cross-node data. Our task parallel model bases on the strategy of dynamic data placement improving system performance by reducing the frequently data access to remote memory and also by keeping load balance between each NUMA domain. The task parallel models involved OpenMP, numactl and libnuma. The evaluation demonstrates that the benchmarks using our task parallel models on a 32-core NUMA computer with various workloads achieve system performance improvement by 50% at least.

Keywords: *NUMA Architecture, Task Parallel Model, Dynamic Data Placement, NUMA Effects*

1. Introduction

In a sense, NUMA and ccNUMA (cache coherent NUMA) are synonymous. Because the applications for non-cache coherent NUMA machines are almost non-existent and it is different to program for it. Unless specifically stated, NUMA actually means ccNUMA. Compared to NUMA architecture, the SMP (symmetric multiprocessing) architecture has been used widely in small-scale multiprocessor computers. SMP computer uses a single-shared system bus to connect up to 8 processors that have full access to all Input/Output devices. Those processors are controlled by a single OS (operating system) that treats all processors equally and reserve none for special purposes. Usually each processor has an associated private high-speed memory known as cache memory to speedup the shared memory data access and to reduce the system bus traffic. SMP works fine for a relatively small number of CPUs (Central Processing Unit), but the problem with shared bus appears when there are hundreds of CPUs competing for access to shared memory bus. Therefore, a number of SMP computers are integrated as NUMA architecture that become popular because of its scalability and straightforward implementation. Besides, both of message passing and shared memory programming models can establish on NUMA architectures. Therefore, NUMA architectures provide a great tradeoff between cost and feasibility to form an adequate environment for high performance computing [1].

NUMA alleviates the bottlenecks of shared memory by limiting the number of CPUs on any one-memory bus and connecting the various nodes by means of a high speed interconnect, illustrated in fig 1. This is 32-core NUMA computer we adopt in this paper. In this scenario, NUMA domain is a set of CPUs that all access to the same memory (called local memory) at the same speed. There are three levels of cache memory that bridge the speed gap between cores and the local memory. It should be noted that all the cores in the same NUMA domain share the third level cache memory. Since the low speed of accessing to main memory cannot match the high speed R/W (read/write) that CPUs require, modern processors provide fast local memory called cache to bridge this gap. According to the principle of locality, cache memory is particularly effective for reusing data. The principle of locality also called locality of reference includes two essential types, i.e. temporal locality and spatial locality. Temporal locality means the data that has been used recently may have a high likelihood of being used again. Spatial locality means that the data is fetched from main memory to cache memory in blocks called cache lines as there is a high likelihood that the data nearby will be used together [2]. The NUMA domains in the same processor can communicate by QPI (quick path interconnect). The memory locates on the other processors called remote memory, connected by a fast interconnection called

hypertransport. From the software point of view the remote memory can be used in the same way as local memory. It is fully cache coherent. The only difference is that access to remote memory is slower than to local memory. The main benefit of NUMA is, as mentioned above, scalability. It is extremely difficult to scale SMP more than 8 CPUs. Therefore, NUMA is a good way to reduce the number of CPUs competing for access to a shared memory bus.

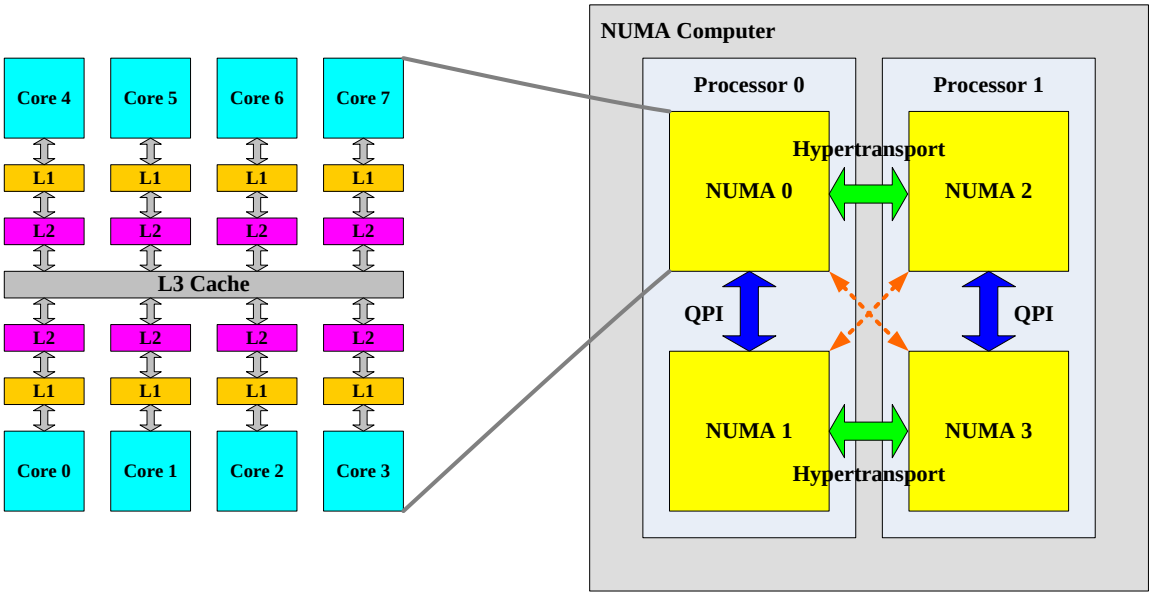

Fig. 1 32-core NUMA Computer

NUMA effects come from the memory latency between local and remote allocations. NUMA effects arise when threads excessively access memory on a different NUMA domain. As we mention above, the access speed to remote memory is slower than the speed to local memory. So we should avoid the NUMA effects. The concept of affinity is relative to reducing NUMA effects. There are two types of "affinity" in NUMA architectures, i.e. CPU affinity and memory affinity.

1) CPU affinity is to bind a process or thread to a particular core. If the operating system interrupts the task, it doesn't migrate it to another core but waits until the core is free. In most HPC scenarios where only one application is running on a node, these interruptions are short.

2) Memory affinity is to allocate memory as close as possible to the core on which the task that requested the data is running.

Both CPU affinity and memory affinity are important if we are to maximize memory bandwidth on NUMA nodes. If memory affinity is not enabled, bandwidth will be reduced as we go off-socket to access remote memory. If CPU affinity is not enabled then allocating memory locally is of no use when the task that requested the memory might no longer be running on the same socket [3] [4].

The difficult part to reduce NUMA effects is to determine which NUMA domain is the best one to allocate memory on. It brings the concept of "data placement" that is the program's memory access pattern. The placement policies determine how the memory of virtual pages is allocated. There are three data placement policies wildly used, i.e. first touch, fixed and round robin.

1) "First touch" is default preference of Memory Binding. The memory is allocated on the NUMA Domain containing the page-faulting CPU cores.

2) "Fixed" allocates memory from a specified subset of NUMA domains based on virtual address.

3) "Round robin" allocates memory in sequence from a specified subset of NUMA domains.

For "first touch", it is important that user programs' memory is allocated on a NUMA domain close to the one

containing the CPU on which they are running. Therefore, by default, page faults are satisfied by memory from the NUMA domain containing the page-faulting CPU. The program's topology is that the processes making up the parallel program should run on NUMA domains that minimize access costs for data they share. The page placement is that the memory a process accesses the most often should be allocated from its own NUMA domain, or the minimum distance from that NUMA domain [5][6].

Moving pages between NUMA domains as an application is running is dynamic data placement. It is data-oriented cache memory optimization technology. There are two major requirements of dynamic data placement that we meet in NUMA architectures, i.e. loading data to appropriate NUMA domain and keeping load balance between different NUMA domains. In those policies, pages can either be migrated or replicated. Migration involves the relocation of a page to a new home NUMA domain. Replication involves the creation of a "shadow" of the page on another NUMA domain. It should be noted that cache coherency is still maintained by hardware on a cache block basis.

Since NUMA largely influences memory access performance, certain software optimizations are needed to allow scheduling threads and processes close to their data. In this work, according to the data's priority, an auto-tuning scheduler dynamically load data from main memory to cache memory or if necessary from remote memory to local memory. The primary way to do this is to allocate memory for a thread on its local NUMA domain and keep the thread running there. This gives the best latency for memory and minimizes traffic over the global interconnect. SMP Systems try to optimize the use of every cache memory in the similar way. There is an important difference: on a SMP system when a thread moves between CPUs, its cache contents will eventually move with it; on a NUMA system once a memory is allocated to a specific NUMA domain, even if a thread running on a different node the memory is not moved. It will always arise traffic for interconnect.

The dynamic data placement includes memory allocation using implicit operating system policies and the use of the system APIs for assigning and migrating memory pages using explicit directives. From data management perspective, the physical location of data and the characteristics of the underlying query engine are transparent to applications. In this work, our experiment illustrates the challenges behind the optimization for data placement and data access on NUMA architecture.

2. Backgrounds and Motivations

Before this work, we have designed an auto-tuning JIT compiler for accelerating multiple stencil computations, in particular for processing large-scale scientific images, such as astronautics, biologics and geographic images. With this tool, scientists can conveniently run high-performance parallel program of image processing. As showed in Figure 2, They don't have to pay attention to the hardware's configuration, such as, cache memory size and the number of CPU core and so on. They even don't have to own professional program skills. What they need to do is to provide the execution order of image algorithms and the target images' address. This tool is even simpler than Matlab to use. We chose edge detection as our benchmark. Compared with the sequential naive program, execution optimized by the JIT compiler achieves linear acceleration performance on SMP platforms. Unfortunately, the performance promotion on NUMA platforms is not as exciting as we expect. The system's bottleneck comes from the data movement between cross-node and the imperfect task scheduling strategy. Since NUMA architecture is widely used for scientific computations, it becomes essential to dominate this bottleneck.

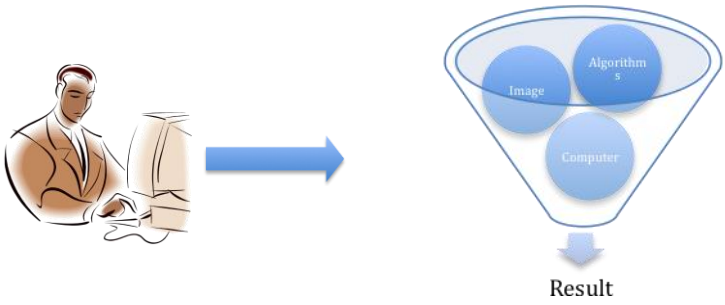

Fig. 2 User Perspective of Auto-Tuning JIT Compiler

3. Task parallel models

SIMD (Single instruction, multiple data) architectures can exploit significant data-level parallelism for vector-oriented scientific computing as well as for media-oriented image and sound processing. In this work, we focus on optimizing SIMD applications. Most SIMD applications are memory bandwidth limited and need to make better use of cache memory.

3.1 Programming Tools

There are several common programming tools used for task parallel models on NUMA architectures.

1) OpenMP (Open Multi-Processing)

It is an API that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran.

2) MPI (Message Passing Interface)

MPI is a standardized and portable message passing system designed for various parallel computers.

3) POSIX Threads

POSIX Threads also called Pthreads is a POSIX standard for threads. The standard defines an API for creating and manipulating threads on Unix-like OS.

4) numactl

It is a command line tool to run processes with a specific NUMA policy.

5) libnuma

It is a shared library that can be linked to programs and offers a stable API for NUMA policy.

6) Unix-like System Calls

The system call interface is declared in *numaif.h*. Since the higher-level interface libnuma has involved the system calls, we would not call system calls directly in this work.

In our task parallel models, numactl and libnuma are involved for reducing NUMA effects. OpenMP used for parallelism is transparently extended for non-shared memory system. In an OpenMP application running on one node, the threads running on any NUMA domain see one unified memory space and therefore can read and write to remote memory that is local to other NUMA domains.

3.2 Memory Affinity

NUMA APIs can use the MMU (Memory Management Unit) in CPU to allocate memory. The consecutive pages can be mapped into different NUMA domain. As we mention above, there are three data placements used for NUMA memory policies, i.e. first touch, fixed and round robin. The libnuma allocation functions should be used for allocating large memory objects that exceed the cache memory size. It allocates memory by pages (4KB for AMD64 systems) [7].

There are two approaches to enhance memory affinity, i.e.:

1) Reducing the amount of shared data by duplicating the data

2) Reducing the interaction between different nodes on algorithm aspect

Both of those approaches can reduce the remote memory accesses that are far slower than local memory accesses. However, there is a tradeoff between the time consumption for one-time load data and the performance decline by remote memory accesses in the first approach.

3.3 CPU affinity

Linux kernels can bind threads to specific CPUs (using the system call *sched*). NUMA API extends this to allow

programs to specify on which node memory should be allocated. One NUMA policy called CPU affinity is to run threads on the special nodes. By the use of libnuma, CPU affinity is done by the *numa_run_on_node()* function which binds the current thread to all CPUs in node. *Numa_run_on_node_mask()* binds the current thread to any of the CPUs included in a nodemask.

The NUMA API separates placement of threads to CPUs and placement of memory. Primarily it is concerned about the placement of memory. In addition the application can configure CPU affinity separately by using numactl. Here is an example on how to use numactl.

```
numactl --cpubind=0 --membind=0,1 program
```

Run program bound to the CPUs of node 0 and only allocating memory from node 0 or 1. It should be noted that *cpubind* uses node numbers, not CPU numbers. On a system with multiple CPUs per node these can be different. [7]

As we mention above, a scheduler using NUMA API or system calls can bind threads to special CPU cores, according to the system state and various optimization policies. Before the given thread being swapped out of the core, other threads have no chance to execute on this core. To keep load balance, the waiting threads may be migrated to another core to ensure timely execution. We expect that all the threads run on the NUMA domain where their memory is allocated on. The access to remote memory will be avoided. Unfortunately, in some scenarios, the threads will be migrated to another CPU core which memory is not local memory where the data stored. It bring heavy memory access though interconnect between different NUMA domain. The bottleneck causes the sharp decline in system performance [8][9].

3.4 Dynamic Data Placement Policy

Our data placement policy is to allocate memory as close as possible to the core on which the threads that requested the memory is running. Both CPU affinity and memory affinity are important to maximize memory bandwidth on NUMA nodes. If memory affinity is not enabled, then accessing of remote memory will reduce the bandwidth. If CPU affinity is not enabled, then allocating memory locally is of no use when the task that requested the memory might no longer be running on the same node.

As mentioned above, there are three data placement policies wildly used, i.e. first touch, fixed and round robin. The default policy is first-touch. It works well with single-threaded programs, because it keeps memory close to the program's one process. It also works well for programs that have been parallelized completely, so that each parallel thread allocates and initializes the memory it uses. The policy of fixed is useful when the programmer specifies the optimal placement of data. For situations where a large number of threads will randomly share the same pool of data from all nodes, we recommend round robin for data placement. It avoid bottleneck when access pattern on s single node if the memory accesses are divided for all the thread.

At run time, our scheduler automatically checks the algorithm types and then chooses an appropriate data placement policy. Besides, when we find the memory allocated on an inappropriate place at run time, a useful capability through NUMA APIs is memory migration. It is quite expensive to migrate memory pages from one NUMA domain to another. When the application is both long-lived and memory intensive, it is not a good idea to migrate memory pages. It is a tradeoff between re-establish a cache-friendly environment and migrate huge data cross nodes.

4. Evaluation

We run benchmark on a 32-core NUMA computer that is two AMD OPTERON^{TM} processors 6272. Its CPU frequency is 2.1GHz. Its core number is 16/processor. Level 1st cache memory size is 16KB for data and 64KB for instructions. Level 2nd cache memory size is 1MB(x16) and Level 3rd cache memory size is 16MB. We choose laplacian filter as our benchmark. The laplacian filter using 3×3 mask is presented as follow:

```
for i=1 to (height-1)
  for j=1 to (width-1)
    index=0;
    sum=0;
    for m=(i-1) to (i+1)
      for n=(j-1) to (j+1)

sum=sum+oldData[m,n]*mask[index++];
    newData[i,j] = sum;
```

Figure 3 plots the speedup ratios of different data placement policies compared to the sequential codes, versus the number of threads. Larger speedup ratios are better. The results are for the laplacian filter as mentioned above.

1) OpenMP + non-NUMA strategy

The program runs in parallel by OpenMP. The threads do not bind to any CPUs and the memory is allocated on random nodes.

2) OpenMP + First Touch

The memory is allocated on the NUMA nodes containing the page-faulting CPU cores.

3) OpenMP + Round Robin

Interleave memory allocates on a sequential set of NUMA nodes.

With NUMA API libnuma, the programs can bind the threads to a particular node and also guarantee that the memory is allocated on a particular node. There are 4 NUMA nodes (8 CPU cores/node) in the NUMA computer on which the benchmark runs on. The operation matrix of laplacian filter in this benchmark is 10,000x10,000 pixels.

It illustrated in Figure 3 that first touch policy approaches better performance than other policies when the threads increase. The program without NUMA aware dynamic data placement policy is slower than the programs improved by NUMA API. It proves that our task parallel models can speedup the laplacian filter processing large-scale vectors on the NUMA computers. We trust those strategies can be adopted by other algorithms that process large-scale vectors.

From figure 4 we also obtain the similar conclusions as figure 3. It should be noted that the number of threads created in figure 4 is the same as the CPU cores. It means that there is not thread waiting for CPU core. In those scenarios, we focus on improving system performance by enhancing memory affinity. We try to reduce the traffic over interconnect by ensuring most threads load data from local NUMA node. The first touch policy is slightly better than round robin policy for memory allocations in this benchmark.

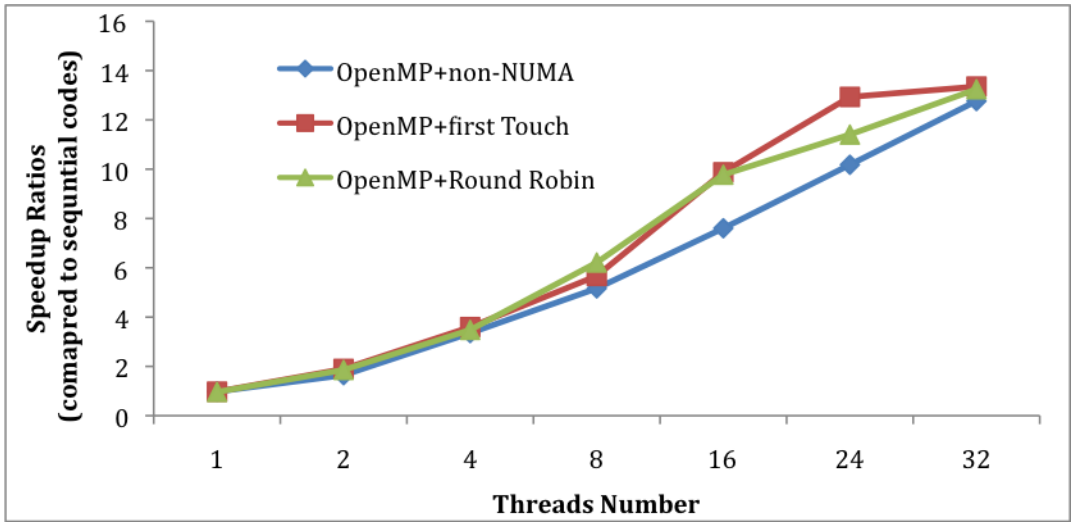

Figure 3 Performance of different Data Placement, compared to sequential codes with different threads numbers

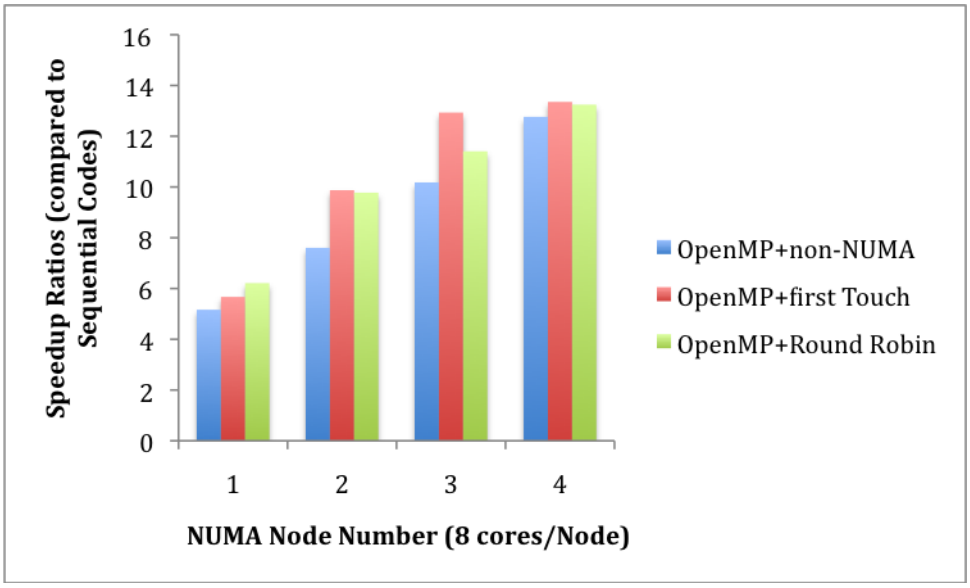

Figure 4 Performance of different Data Placement, compared to sequential codes with different NUMA node number

5. Related work

There are lots of related works including, but not limited to, measuring NUMA effects, data placement, NUMA APIs and so on. Since NUMA architectures becomes popular in scientific and industrial fields. It brings great challenges to transfer the earlier projects, most of which do not satisfy with NUMA architectures, onto NUMA systems.

Lars Bergstrom described and measured the memory topology of two different high-end machines using Intel and AMD processors. These measurements demonstrate that NUMA effects exist and require engineering beyond that normally employed to achieve good locality and cache use [3]. Tim Kiefer et al. show that partitioning a database's memory with respect to the data's access patterns can improve the query performance by as much as 75%. They use a self-developed synthetic, low-level benchmark as well as a real database benchmark executed on the MySQL DBMS to verify their hypotheses. They also give an outlook on how their findings can be used to improve future DBMS performance on NUMA systems [4]. In Iakovos Panourgias's thesis, they investigates the effects of NUMA architecture on performance. They found that the location of memory on a multi socket platform could affect performance. They believe that it is preferable to override the default processor binding in most case [10].

Tevfik Kosar's dissertation propose a framework that de-couples computation and data placement, allows asynchronous execution of each, and treats data placement as a full-fledged job that can be queued, scheduled, monitored and check-pointed like computational jobs. They regard data placement as an important part of end-to-end process, and express this in a workflow language [11]. Bo Wu et al. examine the implications that modern heterogeneous Chip Multiprocessors (CMP) architecture imposes on the optimization paradigm. They develop three techniques to enhance the optimizations. They prove that working with a dynamic adaptation scheme, the techniques produce significant performance improvement for a set of dynamic simulation benchmarks. [12]

Libnuma and numactl are the most common NUMA APIs [7]. There are several special-purpose NUMA APIs, such as, NUMA-ICTM [13], minas [14], vNUMA-mgr [15] and so on.

6. Conclusions and future work

The key issue in determining whether the performance benefits from NUMA architecture is data placement. The more often that data can effectively be placed in local memory, the more overall access consumptions will benefit from NUMA architecture. Data placement issues do not arise for all parallel programs. Dynamic data placement should be considered only for the parallel programs that are memory intensive and are not cache friendly. It can achieve high system improvement when

false sharing, and meanwhile other forms of cache contention are not problems. Our strategies are especially useful for the data-intensive parallel program. By providing each node with its own local memory, memory accesses can avoid throughput limitations and contention issues associated with the shared memory bus.

This task parallel model will be improved for Heterogeneous Computing System in future. We hope it could achieve high system performance as well.

Acknowledgements

This work is funded by grant # 09-067060 from the Danish Council for Strategic Research.

References

[1] J. Tao, "Data Locality Optimization of Shared Memory Programs on NUMA Architectures Using an Integrated Tool Environment", Ph.D thesis, Technischen Universität München, Munich, Germany, 2002

[2] J. L. Hennessy and D. A. Patterson, Computer architecture: A Quantitative approach, Fourth Edition, Singapore, Elsevier, 2007

[3] L. Bergstrom, Measuring NUMA effects with the STREAM benchmark, Technical Report TR-2012-04, Department of Computer Science, University of Chicago, 2012.

[4] T. Kiefer, B. Schlegel and W. Lehner, "Experimental Evaluation of NUMA Effects on Database Management Systems," in BTW, 2013, pp.185–204.

[5] Silicon Graphics library, Origin2000™ and Onyx2™ Performance Tuning and Optimization Guide, chapter 8, Document Number: 007-3430-002

[6] http://lse.sourceforge.net/numa/status/description.html, Linux Support for NUMA Hardware, NUMA Status: Item Definition, on-line manual

[7] A. Kleen, "A NUMA API for Linux", SUSE Labs white paper, 2004.

[8] D. Ott, Optimizing Applications for NUMA, technical Report, Intel, 3011

[9] C. Lameter, "Local and remote memory: Memory in a Linux/NUMA system". ftp://ftp.kernel.org/pub/linux/kernel/people/christoph/pmig/numamemory.pdf 2006.

[10] Iakovos Panourgias, "NUMA effects on multicore, multi socket systems", *The University of Edinburgh*, 2011

[11] T. Kosar, "Data Placement in Widely Distributed Systems", Ph.D thesis, University of Wisconsin-Madison, USA, 2005

[12] B. Wu, E. Zhang, and X. Shen. "Enhancing data locality for dynamic simulations through asynchronous data transformations and adaptive control", In Proceedings of the International Conference on Parallel Architecture and Compilation Techniques (PACT), 2011

[13] M. Castro, L. Gustavo Fernandes, et al. "NUMA-ICTM: A Parallel Version of ICTM Exploiting Memory Placement Strategies for NUMA Machines". In: International Parallel and Distributed Processing Sympo- sium (IPDPS). Rome, Italy: IEEE Computer Society, 2009, pp. 1–8

[14] C. P. Ribeiro, M. Castro, et al. "Improving Memory Affinity of Geophysics Applica- tions on NUMA platforms Using Minas", In 9th International Meeting High Performance Computing for Computational Science, VECPAR, US, 2010. LNCS. 71, 134, 188

[15] D. Rao and K. Schwan. "vnuma-mgr: Managing vm memory on numa platforms." In HiPC, Goa, India, 2010.

Yan Wang received her bachelor of science from Northeast Normal University in 2006 and her master of engineering from Beihang University in 2009. She is a PhD student in the university of Copenhagen since 2010. Her researches focus on multicore computing and scientific computing.

Brian Vinter is a professor in the university of Copenhagen. He received his PhD from Tromsø University in 1999. His researches include grid computing, super computing and multicore architecture.