OpenMP Scalability Limits on Large SMPs and How to Extend Them.

Von der Fakultät für Mathematik, Informatik und Naturwissenschaften
der RWTH Aachen University zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften genehmigte Dissertation

vorgelegt von

Diplom-Informatiker Dirk Schmidl
aus Erkelenz

Berichter: Universitätsprofessor Dr. Matthias S. Müller
Universitätsprofessor Dr. Christian Bischof

Tag der mündlichen Prüfung: 28. Juni 2016

Zusammenfassung

Aktuell sind Rechenknoten mit zwei Prozessoren die am häufigsten verwendeten Knoten im Bereich des Hochleistungsrechnen. Viele tausend dieser Knoten können über ein schnelles Netzwerk miteinander gekoppelt werden zu einem Rechencluster. Um diese Cluster zu programmieren wird üblicherweise das Message Passing Interface (MPI) verwendet. MPI erfordert es die Parallelität und die verwendeten Datentransfers sehr explizit über Funktionsaufrufe zu realisieren.

Eine Alternative zu MPI, welche eine Parallelisierung auf höherer Ebene erlaubt ist OpenMP. In OpenMP können serielle Programme mit Pragmas angereichert werden um rechenintensive Teile der Anwendung parallel auszuführen. In vielen Fällen ist dies mit weniger Aufwand verbunden wie eine Parallelisierung mit MPI, bei der die gesamte Datenverteilung über alle Knoten im gesamten Programm implementiert werden muss. Der Nachteil von OpenMP ist, dass es nur auf Maschinen mit geteiltem Hauptspeicher und nicht auf den weit verbreiteten Clustern eingesetzt werden kann. Eine Reihe von Herstellern hat sich aber darauf spezialisiert große Maschinen mit geteiltem Hauptspeicher herzustellen. Da geteilter Hauptspeicher und damit einhergehende Anforderungen an die Koheränz der Speicher und Caches kompliziert zu implementieren sind, haben solche Maschinen Eigenheiten, die bei der Programmierung mit OpenMP berücksichtigt werden müssen um eine gute Parallelisierung zu erreichen.

In dieser Arbeit beschäftige ich mich damit die Eigenschaften verschiedener dieser großen Maschinen mit geteilten Hauptspeicher und die Programmierbarkeit mit OpenMP zu untersuchen. An Stellen an denen OpenMP nicht die nötigen Mittel für eine gute Parallelisierung liefert, werde ich Verbesserungen aufzeigen.

Weiterhin beschäftige ich mich in der Arbeit damit, wie Anwendungen mit OpenMP für solche Maschinen systematisch optimiert werden können. Hierbei wird die Nutzbarkeit von Performance-Analyse-Werkzeugen untersucht und Verbesserungen im Bereich der Task-basierten Analyse vorgestellt, welche die Optimierung für große Systeme vereinfachen. Abschließend stelle ich noch ein Modell vor, welches verwendet werden kann um eine Performance-Abschätzung für eine Anwendung auf einem solchen System vorzunehmen.

Abschließend wird anhand von zwei Anwendungen gezeigt, dass es die vorgestellten Optimierungen erlauben mit echten Nutzeranwendungen eine Skalierbarkeit mit OpenMP auf großen Systemen zu erreichen.

Abstract

The most widely used node type in high-performance computing nowadays is a 2-socket server node. These nodes are coupled to clusters with thousands of nodes via a fast interconnect, e.g. Infiniband. To program these clusters the Message Passing Interface (MPI) became the de-facto standard. However, MPI requires a very explicit expression of data layout and data transfer in a parallel program which often requires the rewriting of an application to parallelize it.

An alternative to MPI is OpenMP, which allows to incrementally parallelize a serial application by adding pragmas to compute-intensive regions of the code. This is often more feasibly than rewriting the application with MPI. The disadvantage of OpenMP is that it requires a shared memory and thus cannot be used between nodes of a cluster. However, different hardware vendors offer large machines with a shared memory between all cores of the system. However, maintaining coherency between memory and all cores of the system is a challenging task and so these machines have different characteristics compared to the standard 2-socket servers. These characteristics must be taken into account by a programmer to achieve good performance on such a system.

In this work, I will investigate different large shared memory machines to highlight these characteristics and I will show how these characteristics can be handled in OpenMP programs. When OpenMP is not able to handle different problems, I will present solutions in user space, which could be added to OpenMP for a better support of large systems.

Furthermore, I will present a tools-guided workflow to optimize applications for such machines. I will investigate the ability of performance tools to highlight performance issues and I will present improvements for such tools to handle OpenMP tasks. These improvements allow to investigate the efficiency of task-parallel execution, especially for large shared memory machines. The workflow also contains a performance model to find out how well the performance of an application is on a system and when to stop tuning the application.

Finally, I will present two application case studies where user codes have been optimized to reach a good performance by applying the optimization techniques presented in this thesis.

Contents

List of Figures

List of Tables

1 Introduction

Several programming paradigms exist for parallel programming today. Two industry standards have become the de-facto standards in high-performance computing for this purpose over the past decades, namely the Message Passing Interface (MPI) and Open Multi-Processing (OpenMP). While MPI can be used on shared and distributed memory systems, OpenMP requires a single system with a shared memory for execution. However, OpenMP allows to extend the time consuming parts of a serial program incrementally with directives for parallel execution, without the need to rewrite the overall application. MPI, in contrast, requires an explicit data distribution by the programmer and it requires to manually send messages whenever data needs to be transferred between processes. Therefore, the complete application needs to be adjusted to the distributed data layout, which requires a complete rewrite of the application code in many cases. MPI therefore is often called the "assembly language of parallel programming", as e.g. mentioned in [Mattson et al., 2004] and [Pacheco, 1996], whereas OpenMP was often found to be easier to use but it is also said to be very limited in scalability [Jost et al., 2003].

To combine the advantages of both paradigms, many approaches have been investigated to allow OpenMP programming on large core counts. One way is to couple multiple shared memory boards with a special interconnect that allows to maintain shared memory over all boards. SGI designed their proprietary NUMAconnect for this purpose, which allows to build shared memory systems with thousands of cores. Although, SGI was one of the first and maybe the most successful vendor producing large shared memory machines in the past, other vendors provide similar solutions as well. Bull, for example, completed the development of the bull coherence switch, a chip that can be used to couple standard 4-socket servers to a larger shared memory system with up to 16 processors. Numascale is another company which developed a chip to combine several boards to a single machine. This chip can couple hundreds of boards in a 3D-torus topology, forming single systems with thousands of cores. The company ScaleMP uses a different approach to reach the same goal. Instead of providing a hardware solution to build a single machine, ScaleMP offers a virtualization software, called vSMP Foundation,to provide shared memory on a cluster of standard servers. vSMP Foundation uses the standard infiniband network of the cluster for the actual data transfers, but provides the view of a shared memory and a single operating system image to the user.

All these machines allow to use OpenMP on hundreds or thousands of cores. Although every OpenMP program can be run on these machines, the performance is often worse than expected. On standard 2- or 4-socket servers the cache coherence is typically maintained by a protocol called snooping. This implies that all changes to a cache line, i.e. writes to a memory location, are propagated to all other caches in the system. This allows to invalidate all other copies of the cache lines in the system because these copies now contain an old value. The network traffic for such a protocol increases drastically with the number of processors in the system which need to be cache coherent, because every processor has changes to communicate and needs to invalidate own copies when others change those cache lines. Therefore, snooping does not scale for larger systems. All vendors of larger systems, mentioned before, use a different protocol to connect systems in a cache coherent manner. The second protocol is a directory based cache coherence protocol. Here, the owner of a cache line has a directory with instances holding a copy of this cache line. When the cache line is changed, the owner is informed and the owner informs all instances with a copy of the cache line. This reduces the traffic substantially, because typically a cache line is now not shared by all cores in a system but only by a few cores or not at all. Such a directory based protocol allows to build large systems, but the disadvantage is that a changes to a cache line are propagated slower, since first the owner of the cache line needs to be informed and second everyone with a copy must be informed, whereas snooping directly informs everyone in one step. As I will show later in this work, this results in different characteristics of large NUMA systems compared to standard 2- or 4-socket servers. To achieve performance a programmer needs to be aware of these characteristics and must keep them into account during the parallelization of his code.

This work presents new and extends well-known methods for application tuning on shared memory machines to achieve scalability to hundreds or thousands of cores with OpenMP parallel applications. I will first characterize different large shared memory machines, like an SGI Altix Ultraviolet, a Bull Coherence Switch system and a ScaleMP machine with the use of standard benchmarks as well as self-implemented tests to investigate performance relevant parameters of these machines and I will show how this information can be used to automatically generate a machine description as it is needed in OpenMP's affinity support (i.e. a `place-list`). I cover standard techniques and tools as a basis before I present self implemented methods for data migration and NUMA-aware load-balancing, as well as extensions to standard performance tools which allow to detect performance issues which could not be detected before, especially when OpenMP tasks are used. Finally, ways to model OpenMP performance especially on large NUMA machines are presented and I will present a recommended workflow to optimize applications for large NUMA machines to achieve this performance.

The rest of this work is structured as follows: First I will explain types of NUMA machines and introduce the systems used for experiments throughout this work in the rest of this chapter. Afterwards, performance characteristics of the investigated machines and different OpenMP runtime systems on these machines are evaluated and discussed in section 2, where I will also present how the gathered information can be used to set up a distance matrix as a way to describe the machine properties with respect to non-uniform memory access. I will cover general NUMA optimizations and present the OMPX library for memory placement in chapter 3, and investigate load balancing issues on large NUMA systems in chapter 4. In chapter 5 I will cover the ability of performance tools to handle OpenMP tasks and I will present extension to a performance measurement system, called Score-P, to improve the investigation of performance problems related to tasks on large SMP systems. Furthermore, I developed a performance model for OpenMP applications on NUMA systems and a workflow to optimize applications. This and two applications case studies which prove the optimizations useful on real application codes are presented in chapter 6. Finally conclusions are drawn in chapter 7.

1.1 Main Contributions

This work provides new contributions over state-of-the-art practices in the area or high-performance computing in the following fields.

First, as mentioned before, large shared memory machines exist from many vendors and with different underlying technical solutions. Characteristics of these different machines are investigated with the help of kernel benchmarks to classify the different architectures. Distance information of the NUMA nodes in a system are used to automatically build a *place-list*. A *place-list* is an abstract machine description used by OpenMP to support affinity. The generation of this list is left to the programmer by the OpenMP specification, so a systematic way to generate it can further ease the use of OpenMP's affinity support. Furthermore, the results of the kernel tests are used to form an easy-to-use performance model for OpenMP programs on these machines. Existing performance models nowadays are either very simple, like the roofline model [Williams et al., 2009], where peak-performance and memory bandwidth are the only restrictions for application performance, or models are extremely complicate to apply. E.g. in [Treibig and Hager, 2010] a model is presented to predict the performance of serial programs, based on an analysis of the generated assembly code and detailed hardware information about throughput and latency for all cache levels, based on processor specific vendor information. According to the authors, this is not done for parallel programs, because the transfers of cache lines cannot be predicted that accurate by looking at the assembly code only. In [Culler et al., 1993] or [Ramos and Hoefler, 2013] parallel applications are considered, but the models concentrate only on the network and only network transfer times are considered. Both models require in-depth knowledge of all transfers, which is useful in message passing applications, but most often not realistic for OpenMP programs, since data is transferred transparently by the cache coherence mechanisms of the hardware. [Calotoiu et al., 2013] presented a tool for automatic performance modeling, namely Extra-P. This tool allows to extrapolate the performance of an application runs with many cores based on a few previous measurements with smaller core counts. The tool focuses on MPI applications, but the techniques might be applied to OpenMP applications as well. However, such an approach will very likely lead to inaccurate predictions on hierarchical NUMA systems, as the small scale measurements on a few cores do not provide any reliable data to predict memory access performance outside of the local NUMA domain. The model presented here will use results from the kernel benchmarks to provide a more accurate performance estimation than the roofline model, while it will still be usable by non architecture experts and for OpenMP programs.

Second, data and thread placement are the most important factors in shared memory programming on NUMA systems. The current standard of OpenMP allows the placement of threads on *places* with different strategies. For the data placement no OpenMP support exists, but it is common practice to use the *first-touch* data placement policy to distribute the data across NUMA nodes in the desired way. This means, that the data needs to be accessed first on the node where it should reside in memory, this can be achieved in many cases by parallel initialization with the same access pattern that is later on used during computation. If this is not sufficient for an application, e.g. because the data access pattern changes over time, like in adaptive algorithms, data migration mechanisms exist for the Linux operating system. This work will investigate the usefulness of these mechanisms on modern architectures. Furthermore, a method will be presented to achieve *migration-on-next-touch* in user space for the Linux operating system. This approach means that memory is migrated to the thread which uses the memory the next time. The Solaris operating system offers support for this type of migration, but for Linux no support exists so far. The advantage of this method is not that it is faster than explicit migration, but that it can be applied easier if the access pattern is unknown or hard to describe for explicit migration calls.

Third, load balancing is one of the major challenges in parallel computing in general. OpenMP provides standard methods to deal with load imbalance for unbalanced computation loops, namely `dynamic` or `guided` schedules can be applied, leading to a balanced but unpredictable and unreproducible distribution and order of the loop iterations of the parallel loop. This makes it impossible to allocate data on the right NUMA nodes in advance. This work will present a NUMA-aware scheduling mechanism, providing load balancing while still maintaining data locality where possible.

Fourth, since OpenMP version 3.0, tasks provide a different way to express parallelism in OpenMP. The concept of tasks adds an additional layer of parallelism but also of complexity to OpenMP programs. An OpenMP runtime has a lot of flexibility in the way tasks are executed. This allows a lot of optimization inside the runtime system, e.g. automatic work balancing, and for the programmer it allows to express complex parallel algorithms in a simpler way in many cases, e.g. when recursions are used. The downside for a programmer is that it is hard to analyze the performance of OpenMP task parallel programs, since the runtime system makes many decisions for the programmer. This work will investigate the behavior of different OpenMP runtime systems and analyze how tasks are handled on large NUMA machines. Based on this information, different methods to use tasks by a programmer are presented and advice is given on which methods should be used when on large NUMA machines.

Fifth, this work investigates common performance problems occurring in standard OpenMP programs in general and on large NUMA architectures in particular. The ability of performance tools to detect these problems is investigated and for issues which cannot be detected easily, methods are presented to extend performance tools to detect these issues. The problematic issues are related to OpenMP tasking, since this relatively new concept is not well supported by many performance tools. To improve this, the Score-P measurement system is used as a basis which allows to profile and trace application runs of parallel programs. For traces, a trace-to-trace converter is used which adds additional information to the trace file and runs several tests on the measured data to highlight the mentioned tasking related performance issues.

1.2 NUMA Architectures

All major processor vendors building chips for high-performance computing servers build so called Non-uniform Memory Access (NUMA) architectures. In these architectures a core can access all the memory in the system, but the access time differs depending on the physical location of the memory. Typically there is a part of the memory attached to every socket of the system which can be accesses faster than memory attached to remote sockets. The advantage of NUMA architectures is that the available memory bandwidth in a system is increased with the number of sockets. The disadvantage is, that a programmer needs to take the NUMA characteristics into account to achieve good performance on such a system.

A commonly used way to program such systems is to use OpenMP, which provides a high-level pragma-based way for parallel programming, which is in many cases easier to apply than distributed-memory programming. Of course a shared-memory program is limited to a single system running a single OS instance. This is why some hardware vendors build large shared-memory machines based on specialized interconnects, like SGI builds the Altix UltraViolet systems with up to 2048 cores based on their proprietary NUMAconnect interconnect.

Besides hardware solutions to build large shared-memory machines software solutions have been investigated to allow shared memory programming on clusters. Some approaches modified the Linux kernel to achieve a Single System Image (SSI) on top of a cluster, for example MOSIX [Barak et al., 1993] or OpenMOSIX as well as Kerrighed [Morin et al., 2004] [Vallée et al., 2003]. Multi-threaded applications have not been in the focus of these projects and thus they did not turn out to be very useful for OpenMP programs. MOSIX / OpenMOSIX both only allow the migration of processes and not of threads, which makes it unsuitable for OpenMP. It has been shown that Kerrighed can run OpenMP applications employing a modified threading library [Margery et al., 2003], but this project did not progress over a proof-of-concept state allowing the execution of commercial codes. In addition, the achieved performance was not suitable for productive use. ScaleMP is the first company offering a product to couple cluster nodes into a single system. The software solution, called vSMP, allows to run a single system Linux image on an infiniband cluster. Any shared-memory program can run on these machines without recompilation.

Specifically targeting the execution of OpenMP programs on clusters has also been the focus of several research projects and even commercial products. In [Lu et al., 1998] an OpenMP implementation for the TreadMarks software has been presented, which supports a subset of the OpenMP standard. In [Sato et al., 2001] an OpenMP implementation on top of the page-based distributed shared memory (DSM) system SCASH has been presented for the Omni source-to-source translator. In this approach, all accesses to global variables are replaced by accesses into the DSM and all shared data is controlled by the DSM. Although the full OpenMP specification is implemented, support for the C++ programming language is missing. Intel's ClusterOpenMP [Hoeflinger, 2006] has been the only commercial approach for OpenMP on clusters with full support for OpenMP 2.5. An examination revealed major shortcomings in the applicability and the memory management [Terboven et al., 2008b] for real-world applications and meanwhile Intel ceased the development of this product.

1.3 State-of-the-art in NUMA Programming

NUMA-aware programming in OpenMP has been investigated for some time. One of the major issues which has been tackled so for is the placement of threads on the architecture. For this problem OpenMP offers support since version 3.1, which has been extended in version 4.0. Details are described in this work in chapter 3. Also for memory placement same state-of-the-art practices have evolved, specifically the use of the first-touch memory placement policy of the OS to distribute data over the system by a parallel data initialization. In OpenMP, no support for data placement or migration exists at all. For the OpenMP tasking feature or for dynamic schedules, which offer more flexibility in terms of work balancing, no NUMA support exists at all.

Besides the functionality to program in a NUMA-aware fashion, support in performance analysis tools is also mandatory to detect and avoid certain issues. Here, some tools exist which give some information on NUMA accesses, but they are typically hard to use, since they are based on hardware counters which are extremely hardware dependent and change with every new processor generation.

In practice, even if many large NUMA systems exist, they are mostly partitioned by a batch system to run many batch jobs in parallel and only a very few user codes scale very well over a complete system. Techniques, as they will be presented in this work, are developed to optimize more user codes for large NUMA systems to make full use of these machines for a higher number of user codes.

1.4 Experiment Hardware

This work contains results carried out on the following architectures.

Westmere

This system is a Bullx B500 blade equipped with two Intel Xeon X5675 ("Westmere") processors with 6-cores each, a clock rate of 3.07 GHz and 24 GB of main memory. The Westmere system represents a standard 2-socket node which is the most commonly used node type in current clusters.

SandyBridge

The SandyBridge system is a 4-socket DELL M820 system equipped with Intel Xeon E5-4620 ("SandyBridge") processors running at a clock rate of 2.20 GHz. The system contains 256 GB of main memory.

HP ProLiant

The HP ProLiant DL980 G7 server used for our experiments is a single server equipped with eight Intel Xeon X6550 processors with 8 cores each. All processors are clocked at 2 GHz and connected to each other through the Intel Quick Path interconnect. Every processor contains a memory controller attached to 32 GB of main memory, making this server a ccNUMA machine with a total of 64 cores and 256 GB of memory.

SGI Altix UltraViolet

The SGI Altix UV system consists of several two socket boards, each equipped with two Intel Xeon E7- 4870 10-core processors clocked at 2.4 GHz. All of these boards are connected with SGIs NUMALink interconnect into a single shared memory machine. Since on one board the cache-coherence is established directly over the QPI, whereas the NUMALink network is needed for different boards, this machine is a hierarchical NUMA machine, with different cache-coherency mechanisms on different hierarchical levels. The machine used in our tests has 2080 cores and about 2 TB of main memory. All of our tests were done on up to 16 processors during batch operation of the system. For a better comparison with the 8-core processors used in the other systems, all tests were done using only eight of the ten available cores on each socket, i.e. using up to 128 cores.

BCS

The BCS system consists of four bullx s6010 boards. Each board is equipped with four Intel Xeon X7550 processors with 8 cores each and 64 GB of main memory. The Intel Quick Path Interconnect combines the four sockets to a single system and the Bull Coherence Switch (BCS) technology is used to extents the QPI to combine four of those boards into one SMP machine with 128 cores and 256 GB of main memory. So, this system is also a hierarchical NUMA system.

ScaleMP

The ScaleMP machine investigated here consists of 16 boards, each equipped with four Intel Xeon X7550 8-core processors clocked at 2 GHz and 256 GB of main memory. The boards are connected via a 4x QDR InfiniBand network, where every board is connected via two host channel adapters. Thus, from a hardware point of view this is an ordinary (small) cluster. The innovative part of the machine is the vSMP software, which runs below the operating system and creates a single system image on top of the described hardware. The virtualization layer of the processors and the InfiniBand network is used by the vSMP software to create cache-coherency on a per page basis and to allow remote memory access between all the boards. A partition of the main memory is reserved by the vSMP software to run different caching and prefetching mechanisms automatically in the background, as well as a page-based memory migration mechanism. These mechanisms do not only move pages on access, they can also adjust the home node of memory pages if pages are frequently used on a remote node. This is a notable difference to standard x86-based non-uniform memory architectures (NUMA), like the Altix or BCS machine, where page migration needs to be done by the user, if possible at all. From a user point of view the machine looks like a single Linux machine with 512 cores and about 3.7 TB of main memory. About 300 GB of the available memory are used by the vSMP software internally for caching. Linux sees 64 NUMA nodes, each containing about 64 GB of main memory. Due to the End Users License Agreements of ScaleMP, no absolute performance results for the ScaleMP machine may be presented here, but I will show relative improvements for different tuning steps on such a system.

Intel Xeon Phi

The Intel Xeon Phi coprocessor is not a NUMA machine, it is only a single chip with one memory chunk attached. The system is used for comparison to demonstrate the behavior of a non-NUMA system for the tests carried out. The Intel Xeon Phi coprocessor is based on the concepts of the Intel Architecture (IA) and provides a shared-memory many-core CPU that is packed on a PCI Express extension card. The version used here has 60 cores clocked at 1.053 GHz and offers full cache coherency across all cores with 8 GB of GDDR5 memory. A ring network connects all cores with each other and with memory and I/O devices. Every core supports 4-way Hyperthreading, which allows the system to run up to 240 threads in parallel. The comparably small amount of main memory is attached to the Xeon Phi Chip as one NUMA node. Thus, the system is a 240-way parallel system with a uniform memory architecture, which is another difference to the other machines with are large NUMA systems.

The Xeon Phi card used in this work was plugged into a host system with two Intel Xeon E5 processors. For all of our experiments we used the host system only to cross-compile the executables, which were copied and executed stand-alone on the Xeon Phi. This procedure gives us insight in the performance attributes of the chip, independent from the programming model used. Of course, comparing one extension card with complete systems is an uneven comparison, but for sure we will see standalone systems with hundreds of cores in the near future and the Phi might give evidence on the behavior of such systems.

System	#sockets	#cores	clock rate	memory	2nd level interconnect
Xeon Phi	1	60	1.0 GHz	8 GB	-
Westmere	2	12	3.0 GHz	24 GB	-
SandyBridge	4	32	2.2 GHz	256 GB	-
HP	8	64	2.0 GHz	256 GB	-
BCS	16	128	2.0 GHz	256 GB	Bull Coherence Switch
ScaleMP	64	512	2.0 GHz	3789 GB	Infiniband / vSMP
Altix	208	2080	2.4 GHz	2048 GB	NUMALink

Table 1.1: Overview of the machines used during this work.

2 A Benchmark-Guided Characterization of Large Shared-Memory Machines

As already mentioned before, a part of this work is to provide insights in the different types of shared-memory machines, since machine attributes have a high influence on the program's performance. In this chapter, I will show methods to investigate relevant attributes of large SMP machines with standard kernel benchmarks and self-implemented benchmarks. All tests are done on the set of shared memory machines presented in section 1.4, but the methods can easily be applied to any other machine. Since machines with uniform memory access (UMA), non uniform memory access (NUMA) and with software or hardware based multiple NUMA levels are used in these experiment, general differences between these machine types will be highlighted by these tests as well. The benchmark results are later on in this work used as basis for a machine description which can be used for OpenMP programs to improve the affinity support in section 3.2, and to formalize a performance model for OpenMP programs on NUMA systems, overcoming some of the limitations of well established performance models in section 6.2.

Benchmarks to investigate the performance of different aspects of an architecture have been developed in many different studies. Standard benchmarks to investigate the memory performance of a system are the Stream benchmark [McCalpin, 1995] and the LMBench benchmark suite [McVoy and Staelin, 1996]. For OpenMP programs the EPCC microbenchmarks [Bull, 1999], [Bull and O'Neill, 2001], [Bull et al., 2012] can be used to measure the overhead of OpenMP constructs. Furthermore, benchmark suites exist which can be used to compare the performance of architectures and OpenMP implementations for application-like kernels, e.g. the SPEC OMP benchmark suite [Aslot et al., 2001], [Müller et al., 2012] or the NAS parallel benchmarks [Bailey et al., 1991]. For OpenMP programs using the new tasking paradigm the Barcelona OpenMP Task Suite (BOTS) [Duran et al., 2009] can be used.

All of these benchmarks come with published performance results on different architectures, but no study has been made to compare the performance of large SMP machines so far. Furthermore, I want to investigate different aspects of the machine and not improve the benchmarks themselves in this work. So, I used some of the benchmarks mentioned, like the EPCC microbenchmark or developed own tests for specific characteristics of the test machines. The rest of this chapter will show that hardware awareness is important on large NUMA systems and which parts of an application are mostly influenced by these hardware characteristics. Furthermore, it shows differences and similarities of the machines and OpenMP runtime systems on these machines.

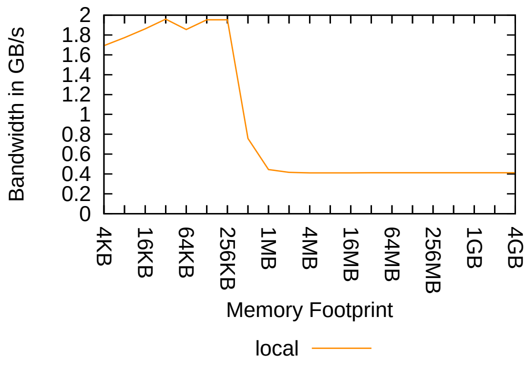
Parts of the results and methods presented in this chapter have been published before. In [Schmidl et al., 2010a], [Schmidl et al., 2010b] and [Berr et al., 2012] kernel tests on a ScaleMP system similar to results presented here have been reported. Furthermore, the performance of Xeon Phi systems in comparison to hierarchical NUMA systems like the BCS machine were presented in [Cramer et al., 2012] and [Schmidl et al., 2013b]. Finally, a study comparing different NUMA systems as they are used here was published at IWOMP 2013 in [Schmidl et al., 2013a].

2.1 Memory Performance

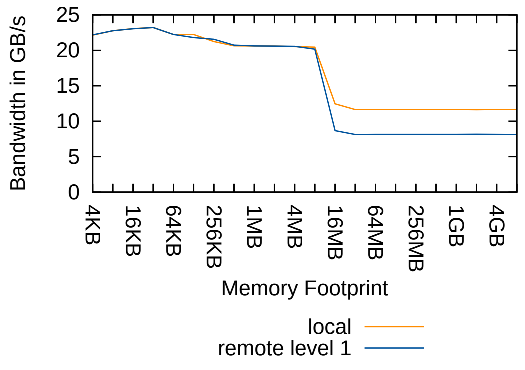
One of the big advantages of NUMA systems compared to UMA machines is the fact that every processor has its own memory and that this memory is directly connected to the processor. Thus, the available amount of main memory and the available bandwidth increases with the number of sockets in such a system. Since many scientific applications are memory bound, a high overall memory bandwidth is important for many applications. Therefore, I will first investigate the main memory bandwidth of the target platforms of this work.

2.1.1 Serial Memory Performance

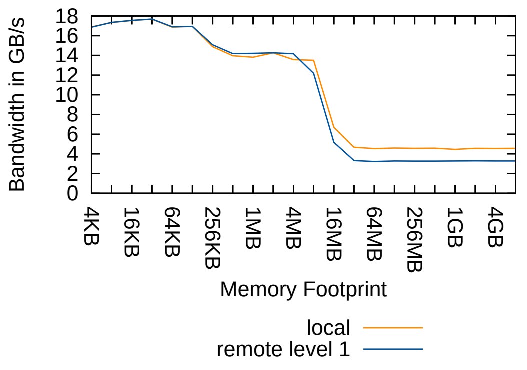
The memory bandwidth and latency on a NUMA system is highly influenced by the physical location of the memory in relation to the processor core accessing the memory. As described in section 1.4, the investigated machines have a uniform memory access (Xeon Phi), a non uniform memory access with one level (Westmere, HP) or with two levels (Altix, BCS, ScaleMP). First, I implemented a serial memory bandwidth test which reads or writes an data array 1000 times and computes the reached memory bandwidth. The test can be used to measure the reachable bandwidth for a single thread on different systems. The command line tool `numactl`, provided by the Linux operating system, can be used to set the core and also the memory node used for the benchmark. For example, the bandwidth a thread running on core 0 can reach when data on the fourth NUMA node is accessed can be measured by the following command:



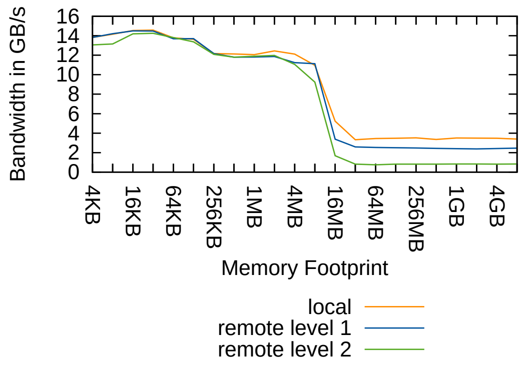
(a) Xeon Phi



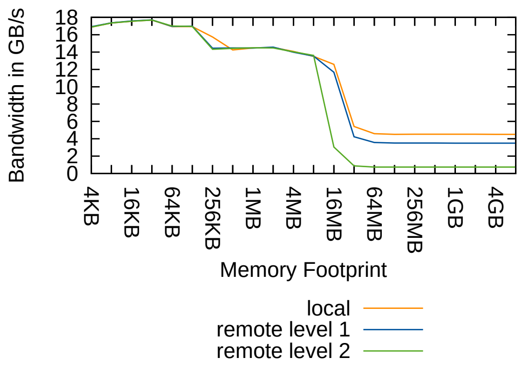
(b) Westmere



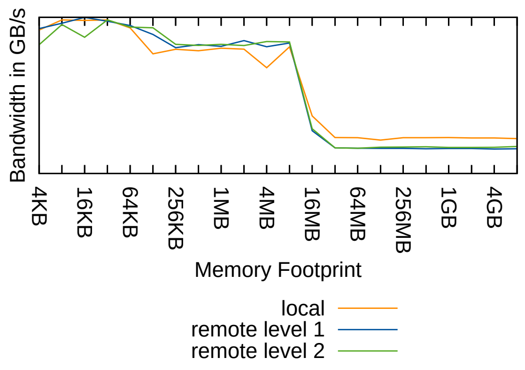
(c) HP



(d) BCS



(e) Altix



(f) ScaleMP

Figure 2.1: Read Bandwidth in GB/s reached for different NUMA levels measured on the Xeon Phi, Westmere, HP, BCS, Altix and ScaleMP system for different memory footprints.

```
numactl --membind=4 --physcpubind=0 ./bandwidth_test.exe
```

In this way I measured the reachable bandwidth for three different scenarios, if applicable on the specific machine:

(i) **local:** Here the core and the memory belong to the same socket in the system.

(ii) **remote level 1:** Here the memory of a remote socket on the same physical board is used. To access the data the QPI can be used by the hardware.

(iii) **remote level 2:** Here the memory of a remote socket on a different physical board is used. To access the data the second level interconnect of the system (BCS, Numalink or vSMP software + infiniband) is needed.

The results of my tests are shown in figure 2.1 for the read bandwidth. Results for the write bandwidth look similar and are therefore omitted. On all machines, a typical cache behavior can be seen. For small data sizes the access is fast, since the data fits into the caches and is not loaded from memory for all but the first of the 1000 accesses. When the size of the last level cache is exceeded, the performance drops down to the memory bandwidth and stays pretty much constant at this level in all measured cases. The differences in the local performance of the systems is due to different processors, clock speed and memory dims. But it can be observed, that the bandwidth for a remote level 1 access is about 25% to 30% slower than a local access. A remote level 2 access in contrast is about 75% slower on the BCS system and 85% on the Altix machine. On the ScaleMP machine no difference can be observed between remote level 1 and 2, because the software caches all accesses internally and serves the requests from the board local software cache.

Overall, these tests show that the memory bandwidth highly depends of the location of the data in the NUMA system, as expected. Moreover, the tests also show, that on hierarchical NUMA machines, the board local accesses are not penalized by the fact that the systems have a second level interconnect. So, for a programmer it is important to realize that remote memory is not equal on these hierarchical machines. The different levels should be taken into account.

2.1.2 Parallel Memory Bandwidth

Since parallel applications in high-performance computing typically use all cores of a system at a time the total bandwidth of a system is more important than the serial bandwidth investigated so far. Thus, I modified the benchmark used in section 2.1.1 to work with several threads on an array and measured the read and write bandwidth. All threads are working on different pieces of the array, so there are no conflicts or synchronization constructs among threads. Threads were placed in a `close` binding using the `OMP_PROC_BIND` environment variable provided by OpenMP for thread placement. The `close` strategy instructs the runtime to fill up all cores and hyperthreads of one socket before the next socket is used.

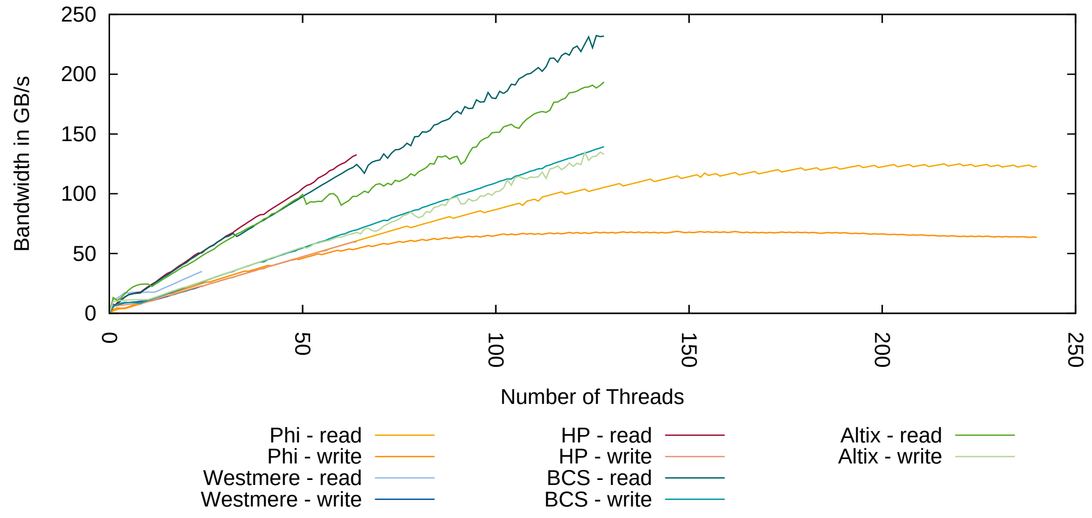


Figure 2.2: Parallel read and write bandwidth on the Westmere, HP, Altix, BCS and Xeon Phi systems for an increasing number of threads.

Figure 2.2 shows the read and write bandwidth for an increasing number of threads on the different platforms for a memory footprint of 16 MB per thread. (Results for the ScaleMP machine are omitted because ScaleMP's EULA does not allow to publish absolute performance results.) On the Intel Xeon Phi machine the maximum memory bandwidth of about 130 GB/s for reading and 60 GB/s for writing can be achieved with about 120 threads. Beyond this, the bandwidth stagnates. This behavior is typical for UMA systems like the Xeon Phi. The bandwidth rises at the beginning, until enough threads are started to consume the total available bandwidth of the memory controller. With more threads the bandwidth does not raise any further, which leads to the flattened curve in figure 2.2.

On the NUMA systems the bandwidth rises with the number of sockets used and does not stagnate at all. Of course, this is due to the increasing number of memory controllers and memory banks for a larger number of threads. However, it also indicates that the cache coherence on all systems for mostly local memory accesses is low and does not prohibit scaling on such systems.

On all systems the read bandwidth is higher than the write bandwidth, since a write requires a prior read of a cache line which leads to half the bandwidth of a write operation compared to a read operation.

2.1.3 memory_go_around

The parallel bandwidth investigated so far is the optimal bandwidth which can be reached on a system. All threads were working on distinct pieces of a shared array which was distributed across the NUMA nodes of the system. Thereby it can be achieved that nearly all accesses to data are served by the local memory of a processor. This is a typical case in dense linear algebra, where matrices and vectors can easily be split between threads, or in embarrassingly parallel algorithms were often all threads work on their own data. However, there also exist many algorithms where some data sharing is required. Here, a certain amount of remote accesses cannot be avoided. To investigate the drop in the available memory bandwidth with remote accesses, a different benchmark is used. The unpublished benchmark was originally designed by Dieter an Mey and only slightly modified for this work to contain automatic thread placement. The benchmark increases the number of remote accesses from step to step and measures the effect on the memory bandwidth reached.

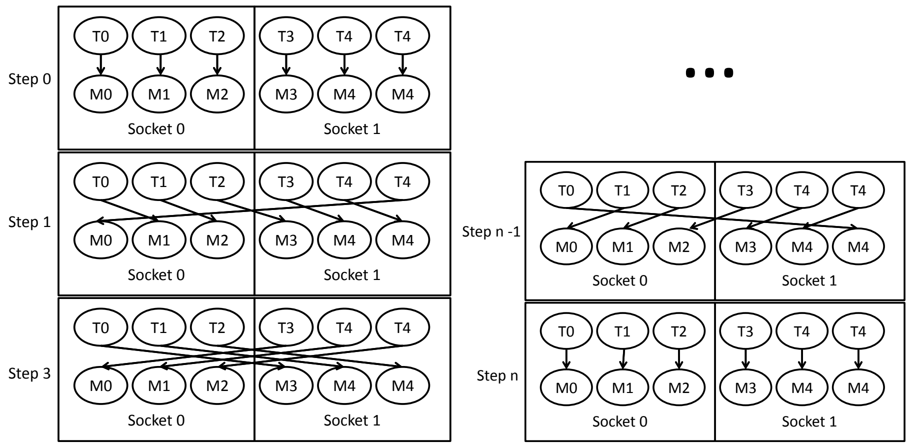


Figure 2.3: The memory_go_around benchmark works in n+1 steps. In the first step the memory of the right neighbour is used to measure the bandwidth, in the next step the memory of the next neighbour and so on. This increases the distance between thread and memory in every step, until half of the steps are done, then the distance decreases until it reaches zero in the last step.

The benchmark is called `memory_go_around` and it was first presented in [Schmidl et al., 2013a]. In the first step, every thread initializes its chunk of data and measures the memory bandwidth to access this chunk. This step is similar to the parallel bandwidth test used in section 2.1.2. In the next step, all threads work on the data of their right neighbor, so thread t works on memory initialized by thread $(t + 1)$ mod (n), where n is the total number of threads used, as exemplified in figure 2.3. Threads are placed in a way that neighboring threads run as close to each other as possible. So, I first fill up all hyperthreads of a core, than all cores of a processor and then all processors of a board and all boards in a system. So, there is a high chance that neighboring threads run on the same NUMA node or board. Only the one thread running on the last core of a socket or board will work on data on the next socket or board. In the next steps the distance is increased, since all threads t work on the data of thread $(t+s)\ mod\ (n)$ in step s. Hence, the number of remote memory accesses rises for the first $n/2$ steps. Then the number shrinks again until in step $n-1$ every thread works on the memory of the left neighbor and in step n again on its own local memory.

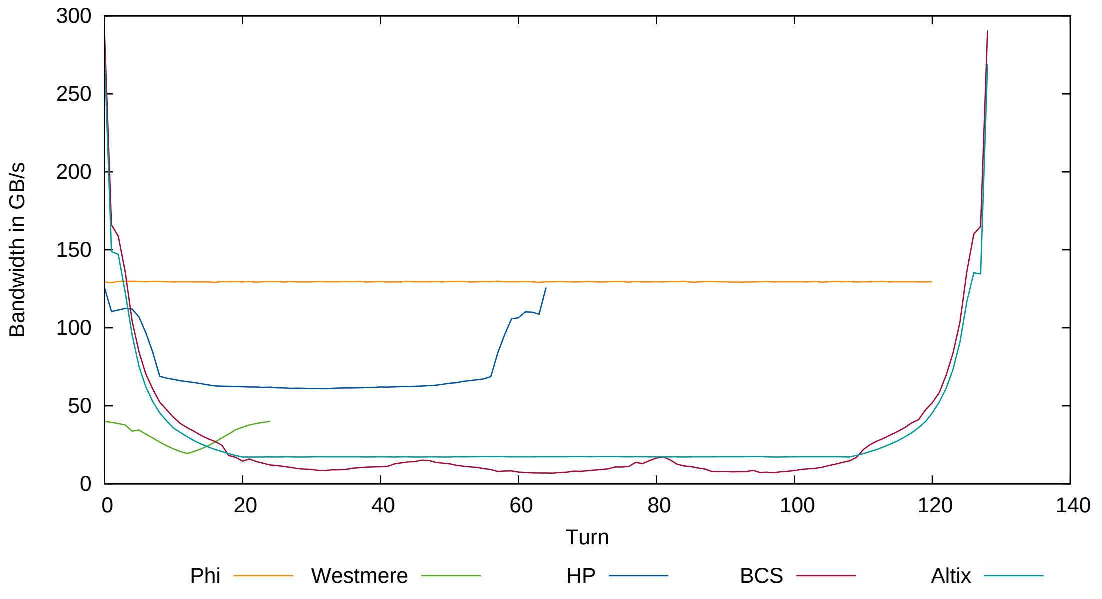


Figure 2.4: Bandwidth measured with the memory_go_around benchmark for n+1 steps with n threads on the Westmere, HP, Altix, BCS, and Xeon Phi system.

Figure 2.4 shows the result for 24 Threads on the Westmere, 64 Threads on the HP, 120 threads on the Xeon Phi and for 128 threads on the Altix and BCS machine. The Xeon Phi machine shows the typical behavior of a UMA system. Since only one memory location exists in the system, it does not matter at all, which thread initialized the data and a constant bandwidth of about 130 GB/s is reached in all turns.

On the other machines the bandwidth declines for the first half of the steps and then rises up again. Of course this is related to the increasing number of remote accesses and the increasing distance between these accesses. On the 2-socket Westmere system, which represents the most typical architecture in HPC, the performance drops rather slow in the first 12 turns until it reaches the minimum of about 50% of the starting performance. On the HP system, the performance drops down from about 120 to 60 GB/s, so also here about 50% of the performance are still reached, even when all accesses are remote accesses across the whole system. But the curve drops faster than on the Westmere system.

On the Altix and BCS machines the drop is from about 250 to 18 or 8 GB/s which is 6% or 3% of the available maximum bandwidth. Furthermore, the performance drops down very fast starting at turn 1. The performance penalty is much more severe on the hierarchical systems and already reached for a smaller number of remote accesses. This makes it much more important to minimize remote accesses compared to the standard 2-socket Westmere machine, for example. However, if an application does not require a lot of data sharing between the threads, proper data placement or migration can avoid these problems, as I will show in chapter 3.

2.1.4 Data Management

Besides the performance of memory accesses on a system, which I investigated so far, data management can be a source of overhead. Especially on large shared-memory machines with lots of threads, managing a shared address space and mapping virtual to physical addresses is a challenging task which might require locking by the OS and thus can hinder the scalability of a program. In [Schmidl et al., 2010b] we have shown that allocation and initialization of an array can introduce significant overhead on a hierarchical system. Here, I want to investigate the allocation and initialization process separately on the target platforms to get more insights in the source of overhead.

Data Allocation

Since all threads use a shared address space, synchronization is required when multiple threads allocate memory simultaneously, to ensure that the virtual addresses returned by the allocation process are disjoint. The overhead of locking depends on many factors, like the number of threads, the OS, the frequency in which allocation calls occur and so on. I wrote a simple benchmark which illustrates the behavior in a worst-case scenario. In the benchmark, all threads are just allocating memory all the time in a parallel region. This is unrealistic for a real program, but it gives an impression of the scaling behavior of memory allocation on different systems. Furthermore, the benchmark was run with standard `malloc` calls and with calls to Intel's `kmp_malloc` function. `kmp_malloc` is an optimized memory allocation function provided by Intel for improved performance in multithreaded programs.

Table 2.1 shows the overhead for memory allocation on the different test systems. First of all, the allocation on the Xeon Phi system is comparably slow, which

	malloc				
#Threads	PHI	Westmere	HP	BCS	Altix
1	15.27	2.12	3.61	4.30	3.46
24		357			
32/30	11023		6075	4902	5146
64/60	19807		12470	14007	10473
128/120	26603			29552	21030
	kmp_malloc				
1	981	2.06	3.04	3.63	2.60
24		16.4			
32/30	37211		411	530	558
64/60	88717		1476	2786	2646
128/120	175641			12958	11742

Table 2.1: Overhead in microseconds for a malloc call allocating 1MB of memory. All threads constantly allocate and free memory. Results are shown for 30, 60 and 120 threads on the Xeon Phi and 32, 64 and 128 on the HP, BCS and Altix.

might be due to the low clock frequency. Furthermore, `kmp_malloc` delivers worse performance for the allocation itself than standard `malloc` calls here. Maybe Intel did not optimize this functionality for the Xeon Phi in version 14.0 of the Intel compiler, which I used for the tests. On all other systems it can be observed, that the allocation time rises with the number of threads. This is not surprising, since more conflicts occur. On the 2-socket Westmere system a slowdown of about 170 can be observed for 24 threads, on the other systems the slowdown is up to nearly 4 orders of magnitude for a large number of threads. In a program with a lot of memory allocation calls, like many C++ programs which tend to call a lot of constructors and destructors during execution, this can be a limiting factor for scalability. The calls to `kmp_malloc` are up to an order of magnitude faster for a large number of threads. Overall, it makes sense to use these calls if possible to reach better scalability for OpenMP programs on large systems. However, since it requires to change all malloc calls in a program and in libraries used, this advice is often hard to fulfill in practice. Furthermore, the `kmp_malloc` function is Intel specific.

Data Initialization

After the allocation, the OS needs to assign physical addresses to the memory pages when they are first used. Typically, this is done during the initialization of the data, since a read access does not make sense before initialization. I measured the bandwidth when initializing a 2 GB array with a different number of threads.

Table 2.2 shows the bandwidth reached. In brackets the speedup compared to a single thread on the same system is shown. It can be observed that the initialization scales well on all platforms. Although the address tables, necessary for the mapping of virtual to physical addresses, need to be maintained for the complete process, this seems to scale well over the complete machine. The less than linear speedup is due to the limited memory bandwidth in the system. Overall, the overhead observed in [Schmidl et al., 2010b] seems to be caused by the data allocation and the initialization seems to be no big problem on large systems, since the overhead scales well over the complete machine.

#Threads	PHI	Westmere	HP	BCS	Altix
1	0.67 (1.0)	2.2 (1.0)	1.42 (1.0)	1.31 (1.0)	1.38 (1.0)
24		17.2 (7.8)			
32/30	17.73 (26.5)		16.70 (11.8)	18.36 (14.0)	18.30 (13.2)
64/60	23.12 (34.6)		32.93 (23.2)	34.24 (26.1)	33.70 (24.4)
128/120	19.32 (28.9)			67.98 (51.9)	72.10 (52.2)

Table 2.2: Bandwidth reached when initializing an 2 GB array with first-touch memory placement. In brackets the speedup compared to a single thread on the same machine is shown. Results are shown for 30, 60 and 120 threads on the Xeon Phi and 32, 64 and 128 on the HP, BCS and Altix systems.

2.2 OpenMP Runtime Issues

Overhead through remote accesses and memory management are one very important source of overhead on large shared memory machines, but not the only one. Another important source of overhead is the overhead of OpenMP constructs used in an application. Of course the overhead of the OpenMP runtime system increases on a larger machine, since more threads have to be managed. However, how much it increases depends on many factors, like the machine, the OpenMP construct and the implementation inside of the runtime.

2.2.1 EPCC Benchmarks

The EPCC microbenchmarks, developed by Bull at al. [Bull, 1999], can be used to measure the overhead introduced by OpenMP constructs. Within this suite, there exist benchmarks to measure the overhead of synchronization, the overhead of data allocation and copies in `private`, `firstprivate` and other constructs as well as the overhead of schedule clauses for worksharing constructs.

Here, I focus on the overhead of synchronization constructs on large NUMA systems. Therefore, I used the `syncbench` benchmark to measure the overhead of OpenMP constructs with the most widely used compilers in high-performance computing, i.e. the Intel, GCC, PGI, Oracle Solaris Studio and Clang compilers. The tests were performed on the BCS machine as a representative for a hierarchical architecture.

#Threads	Intel (14.0)	GCC (4.8)	PGI (15.1)	Studio (12.4)	Clang (3.7)
1	0.34	0.21	0.18	0.22	0.31
2	0.81	0.87	0.61	19.22	1.22
4	0.98	1.34	0.85	53.28	1.56
8	1.40	2.22	1.30	174.0	2.33
16	4.54	14.41	3.92	399.7	8.64
32	5.30	48.45	5.40	3991	9.45
64	13.05	238.6	13.34	9495	24.69
128	15.96	710.1	24.74	19671	27.85

Table 2.3: Overhead in microseconds of the `parallel` construct on the BCS machine, measured with the EPCC `syncbench` benchmark for different compilers.

Table 2.3 shows the results of the overhead measurements for a `parallel` construct with all investigated compilers. Thread binding was activated in the same way as before, so that threads are placed close to each other. The placement of threads across the system takes place in the OpenMP runtime when threads are created, so the overhead is included in the measurements.

It can be observed, that the overhead significantly differs between the different runtime implementations. For all compilers, the overhead increases with the number of threads, which is what a user would expect, since creating and placing 128 threads is more work than creating 4 threads. For the Intel, PGI and Clang Compiler the overhead for up to 128 threads is still around 15 - 30 microseconds, whereas the overhead for the GCC and Studio compiler is much higher: 710 microseconds for the GCC and 19671 for the Studio compiler. If many parallel regions are used in a program, a user should clearly use one of the faster compilers, since the overhead is much lower. But even for the Intel compiler the overhead to create and bind threads on the BCS system is still about 50 times higher than in the serial case. Avoiding the creation of too many parallel regions in general is therefore still necessary on larger machines to keep the overhead low and possibly achieve scalability.

For different OpenMP constructs and numbers of threads the overhead differs. The Intel implementation delivers the lowest overhead in my experiments, therefore it is used to investigate different constructs on the BCS machine. Figure 2.5

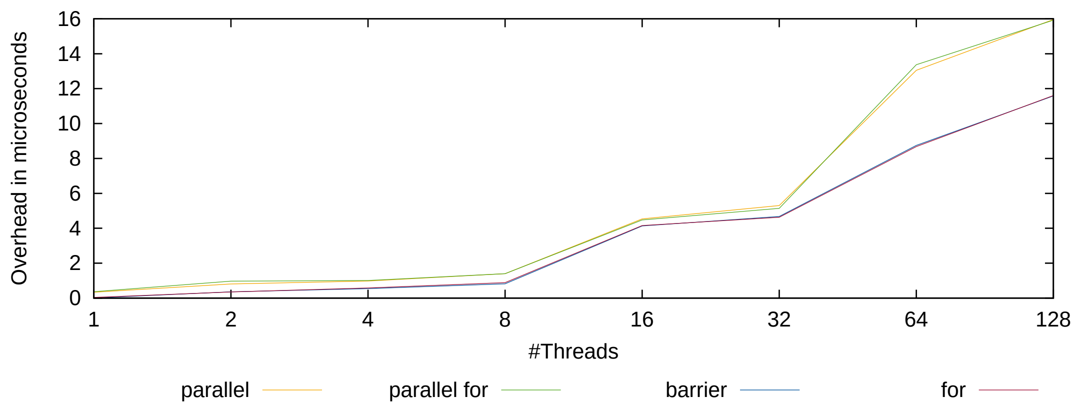


Figure 2.5: Overhead for different OpenMP constructs measured with the EPCC microbenchmarks on the BCS machine and the Intel Compiler 14.0.

shows the overhead for a `parallel`, `parallel for`, `barrier` and `for` construct for an increasing number of threads. It can be observed that the overhead of a `for` and a `barrier` construct are nearly identical as well as the overhead of a `parallel` and a `parallel for` construct. The reason is that the `for` construct includes a barrier. In addition, it only includes an index calculation which needs no synchronization and needs only a few floating point operations which do not take a lot of CPU cycles. Since the `parallel` construct also includes a barrier, the difference to a `parallel for` construct is also the index calculation which is negligible.

Furthermore, the influence of the hardware topology can be observed for all constructs. The BCS machine has two levels of NUMA. The limit of a socket is reached with 8 threads and one board is fully occupied with 32 threads. Correspondingly, figure 2.5 shows two steps in the overhead curves for all constructs.

2.2.2 Nested Parallelism

For a single level of parallelism in an application the overhead can be influenced only by the number of threads and the placement. But, in most cases a thread per core is started and then the placement is easy. Placing the threads consecutive on all cores on a socket, board and machine as it was done for the tests above is the best option to minimize the distance between neighboring threads.

When nested parallel regions are used to express multiple levels of parallelism in an application, the situation gets more complicated. Basically, on every level the user has to choose if the threads created for this level should be placed together on the same socket or be distributed over the complete machine to fill up the sockets on a nested inner level. The overhead for nested parallel regions can differ from the standard case with one level, since more bookkeeping by the OpenMP runtime is needed. To measure the overhead of nested constructs I modified the EPCC benchmarks as described in [Schmidl et al., 2010a] by surrounding the actual tests with an additional outer parallel region. The benchmarks measure the overhead of all constructs on an inner nested level.

#Threads	parallel	parallel for	barrier	for
off	2066	2184	17.6	24.0
on	2819	2791	8.2	9.1
compact	1502	1551	188.6	205.8
distributed	2787	2795	8.5	9.9

Table 2.4: Overhead in microseconds for OpenMP constructs in nested parallel regions measured on the BCS machine and the Intel Compiler 14.0 and 128 threads.

I measured on the BCS system with four outer threads, where each thread spawns an inner team of 32 threads again using the Intel Compiler. So, the tests use all 128 cores of the system. For the placement two strategies are possible, starting one of the outer threads per board and then filling up the boards with 32 inner threads in the nested inner parallel region or placing the outer threads on the first four cores of the system and then starting the other threads on the rest of the systems, depending on the creation time inside of the runtime. The first strategy is called distributed and the second one compact in table 2.4. Here, the overhead is shown for the Intel compiler on the BCS system for these two strategies and for thread binding turned off and on by setting the `OMP_PROC_BIND` variable to `on` and `off`.

First, it can be observed, that the overhead of nested parallel regions is about two orders of magnitude higher than for the outer regions measured before (see Fig. 2.5). The reason is that threads are managed in a thread pool inside of the OpenMP runtime. Since the thread pool is shared, creating four parallel regions simultaneously creates synchronization overhead inside of the runtime system. Second, it can be observed that the binding strategies influence the overhead. Turning thread binding off creates the teams faster than turning it on, but this is because the binding is just omitted. Threads are just running on a single board where they are created in this case. For computation tasks this would waste resources or the OS scheduler would migrate the threads later on and create overhead then. Turning binding on seems to behave like the distributed case, so this might be the default behavior of the runtime system. In the distributed case, the synchronization of an inner team in the `barrier` or `for` construct is much faster. This is because all threads are running on the same board of the system and therefore the synchronization does not involve communication across boards. In the close case, the synchronization takes much longer, but the parallel regions are executed faster. This is because the thread pool is located on the first board and the management of threads can be done by the master threads on this board when all threads are running there. The placement of threads across boards is then done in parallel. Overall the user needs to find a compromise between thread generation and synchronization overhead. But, when data is shared between the threads of an inner team, the distributed binding has the additional advantage that the data is local for all threads of the inner team, whereas the threads are distributed in an unpredictable fashion in the close case.

2.3 Distance Matrices

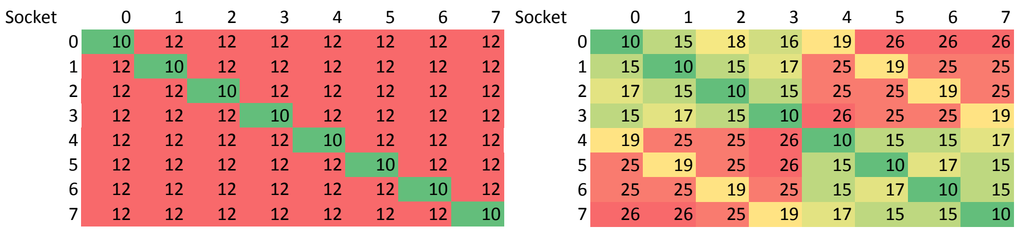
It was been shown so far, that the different levels of NUMA distance in a hierarchical machine introduce different overhead with respect to remote accesses and runtime overhead. For a programmer it is therefore sometimes necessary to have full control over the thread and data placement in a system. Techniques to optimize performance for large NUMA systems will be discussed later in this work, but obviously, a necessary requirement is a representation of the system which can be queried by the programmer to get an overview of the system.

A portable hardware description is addressed by several libraries where the `Portable Hardware Locality (hwloc)` [Broquedis et al., 2010] software package is the most prominent one and most widely used one, since it is provided with the OpenMPI software package. In hwloc the hardware is described in a hierarchical tree structure with different levels for System, NUMA nodes, Sockets, several levels for cores with shared caches and hyperthreads. In [Schmidl et al., 2010b] I used a similar tree-like representation for thread binding on large NUMA systems, but it turned out that such a structure is not sufficient in many cases. The disadvantage is that the distance on a specific level cannot be expressed in a tree. All sockets in a system, for example, are on one level, without a way to distinguish sockets with different distances. For a standard 2-socket server this is appropriate, but on large hierarchical systems this is not sufficient to describe the hardware.

In this work, I develop a machine model based on the concept of a distance matrix and discuss how to exploit it from within applications and the OpenMP runtime system. In chapter 3, I will further discuss how it can be used as basis to obtain a suitable place list for thread binding in OpenMP 4.0. I believe that this model delivers functionality still missing in OpenMP today, where the machine description of a NUMA system is left to the programmer. An automatically generated description would eliminate the need for the user to handle details of the NUMA distances in the system he is currently using and it would allow to program in a more portable fashion for different systems.

2.3.1 System Locality Distance Information Table

Describing distances in a NUMA system has been addressed before. The Advanced Configuration and Power Interface Specification (ACPI) [Hewlett-Packard et al., 2011] of the system BIOS provides the System Locality Distance Information Table (SLIT) listing the distance between hardware resources on different NUMA nodes. However, it is undefined how this table has to be filled, resulting in implementation-defined behavior.



Socket	0	1	2	3	4	5	6	7
0	10	12	12	12	12	12	12	12
1	12	10	12	12	12	12	12	12
2	12	12	10	12	12	12	12	12
3	12	12	12	10	12	12	12	12
4	12	12	12	12	10	12	12	12
5	12	12	12	12	12	10	12	12
6	12	12	12	12	12	12	10	12
7	12	12	12	12	12	12	12	10

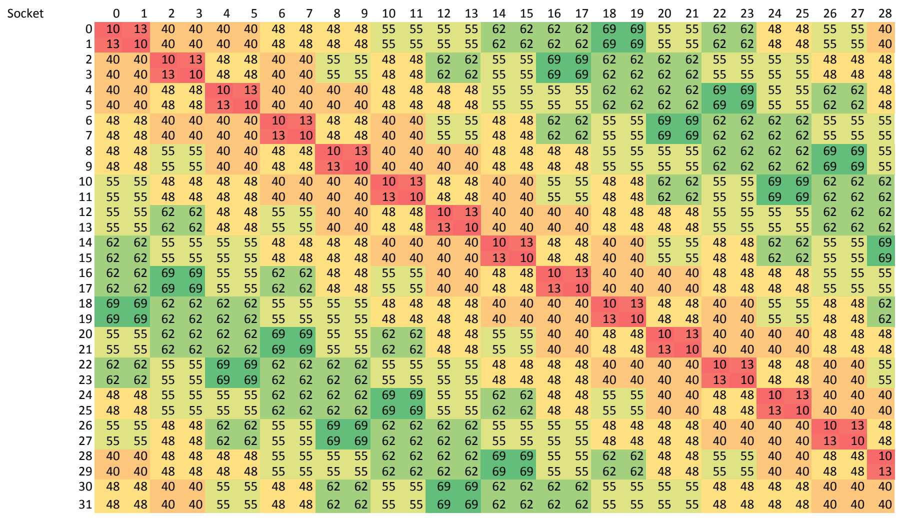
Socket	0	1	2	3	4	5	6	7
0	10	15	18	16	19	26	26	26
1	15	10	15	17	25	19	25	25
2	17	15	10	15	25	25	19	25
3	15	17	15	10	26	25	25	19
4	19	25	25	26	10	15	15	17
5	25	19	25	26	15	10	17	15
6	25	25	19	25	15	17	10	15
7	26	26	25	19	17	15	15	10

Figure 2.6: (a)Distance Information retrieved from Linux and (b) measured distance with our distance matrix benchmark on a 8-Socket Intel Nehalem-EX machine.

On the Linux operating system, this table can be queried either via the `/sys` virtual file system, or it can be retrieved by tools like `numactl`. The `libnuma` library provides a suitable API, the function `numa_distance(int node1, int node2)` returns a number to describe the distance between two NUMA nodes (node1 and node2).

On a Fujitsu system with 8 Intel Nehalem processors, the SLIT showed the distance information shown in figure 2.6 on the left side. As usual, the diagonal values in the table are normalized to 10 and all other values are 12 which indicates that all distances in the system are equal and about 20% slower than local accesses. On the right side of figure 2.6, I show the distance matrix proposed in this work, which is based on measured bandwidth values between the sockets. Details are given below, but obviously the measurements show that the distances are not equal between all sockets. It can be observed that the system is internally hierarchical where the first 4 sockets build a closer block as well as the last 4 sockets. Between these blocks the distance seems to be slightly slower. This demonstrates that the SLIT is not reliable on all systems and a more reliable approach offered by OpenMP is likely to be a better alternative for programmers.

However, the SLIT is not always as problematic as on the 8-socket system. Figure 2.7 shows the SLIT on an SGI Altix UV system. It can be observed, that the matrix is much more detailed than on the Fujitsu system and the distances are expressed more realistically. For a programmer such a matrix might be useful,



Socket	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28
0	10	13	40	40	40	40	48	48	48	48	55	55	55	55	62	62	62	62	69	69	55	55	62	62	48	48	55	55	40
1	13	10	40	40	40	40	48	48	48	48	55	55	55	55	62	62	62	62	69	69	55	55	62	62	48	48	55	55	40
2	40	40	10	13	48	48	40	40	55	55	48	48	62	62	55	55	69	69	62	62	62	62	55	55	55	55	48	48	48
3	40	40	13	10	48	48	40	40	55	55	48	48	62	62	55	55	69	69	62	62	62	62	55	55	55	55	48	48	48
4	40	40	48	48	10	13	40	40	40	40	48	48	48	48	55	55	55	55	62	62	62	62	69	69	55	55	62	62	48
5	40	40	48	48	13	10	40	40	40	40	48	48	48	48	55	55	55	55	62	62	62	62	69	69	55	55	62	62	48
6	48	48	40	40	40	40	10	13	48	48	40	40	55	55	48	48	62	62	55	55	69	69	62	62	62	62	55	55	55
7	48	48	40	40	40	40	13	10	48	48	40	40	55	55	48	48	62	62	55	55	69	69	62	62	62	62	55	55	55
8	48	48	55	55	40	40	48	48	10	13	40	40	40	40	48	48	48	48	55	55	55	55	62	62	62	62	69	69	55
9	48	48	55	55	40	40	48	48	13	10	40	40	40	40	48	48	48	48	55	55	55	55	62	62	62	62	69	69	55
10	55	55	48	48	48	48	40	40	40	40	10	13	48	48	40	40	55	55	48	48	62	62	55	55	69	69	62	62	62
11	55	55	48	48	48	48	40	40	40	40	13	10	48	48	40	40	55	55	48	48	62	62	55	55	69	69	62	62	62
12	55	55	62	62	48	48	55	55	40	40	48	48	10	13	40	40	40	40	48	48	48	48	55	55	55	55	62	62	62
13	55	55	62	62	48	48	55	55	40	40	48	48	13	10	40	40	40	40	48	48	48	48	55	55	55	55	62	62	62
14	62	62	55	55	55	55	48	48	48	48	40	40	40	40	10	13	48	48	40	40	55	55	48	48	62	62	55	55	69
15	62	62	55	55	55	55	48	48	48	48	40	40	40	40	13	10	48	48	40	40	55	55	48	48	62	62	55	55	69
16	62	62	69	69	55	55	62	62	48	48	55	55	40	40	48	48	10	13	40	40	40	40	48	48	48	48	55	55	55
17	62	62	69	69	55	55	62	62	48	48	55	55	40	40	48	48	13	10	40	40	40	40	48	48	48	48	55	55	55
18	69	69	62	62	62	62	55	55	55	55	48	48	48	48	40	40	40	40	10	13	48	48	40	40	55	55	48	48	62
19	69	69	62	62	62	62	55	55	55	55	48	48	48	48	40	40	40	40	13	10	48	48	40	40	55	55	48	48	62
20	55	55	62	62	62	62	69	69	55	55	62	62	48	48	55	55	40	40	48	48	10	13	40	40	40	40	48	48	48
21	55	55	62	62	62	62	69	69	55	55	62	62	48	48	55	55	40	40	48	48	13	10	40	40	40	40	48	48	48
22	62	62	55	55	69	69	62	62	62	62	55	55	55	55	48	48	48	48	40	40	40	40	10	13	48	48	40	40	55
23	62	62	55	55	69	69	62	62	62	62	55	55	55	55	48	48	48	48	40	40	40	40	13	10	48	48	40	40	55
24	48	48	55	55	55	55	62	62	62	62	69	69	55	55	62	62	48	48	55	55	40	40	48	48	10	13	40	40	40
25	48	48	55	55	55	55	62	62	62	62	69	69	55	55	62	62	48	48	55	55	40	40	48	48	13	10	40	40	40
26	55	55	48	48	62	62	55	55	69	69	62	62	62	62	55	55	55	55	48	48	48	48	40	40	40	40	10	13	48
27	55	55	48	48	62	62	55	55	69	69	62	62	62	62	55	55	55	55	48	48	48	48	40	40	40	40	13	10	48
28	40	40	48	48	48	48	55	55	55	55	62	62	62	62	69	69	55	55	62	62	48	48	55	55	40	40	48	48	10
29	40	40	48	48	48	48	55	55	55	55	62	62	62	62	69	69	55	55	62	62	48	48	55	55	40	40	48	48	13
30	48	48	40	40	55	55	48	48	62	62	55	55	69	69	62	62	62	62	55	55	55	55	48	48	48	48	40	40	40
31	48	48	40	40	55	55	48	48	62	62	55	55	69	69	62	62	62	62	55	55	55	55	48	48	48	48	40	40	40

Figure 2.7: Distance information retrieved from Linux on an SGI Altix UV.

but the described approach below offers a portable way to generate distances on all systems, independent of the quality of the SLIT provided by the vendor.

2.3.2 Automatic Matrix Generation

Generating a distance matrix on a system is, in general, a simple task which works in the following five steps:

(i) Linux `libnuma` library is used to allocate chunks of memory on all NUMA nodes.

(ii) Threads are placed on all NUMA nodes in the system.

(iii) All threads running on socket a access the memory chunk allocated by socket b and measure the reached bandwidth.

(iv) This value is stored in an intermediate matrix M as distance between socket a and socket b.

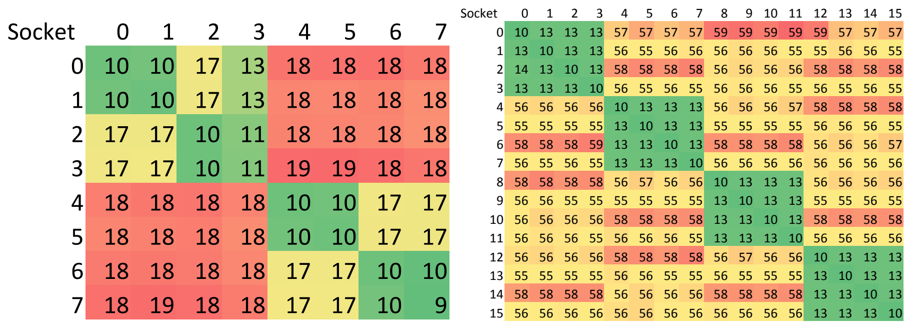
(v) The values are normalized in a way that the upper-left entry $(D(1,1))$ is 10 and the other distances $D(a,b) = (M(1,1)/M(a,b)) * 10$.

Of course the bandwidth measurements deliver lower values for NUMA nodes which are far apart from each other. Through the normalization done in (iv) and (v) the matrix is generated in a way that lower bandwidth values are mapped to higher distances, which is more intuitive for a user. Furthermore, it is more comparable to the values in the SLIT which are familiar to some users already.

These distance matrices can be generated for a full system once and stored in a config file. Of course, it is also possible to measure the matrix when the OpenMP runtime is initialized. The advantage of this approach is, that is can be done also for a subset of the cores in a system. On large systems it is common practice, that a batch job may only run on a subset of the cores, if it does not require the full machine. As a result, only the distances for the sockets used by the batch job are relevant.

Inside of an OpenMP program, I propose an API call as an extension to the OMP API, similar to the `numa_distance(int node1, int node2)` function provided by `libnuma`. The runtime system then shall measure the distances of all sockets used by the program at program start and it shall allow the user to query all relevant distances.

Figure 2.8 shows the measured distance matrices for the HP and BCS machine. The Altix system could not be used exclusively and on the ScaleMP system no remote bandwidth could be measured, because the caching inside of the vSMP system hides the lower remote bandwidth for remote accesses all the time, as I already showed in fig 2.1. Therefore, no matrices for these systems are presented.



Socket	0	1	2	3	4	5	6	7
0	10	10	17	13	18	18	18	18
1	10	10	17	13	18	18	18	18
2	17	17	10	11	18	18	18	18
3	17	17	10	11	19	19	18	18
4	18	18	18	18	10	10	17	17
5	18	18	18	18	10	10	17	17
6	18	18	18	18	17	17	10	10
7	18	19	18	18	17	17	10	9

Socket	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	10	13	13	13	57	57	57	57	59	59	59	59	59	57	57	57
1	13	10	13	13	56	55	56	56	56	56	56	55	55	56	56	55
2	14	13	10	13	58	58	58	58	56	56	56	56	58	58	58	58
3	13	13	13	10	56	55	56	55	56	56	56	55	56	55	56	55
4	56	56	56	56	10	13	13	13	56	56	56	57	58	58	58	58
5	55	55	55	55	13	10	13	13	55	55	55	55	56	56	56	55
6	58	58	58	59	13	13	10	13	58	58	58	58	56	56	56	57
7	56	55	56	55	13	13	13	10	56	56	56	56	56	56	56	56
8	58	58	58	58	56	57	56	56	10	13	13	13	56	56	56	56
9	56	56	55	55	55	55	55	55	13	10	13	13	55	55	56	55
10	56	56	56	56	58	58	58	58	13	13	10	13	58	58	58	58
11	56	56	56	55	56	56	56	55	13	13	13	10	56	56	56	56
12	56	56	56	56	58	58	58	58	56	57	56	56	10	13	13	13
13	55	55	55	55	56	56	55	55	56	55	55	55	13	10	13	13
14	58	58	58	58	56	56	56	56	58	58	58	58	13	13	10	13
15	56	56	56	56	56	56	56	56	56	56	56	56	13	13	13	10

Figure 2.8: Distance matrix of the HP (left) and the BCS system (right). The matrix is scaled such that the upper left value is always ten, larger numbers indicate higher distances.

Obviously, there are differences between both systems. On the HP board there are always two sockets which seem to be connected very fast, whereas the other sockets have distances between 17 - 19. In contrast to the Fujitsu system shown in figure 2.6, this system seems to be built of 2 socket blocks and not 4 socket blocks. But the distances overall are much lower than on the Fujitsu system.

On the BCS system, all accesses on one board are between 10 and 13, which is faster than most of the connections on the one HP board. All connections using the BCS chip are significantly slower, here a distance of 55 - 59 is reached. So, the BCS machine can provide cache coherence over a larger number of cores, but at a cost.

2.4 Summary

In this chapter, I investigated several architectures with a focus on OpenMP programs. The memory tests have shown that the overhead for remote accesses is significant on all hierarchical NUMA architectures, compared to NUMA on standard 2-socket servers. On the ScaleMP system, this overhead is hidden by the software caching mechanisms of the vSMP software. But on all systems the performance for local accesses is not influenced by the additional layer of cache coherency. Also for the memory management in the operating system, significant overhead could be observed, up to three orders of magnitude for a test program with only malloc and free calls.

Besides the memory system, also the OpenMP runtime system can prohibit scaling of an application across a larger NUMA system. I tested several compilers with the EPCC microbenchmarks and showed that the Intel runtime system delivers best performance. The Clang and PGI compiler delivered comparable performance, whereas the GCC and Studio compilers were orders of magnitude slower. In the case of a nested parallel region, the overhead for inner regions was much higher than on the first level and the placement was more important for performance.

Finally, I presented the distance matrix representation of a machine which can be created easily for all systems. The matrix measured can represent the relevant distances of the system better and more reliably than the SLIT used in the OS in some cases.

3 Improved Memory Allocation and Migration in OpenMP Programs

On NUMA machines an application can profit from the combined memory capacity and bandwidth of several memory controllers and memory channels. Since every node adds a new memory controller to the system, this design has the potential to scale much better than UMA machines, as shown in section 2.1. The downside, however, is that the programmer needs to consider the NUMA characteristics of such a machine to achieve performance. In the next chapters techniques to handle these architectures are discussed.

Thread binding for OpenMP programs, which is covered in parts of this chapter, has been investigated in several studies before. In 2006 Huang et al. proposed a grouping of threads in subteams [Huang et al., 2006]. A new clause, named ONTHREADS, is used here to map work on subteams where subteams can run on different parts of the system. This approach is an alternative to nested parallel regions with thread placement. Ayguadé et al. [Ayguadé et al., 1999] and Zhang et al. [Zhang, 2008] both present methods to bind threads to specific cores in a system. In both cases, the cores for placement must be denoted explicitly by the user, which requires detailed hardware knowledge. This knowledge is often not present for average users. Therefore, easier strategies for thread binding were needed as they are available in the Intel compiler with the `KMP_AFFINITY` mechanism for a single level of parallelism or as they have been developed in my diploma thesis [Schmidl, 2009] and presented at IWOMP [Schmidl et al., 2010a] 2010 also for nested parallel regions. Finally, an approach has been added in the latest version of the OpenMP specification, based on a proposal by Eichenberger et al. [Eichenberger et al., 2012] with a place list and different binding strategies as described in detail later in this section.

Besides the thread placement also memory placement and migration is discussed in this chapter. The content presented is partially based on former work presented in [Terboven et al., 2008a], where our `migration-on-next-touch` mechanism for the Linux OS was presented first. The functionality of this mechanism is equivalent to the `madvice` based implementation on Solaris. But, since Linux was missing such a system call I implemented an algorithm based on the Linux system calls `mprotect` and `sys_move_pages` in user space to make this functionality also available under the Linux OS. At IWOMP 2014, this mechanism was described in more detail together with other proposed OpenMP extensions in the OMPX library (see [Schmidl et al., 2014]). Furthermore, a performance study to investigate the usability of the extensions was presented.

Memory affinity and migration for shared memory programming has also been investigated by others in several studies. Hardware support for automatic migration by the IRIX operating system was implemented in the SGI Origin, here TLB miss statistics and remote memory reference counters were used to identify pages for an automatic migration [Laudon and Lenoski, 1997]. In the Sun WildFire, running a modified version of the Solaris OS, the number of occurences that cache lines from a given page are retrieved in the same sharing state were tracked to find candidates for memory migration, see [Noordergraaf and van der Pas, 1999].

Additional software solutions for the Linux operating system exist as kernel patches, like AutoNUMA [Jonathan Corbet, 2014], where statistics on remote memory accesses are collected and threads, processes or data is migrated to establish a good thread to data locality.

Nikolopoulos et al. [Nikolopoulos et al., 2000] presented an integrated compiler / runtime / OS migration framework for user-level dynamic page migration. They use compiler instrumentation and a sampling approach which improves the locality of the memory pages. It turns out that none of these fully automatic mechanisms works well for all applications and that is why other approaches try to allow users to initiate data migration.

An extension of OpenMP for memory migration on NUMA machines was implemented in Compaq's OpenMP compiler [Bircsak et al., 2000]. With the help of a set of additional OpenMP Fortran directives, the programmer has the possibility to specify user-directed page migration and user-directed data layout. In the OMPX library presented here, I provide a similar compiler-independent functionality for today's standard Linux distributions.

Another study about future OpenMP runtime perspectives was made in [Broquedis et al., 2009]. Here, an extension to the scheduler ForestGOMP and a memory manager called MaMI are proposed which should take the burden to maintain data locality in an application. The programmer therefore shall give hints about the memory access, but hints can also be given by the compiler or by hardware counters. The scheduler then shall migrate threads or data to achieve a good locality. The mechanism is evaluated with a set of modified stream benchmarks. In the solution I present in this work, the responsibility to maintain data locality is given to the programmer. The programmer must explicitly migrate data or ask the system to migrate the data on next-touch. The scheduler is not involved in this decision. The advantage of this solution is that the programmer has full control over the data layout.

An application-driven study for the benefit of *affinity-on-next-touch* mechanisms was done by Löf and Holmgren [Löf and Holmgren, 2005]. The experiments in this study were done on SPARC processors under the Solaris OS, where the next-touch migration is supported by the OS. The study showed that an industrial PDE solver could be improved by 166% in runtime. This is a motivation to enable the migration on next-touch mechanism on other operating systems as well, as it presented here for the Linux OS.

In our previous work [Terboven et al., 2008b], we have already sketched a mechanism for *affinity-on-next-touch* for the Linux operating system in user space. A kernel-level implementation for the same purpose with a higher bandwidth is presented by Lankes et. al. [Lankes et al., 2010]. The downside is that a kernel patch is required, so this approach cannot be used on arbitrary HPC production systems. In addition, Goglin and Furmento [Goglin and Furmento, 2009] compared a similar memory migration mechanism, based on our former publication, in user space with an approach in kernel space. Although the kernel-based implementation turned out to migrate pages in 70 % of the time of the user-space implementation, the advantage of the approach presented here is that no kernel patches are needed and thus it can be implemented for any Linux system. Applying patches to the kernel of a HPC Cluster is typically not possible in a production environment for security reasons.

3.1 Facets of Affinity in OpenMP Programs

As shown in chapter 2, on large SMP machines the memory access time rises significantly between local memory and remote memory on different levels. This makes NUMA-aware programming much more important on large SMPs than on standard 2-socket systems, like the Westmere system used in chapter 2. The main goal here is to avoid remote accesses and maximize local memory accesses. For a shared memory program, several conditions influence the locality between data and threads:

(i) The distribution of data across NUMA nodes.

(ii) The placement and migration of threads on the NUMA nodes.

(iii) The work distribution and resulting memory access pattern across threads. Depending on the algorithm this access pattern can change over time which makes the problem highly dynamic.

For all of these problems solutions exist which are commonly used and which work sufficiently well in most standard cases. However, there still exist many cases where additional support is needed which is not yet provided in OpenMP. In the rest of this section, I will give an overview of common practices used nowadays and discuss their advantages and shortcomings. Then I will present some improvements to overcome these shortcomings in the rest of this chapter and in chapter 4.

3.1.1 Distribution of Data

The decision where data is placed on a NUMA system is taken by the operating system. All operating systems used for HPC nowadays (Linux, Windows, Solaris, AIX) use the so called "first touch" strategy: The data is placed in the NUMA node's memory of the core where the first access to the data occurs. If no free memory is available on this NUMA node, the memory is placed on a NUMA node with the shortest possible distance. This decision is not taken per byte, but per memory page, which is by default four kilobyte on a x86 architecture. For OpenMP programs, this automatically puts all the private data into the local NUMA node of a thread. Since no other thread accesses this data ever, it cannot be accessed first by a different thread.

As a consequence of the `first-toch` policy of the OS, it became common practice to initialize the data in parallel, with the access pattern that shall be used later on in the computation, as we described in [Terboven et al., 2008a]. This distributes the data in the same way as it is used later on and thus maximizes the thread to data affinity. There might be some memory pages which are used by multiple threads, e.g. when the chunks assigned to threads are not a multiple of the page size and thus boundaries of chunks arise in the middle of a page. In such cases, only one thread gets local access to the data and the other threads have to work remotely.

If the memory access changes over time, this first touch placement is often not sufficient. In such cases, it is often beneficial to change the data distribution over time. The libnuma library offers the function sys_move_pages to migrate data on Linux systems, even if the placement has already been done. This can be used to change the data layout during the computation if necessary.

Finally, Linux offers high level control on the command line to restrict the mapping of data to NUMA nodes. E.g. the NUMA control tool numactl offers functionality to restrict programs to use only specific NUMA nodes during execution. Furthermore, numactl allows to change the first touch memory allocation policy of the Linux OS for an application to a "round robin" allocation strategy. This leads to a round robin distribution of memory pages over the specified NUMA nodes, independently of the access pattern during the first use of the data. For algorithms with a random access pattern, where the first touch policy cannot be applied in a useful manner, this can be a useful strategy. Although it does not reduce the number of remote accesses compared to other strategies for a random access pattern, it can distributes the remote accesses evenly across all NUMA nodes and at least allows utilization of all available links in the system. A performance comparison of "round robin" allocation compared to first touch allocation will be presented later in this chapter.

3.1.2 Placement of Threads

Since version 3.0, OpenMP offers some basic support for the placement of threads. Besides the direct support in OpenMP, some techniques have become common practice, based on operating system capabilities. Some of these techniques work on all commonly used operating systems and some are specific for the Linux OS, which is the dominant OS for compute clusters in high-performance computing.

For an efficient use of the data placement mentioned before, it is necessary to achieve a fixed mapping of threads to cores. Per default, the OS scheduler decides which threads are executed on which cores of the system, depending on the number of threads which need to be executed, the load of the system and the number of available processors. (A processor in this context is an execution unit seen by the OS scheduler, namely a core or Hyperthread in modern systems.) The mapping between threads and processors can and normally does change over time, which leads to threads being migrated to different processors. On desktop systems, where hundreds of threads might be running on four or eight processors this makes perfect sense, especially since most threads do not create enough work to fully utilize a processor constantly. For HPC applications this is different. All threads can typically fully utilize a processor constantly. Migrating threads in such an environment leads to a loss of cache locality and data locality on NUMA nodes if a NUMA system is used. This problem is well known for years and meanwhile OpenMP offers a standardized way to achieve a fixed mapping of threads to cores.

Two major points influenced the decisions in the OpenMP language committee to support thread placement of OpenMP threads. First, the standardized method should allow users to place threads on shared memory machines in a portable way. This way should be easy to understand and should not require detailed knowledge of the hardware, since this would contradict the idea of being portable between different platforms. Second, the approach should be flexible enough to allow a user full control over the placement if this is needed for the application.

As a solution to fulfill these requirements an abstract description of the machine was specified, a so called `place list`. A `place list` is a list of places, where each place is a set of one or more processors. Every thread is then bound by the OpenMP runtime to one of these places, according to a strategy specified by the user.

In an informal fashion, the strategies can be described as follow. A formal description can be found in the OpenMP specification [OpenMP ARB, 2013].

- **master:** With the master strategy, all threads are bound exactly to the same place where the master thread spawning the team is running.
- **close:** The close strategy specifies that all threads of the newly created team should run on the places next in the place list after the place of the master thread of the team.
- **spread:** When the spread strategy is used, the threads are spread across the place list of the master thread with the largest possible distance. This is realized by dividing the place list into consecutive partitions, one partition per thread if possible and then each thread gets one partition assigned and is placed on the first place in this partition. If such a thread becomes the master of an inner nested team, only its subpartition is used for this inner team. This leads to a distinct placement for all inner teams created by the different threads of the outer team.

The ideas behind the `close` and `spread` strategy are that threads running close together can most likely share resources like the last level cache and the memory of a NUMA node, whereas threads spread over the complete system will most likely use more of the available resources of a system, which results in a higher overall memory bandwidth and a higher total cache capacity.

If no place in the place list contains processors from different NUMA nodes, this placement method can guarantee that no thread is migrated by the operating system to a different NUMA node. Thus, data which is allocated by a thread using the first touch policy will remain in the local memory of this thread during execution.

3.1.3 Distribution of Work

The last mapping which influences the data locality during execution is the mapping between threads and work chunks. In OpenMP there exist different methods to distribute work across threads. The most commonly used method is the `loop` worksharing construct. When work is distributed using this construct, the user can specify a schedule to influence the work distribution. When three conditions are fulfilled, OpenMP guarantees that the work distribution across several loop worksharing constructs is identical. This allows to use this distribution during initialization and execution, which maximizes the data locality as described above. The conditions are:

- The loops need to have the exact same iteration space.
- The loops need to use a `static` schedule.
- The loops need to use the same `chunkzize` parameter or both need to use no `chunkzize` parameter.

In many cases these conditions can be easily fulfilled, as long as the load balance of the loop is regular. For example, in dense linear algebra algorithms often a parallelization over vector elements or matrix rows can be reused for several mathematical operations. Loops with an unbalanced work distribution are discussed later in chapter 4.

For other worksharing constructs like a `single` region or a `sections` construct, the mapping of work to threads is not specified in OpenMP and thus cannot be predicted. However, `sections` are only seldom used and `single` regions normally are not very time consuming, so the data locality of these constructs is normally not a problem in real applications.

A much more challenging problem in this respect is the `task` construct. Tasks are work items in combination with an own data environment. These tasks can be executed by any thread in a thread team, making it very hard to predict the NUMA node where they are executed. Currently, no method exists in OpenMP to achieve local accesses in a task, other than initializing the data in the same task. First attempts for NUMA-aware task programming are discussed further in chapter 4 and tools support needed for such techniques in chapter 5.

3.2 Generating Reasonable Place Lists

The mapping of threads to places is specified in OpenMP. However, a method to generate these place lists is not standardized.

Currently the user can:

(i) use an abstract name (`threads, cores` or `sockets`) to specify the granularity of a place in the place list, or

(ii) explicitly provide a list of places based on processor numbers used by the operating systems.

To fulfill the intention of the placement strategies `close` and `spread`, of course threads being close together in the place list should also be close together in the system. The generation of such a place list fulfilling these requirements is a challenging task. For small systems, e.g. with two sockets, it is typically easy to take all cores of a processor as a group of places close to each other and then putting these two groups behind each other in a place list. However, for large machines, like the SGI Altix UV machine (see figure 2.7 for the SLIT of the system), where several boards are connected with different network topologies, the task to generate a suitable place list is much harder.

The distance information from the system locality distance information table can be used as a basis for this task, but the distance matrix presented in section

2.3 is a more suitable basis, as explained in chapter 2. Still, cores inside of a processor can be seen as close together, since communication inside of a chip is much faster than external communication. Now, it is necessary to sort the sockets in a way that neighboring sockets have the shortest possible distance. However, this problem is equivalent to the traveling salesman problem (TSP), well known in graph theory.

In the TSP, a salesman has to visit a set of cities to sell his goods. The distance between every two cities is given. Every city shall be visited exactly once. Furthermore, the salesman shall return to his hometown at the end, so he should travel in a circle. Of course, a lot of different routes exist to visit all cities. The goal of the TSP is to find the optimal tour with the shortest overall distance.

In our representation of this problem, we have NUMA nodes with distances between the sockets and not cities with distances between each other. But, as mentioned the `place-list` in OpenMP is used by the binding strategies in a way, that neighboring sockets should have a distance as small as possible. Furthermore, if the end of a `place-list` is reached, the strategies perform a wrap-around and restart at the beginning. So, the distance between the last and the first place in the list should also be as small as possible. This results in the requirement to find a list with minimal overall distances between neighbors and between the last and the fist element. This is an equivalent goal than finding the shortest tour in the TSP.

3.2.1 The Traveling Salesman Problem

The Traveling Salesman Problem is known to be NP-hard, so no algorithm is known to solve it in polynomial time.

To solve this task, different options exist:

(i) Solve the problem with a brute-force algorithm in exponential time.

(ii) Use information of symmetry in the network topology to solve the problem more easily.

(iii) Approximate the solution with an appropriate heuristic.

Option (i) will result in an optimal place list, but the algorithm has an exponential runtime. Option (ii) is the best option, if symmetry information on the topology is available and if the complete system is used, since the runtime should be feasible. But normally only the system locality distance information table is available and no further information on the network itself. Furthermore, when only a part of the system is assigned to a job, as it is typical for large systems like the SGI Altix UV, the place list only needs to be generated for the subset of the nodes which are assigned to the job. Providing place lists for all possible job assignments is not possible. An approximation algorithm (option (iii)) is a good compromise, the solution must not be optimal, but it can be computed in polynomial time. And in contrast to option (ii) it can be used even on a subset of the nodes of a system based on a distance matrix which can be easily computed at the beginning of a job as discussed in section 2.3.

3.2.2 Approximation algorithms

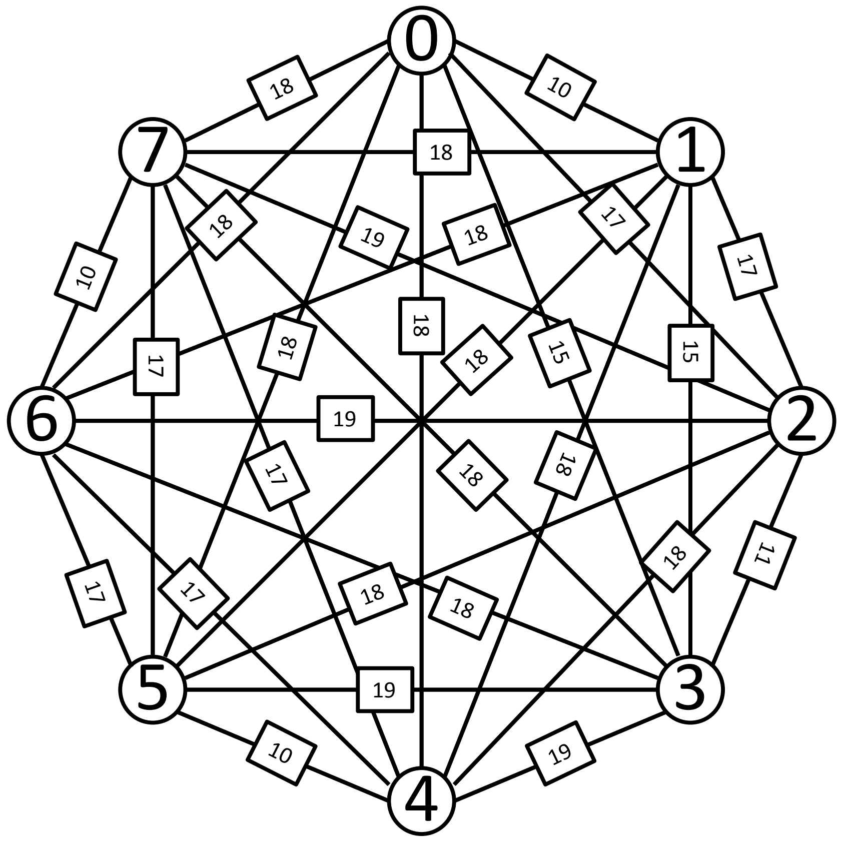


Figure 3.1: Graph representation of the measured distances between the sockets on the 8-socket HP system.

The graph representation of the problem is defined as follows:

Let $G = (V, E)$ be a clique with:

(i) Every NUMA node in the system is represented by a node $v \in V$.

(ii) Every two nodes v and w in the system are connected by an edge e_{vw} with the weight $d_{vw} = \frac{M(v,w)+M(w,v)}{2}$ where M is the distance matrix measured before.

As an example, the resulting graph for the 8-socket HP system based on the measured distance matrix as described in section 2.3 is shown in figure 3.1.

The TSP is NP-hard and it is not possible to approximate the general problem with a constant factor. However, if the edge weights are a metric, there exist approximation algorithms with constant factors, like the Christofides algorithm that produces a 1.5 approximation, see [Christofides and GROUP., 1976] for details. To check if our weights are a metric on the graph we need to verify two points:

- Every two nodes in the graph must be connected by an edge, i.e. the graph must be a clique.
- For all nodes the triangle inequality must hold, this means for nodes v_1, v_2 and v_3 it is always true, that $d_{v_1 v_2} < d_{v_1 v_3} + d_{v_3 v_2}$.

Obviously, the graph generated from the distance matrix or taken from the SLIT is a clique, since I measured all possible connections between sockets and the SLIT also describes all possible connections. Formally showing that the triangle inequality holds is not possible, since the results are based on the bandwidth measured between the sockets. However, it is very likely, that the triangle inequality holds. Assuming it would not hold implies there exist 3 sockets A,B, and C so that the bandwidth measured from A to B is less than the bandwidth from A to C plus the bandwidth from C to B. But the routing algorithm can take any connection on the system, since the bandwidth tests were done exclusively, so it could also take the connection from A to C and then to B as well without the need to intermediately store the data in the memory of C. This intermediate store is necessary with our test, so the direct connection will most likely be faster. In this case the edge weights in our graph build a metric and the algorithm of Christofides can produce a 1.5 approximation of the TSP, which means the length of the tour generated by the Christofides algorithm is smaller than 1.5 times the optimal solution.

Furthermore, it is possible to approximate the solution of the TSP with a simple greedy algorithm. Here, the tour starts at one node and as the next node it always adds the node with the shortest distance which has not yet been chosen. There is no guarantee on the quality of this solution, but it can be computed very easily and fast based on the distance matrix or the system locality distance information table, depending on which information is available.

3.2.3 Evaluation

To compare the different algorithms to generate a path through the distance graph I implemented a brute-force algorithm which generates the distance for all possible tours through the graph. Then it picks the best tour which is possible. This algorithm delivers the best possible place list, but the runtime is exponential in the

System	Algorithm	Distance	Time (sec.)	Tour
HP	brute-force	106	0.00064	7 6 5 4 1 0 3 2
HP	Christofides	106	0.00033	0 1 3 2 4 5 7 6
HP	Greedy	106	0.00004	0 1 3 2 4 5 6 7
BCS	brute-force	376	404k	15 14 13 11 10 8 9 7 6 4 5 3 0 2 1 12
BCS	Christofides	512	0.00084	0 1 2 3 7 5 4 11 8 6 13 12 9 10 14 15
BCS	Greedy	379	0.00004	0 1 2 3 5 4 6 7 8 9 10 11 12 13 14 15
Altix	brute-force	n.a	n.a	n.a
Altix	Christofides	863	0.0022	0 1 3 2 6 7 10 11 14 15 18 19 22 23 26 27 31 30 4 5 8 9 12 13 16 17 20 21 24 25 29 28
Altix	Greedy	848	0.00005	0 1 2 3 6 7 4 5 8 9 10 11 14 15 12 13 16 17 18 19 22 23 20 21 24 25 26 27 30 31 28 29

Table 3.1: Shortest paths generated by the brute-force, greedy or Christofides algorithm.

number of sockets. Furthermore, I implemented a greedy heuristic which always takes the nearest neighbor to the actual socket and the Christofides heuristic. Table 3.1 compares the runtime of these algorithms and the generated tours through the distance graph.

For the 8-socket HP system, the brute-force algorithm finishes in reasonable runtime, but on the 16-socket BCS system it takes a bit less than 5 days and for the Altix system it is not feasible. The heuristics both run in reasonable time, even on the large BCS and Altix systems. The tour produced by the Christofides algorithm is in all cases equal or worse compared to the greedy algorithm. While the greedy algorithm can not guarantee the quality of the result, in our cases it is pretty good. On the HP system the results are of equal quality and on the BCS system the greedy algorithm produces a tour with length 379, with 376 being the optimal result. Because the runtime of both algorithms is low and the place-list only needs to be computed once per program run, we suggest to use bath Christofides and the greedy algorithm.

3.2.4 Recommendation

After evaluating these options, my recommendations to generate a place list for OpenMP is the following:

(i) At the beginning of a program run, detect which cores are in the cpuset of the program.

(ii) Evaluate which sockets are covered by this cpuset.

(iii) Generate a distance matrix for these sockets.

(iv) Use the greedy algorithm to approximate the TSP on this matrix.

(v) Use the Christofides algorithm to approximate the same problem.

(vi) Choose the better solution.

(vii) Take all cores of the sockets in the order in which the sockets appear in the tour as a place list.

Following these steps, it is possible to generate a place list for OpenMP without much overhead which orders cores in the desired way. Based on these place lists, the binding strategies `close` and `spread` can be used for thread pinning as described in the OpenMP specification with the desired result that places with a close distance in the place list also have a close distance on the hardware.

3.3 OMPX: An OpenMP Extension Library for Memory Placement

So far, I discussed the mapping of threads to cores in the system. Furthermore, the mapping of data on NUMA nodes is also important to achieve a good usage of local data for an application. OpenMP so far does not provide any mechanisms to achieve a desired mapping of data on NUMA nodes in a system. However, all operating systems provide some support for this task, like the first touch initialization mentioned at the beginning of this chapter. Here I further want to explain existing methods and propose extensions to the OpenMP specification to provide a platform-independent and user-friendly way to handle the problem of data placement. All OpenMP extensions proposed here have been implemented in the RWTH OMPX library [1] as a proof of concept.

[1] `https://bitbucket.org/rwth-itc-hpc/ompx`

3.3.1 Memory Allocation and Initialization

As described earlier, for OpenMP placement the most common way to achieve a desired data placement is to use the first-touch memory distribution mechanism of the operating system and a parallel initialization of data. For serial or distributed memory programs, this leads to only local accesses of a program if possible, since the data is only used by a single process and thus initialized and used always on the same NUMA node. If the local NUMA node of the thread has no more free pages, of course a page on a remote node is chosen and also remote accesses can occur for serial programs. First touch is applied nowadays by nearly all operating systems, e.g. Linux, Windows, Solaris or AIX, as default policy.

In an OpenMP program this strategy can be used to achieve a distribution of data across nodes by allocating the data in parallel, with threads running on these different nodes. Therefore, an access to the data has to be used in the same way as it will be used later in the computation. E.g. when a vector is initialized by all threads with a `static` schedule clause, it should be used later on in a parallel loop with a `static` schedule. This will then lead to mostly local memory accesses. Since the distribution is on the granularity of a page, there might be some remote accesses if a page is shared between two threads.

This strategy can be easily applied, if the access pattern of the data is known a-priori. If this is not the case, it is impossible to allocate the data with the right access pattern. In such cases, the data distribution cannot be done in a way that mostly local accesses occur. Often serial data initialization is used in such cases which leads to a memory placement where all the data is located on one NUMA node with the first touch OS policy. But, placing the data only on one NUMA node will not only lead to many remote accesses by all threads running on remote nodes, it will furthermore focus all remote accesses on one single NUMA node. A better compromise in such cases is often to distribute the data in a round-robin or random way. This is possible for the whole program, e.g. using the `numactl` tool under the Linux OS as mentioned before. Furthermore, the `libnuma` library provides an API to apply this strategy only for some arrays in a program. I propose a function (`r_ompx_interleaved_alloc`) for OpenMP to achieve this distribution in a standardized way at allocation time. The function shall provide similar functionality as the function `numa_alloc_interleave` from `libnuma`, but a close integration into the OpenMP runtime allows to check the OpenMP place list and distribute data only over NUMA nodes currently used by the OpenMP team. With the `libnuma` functionality this is also possible, but the user would need to query the cpuset of all threads in the team and generate a bitmask for the NUMA nodes to use in the alloc call. A standard way provided by OpenMP would make this much more user-friendly.

3.3.2 Memory Migration

For cases where the memory access pattern is random or unpredictable, spreading data across the whole system is advisable. But sometimes the access pattern of an application changes over time, when different phases of an application run, or when data is adaptively refined. Also if the placement is done in a library which does not respect NUMA characteristics of a machine, a bad data placement may happen which needs to be corrected. In such cases, the placement of data needs to be changed during the execution of a program. Therefore, the operating system needs to provide access for memory migration, which most systems do.

For example under Linux, pages can be migrated using `libnuma` but the user needs to provide a destination for every page of an array which is a complicated task. Therefore, I propose the following additional functions to the OpenMP API for better usability:

r_ompx_migrate and r_ompx_fetch: The function `r_ompx_migrate` provides functionality to migrate a chunk of memory to a specified node and the function `r_ompx_fetch` migrates data to the node where the calling thread is currently running. The first function allows a master thread to distribute data for a team of threads in advance, whereas the latter function allows all threads to fetch data to the local NUMA node. This can be done without the need to find out which cores and NUMA nodes have been chosen by the OpenMP runtime for which thread.

r_ompx_next_touch: This function is useful when the data access pattern is repetitive but not known exactly in advance or when it is hard to divide the data into consecutive chunks, e.g. when the data is accessed indirectly through pointers. One thread can flag a chunk of memory to be moved when it is accessed the next time. The page is then moved to the thread which tried to access the page. This allows to distribute pages during computation without the need to explicitly specify the distribution pattern.

The algorithm to achieve this functionality under the Linux operating system is outlined in code 3.1 and works in two phases. First, all pages of the user specified memory chunk are protected for read/write access and a signal handler for the signal `SIGSEGV` is installed. Second, when a `SIGSEGV` signal occurs, this means a thread has tried to access the page, the installed signal handler migrates the page to the NUMA node of the accessing thread and releases the read/write lock on the page. Then the library returns to the user program where the page is now accessible and located on the local NUMA node.

```
function r_ompx_next_touch(addr, size)
        foreach page p from addr to (addr+size)
                lock p for read and write access;
        end foreach
        install sigh() as signal handler for SIGSEGV;
end function

function sigh()
        p = page causing the SIGSEGV;
        n = NUMA node of the current thread;
        migrate p to n;
        release the read/write lock on p;
end function
```

Listing 3.1: Pseudo-Code illustrating the `next_touch` algorithm.

3.3.3 Performance Modeling for Memory Migration

After the migration methods have been presented, I will analyze the overhead of these methods and compare it with the benefit of the improved memory access pattern. This leads to a simple performance model, which can be used to decide when these migration techniques are beneficial. The tests were done on the Westmere and SandyBridge systems described in section 1.2. Both machines run Linux kernel 2.6.32 and we used 4k pages for all test. The tests perform a daxpy operation on three vectors and measure the access time of all threads and then compute a per page average access time for all threads. The tests use one thread per core on both systems. I investigate the following cases:

- **serial:** The data is initialized by a single thread (in serial) and thus all data is placed on only one of the NUMA nodes.
- **parallel:** The data is initialized in parallel so that the data is distributed across all NUMA nodes and the daxpy operation works on local data.
- **interleaved:** The data is distributed round robin across the NUMA nodes using `r_ompx_interleave_alloc`. During the daxpy operation the data is still used with a static schedule, so the number of remote accesses is not reduced compared to serial. However, the advantage is that the remote accesses are now evenly distributed across the nodes.
- **migrate:** Here the data is initialized serially and then the data is migrated to the thread that will use it later on. We use the function `r_ompx_fetch` for this purpose, but `r_ompx_migrate` has similar overhead, since it uses the same mechanism internally.

- **next_touch:** The data is also initialized serially and then migrated, but in contrast to the former case, the function `r_ompx_next_touch` is used to flag the data for migration and to move it in the first iteration of the daxpy kernel.

Strategy	serial	parallel	interleaved	migrate	next_touch
2-socket Westmere					
Access Time	0.204	0.102	0.141	-	-
Migration Overhead	-	-	-	1.295	15.593
After Migration	-	-	-	0.102	0.102
4-socket SandyBridge					
Access Time	0.334	0.048	0.095	-	-
Migration Overhead	-	-	-	14.458	73.939
After Migration	-	-	-	0.047	0.047

Table 3.2: Average page access time and overhead to migrate a page in μs on the 2-socket Westmere and 4-socket SandyBridge system.

Table 3.2 shows the average time to access all variables of a complete page as well as the overhead for the migration of a page. Investigating the overhead per page allows the evaluation of the migration independent of the algorithm or dataset, as long as the data at least exceeds the size of a memory page. An evaluation on a smaller granularity is not possible, since Linux only allows the migration of a complete page as mentioned before. In the serial case, all pages were located on one NUMA node whereas they were located perfectly distributed across both NUMA nodes in the parallel case. In the latter case, on the Westmere system, this leads to an average access time of about 0.1 μs which is 2x faster than the 0.2 μs when the data is initialized serially. Since two memory controllers are used instead of one this is as expected. When the pages are initialized interleaved, the access time is 0.141 μs. Both migration strategies deliver the same good access time as the parallel strategy after migration, which verifies that the migration works as desired, but they introduce overhead of 1.3 μs for explicit migration and 15.6 μs for next_touch.

On the SandyBridge system the difference between serial and parallel initialization is higher: 0.33 μs compared to 0.05 μs. The performance advantage of the optimized data layout on the 4-socket system is more significant. Because of the more complex topology, the cost of page migration is higher for both migration techniques. The measured time of explicit migration is 14.5 μs and 74 μs for next_touch migration takes.

Since the overhead is much higher than a single remote access of the data, it is only useful if the data is needed multiple times. Given the overhead (O) and access time (A) of table 3.2 the total access time (T) for x memory accesses can

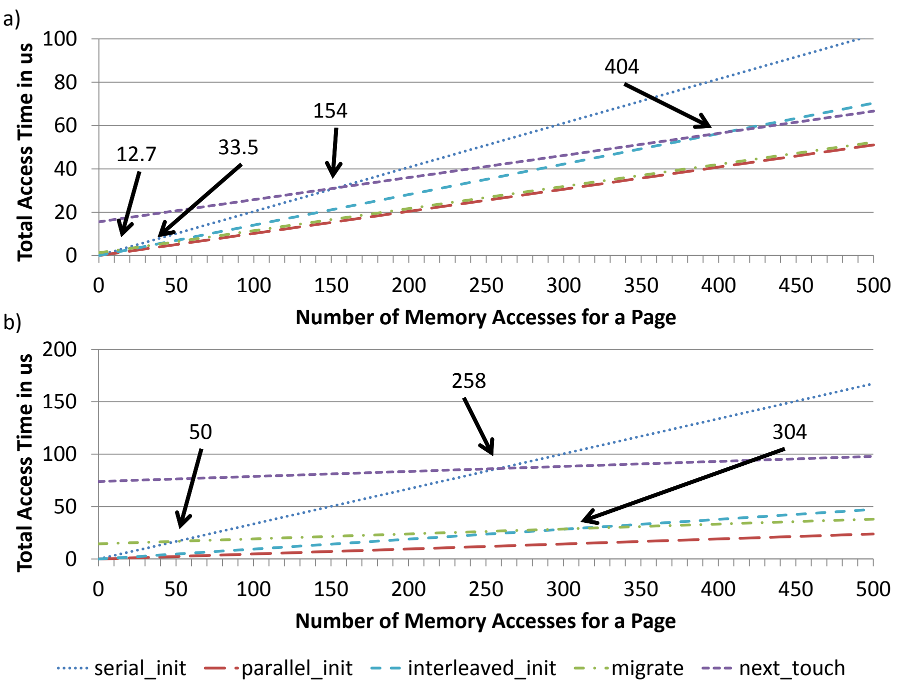


Figure 3.2: Access time for different numbers of accesses of a complete page for different initialization and migration strategies on the 2-socket Westmere a) and 4-socket SandyBridge b) system. The points of intersection indicate when migration is beneficial over `serial` or `interleaved`.

be calculated as $T = O + (x \times A)$. Figure 3.2 shows the access time for the investigated memory initialization and migration mechanisms. The intersection points of the straight lines indicate the break-even point where the migration starts to be beneficial. For example on the Westmere system the lines of `serial` and `migrate` intersect at 12.7 (50 on SandyBridge). This means if the data is initialized serially it is beneficial to migrate it in advance if all the data of the complete page is used 13 (50) or more times. Migration with the `next_touch` mechanism is beneficial for 154+ or 258+ accesses on the Westmere system or SandyBridge system, respectively. This means that explicit migration should be used if possible, i.e. if the data access pattern is known, as explicit migration induces much less overhead. If only a fraction of the data on a page is used, e.g. only one variable, migration will of course pay off only for a higher number of accesses.

3.4 Summary

In this section, I discussed problems regarding the mapping of threads to cores and data to NUMA nodes. Both problems need to be taken into account to achieve a good rate of local accesses on a NUMA system. The larger the system gets, the more important this optimization is, since the cost of remote accesses increases as shown in chapter 2. For the mapping of threads to cores, OpenMP provides good mechanisms to use different placement strategies. The only downside is that the problem to describe the hardware in a so called place list is left to the user. I showed, that the problem can be seen as TSP on the distance matrix or the system locality distance information table, if the distance matrix is not available. The TSP can be approximated with a greedy approach or with the Christofides algorithm in polynomial time and for all investigated systems the result was very close to the optimal solution generated with a brute-force algorithm in exponential time. Since I recommend to use the better solution of the greedy and the Christofides algorithm, the guaranty of Christofides to have at least a 1.5 approximation is always given for the place list.

The problem of data placement on a system can be solved using the first touch mechanism provided by the OS, if the memory access pattern is known in advance and does not change over runtime. If this is not the case, I proposed OpenMP extensions implemented in the OMPX library to handle these problems with interleaved memory allocation or memory migration. Migration can be done explicitly or semi-automatic with the explained next-touch algorithm. Here, the data is migrated to the thread which uses the data next, but the algorithm comes with a significant overhead. I also investigated the overhead and showed how a simple model can be used to decide if migration is beneficial or if remote accesses should be accepted, because the overhead of migration is higher than the gain of it. The problem left for the next sections is the mapping of work to threads which also influences the amount of local and remote memory accesses in a program. Especially for OpenMP tasks this problem is not handled by OpenMP at all so far.

4 NUMA-aware Scheduling Strategies

Distributing work onto threads or processes in a parallel program is one of the most essential tasks in parallel programming. Therefore, this chapter will discuss challenges in work scheduling for OpenMP programs on large NUMA systems. The work I will present here is partially based on prior work. A former version of the load balancing benchmark used for evaluation in this chapter was part of my contribution to [Terboven et al., 2012b] and [Terboven et al., 2012a]. And an initial version of the NUMA aware scheduler was presented in [Berr et al., 2012]. Although the scheduler was used only for a specific application on the ScaleMP architecture, it has proven its ability to lead to better scaling behavior than the traditional OpenMP schedulers. A more general version of this scheduler is presented during this chapter.

The overall goal in work scheduling is to minimize the total execution time of a program. This is typically achieved by an even distribution of work to processes or threads. Depending on the work which needs to be scheduled, different general strategies can be applied.

Static Work Scheduling For many problems the execution time of different work items can be determined a-priori and does not change during the execution of the program. For those problems, typically a work scheduling can be computed locally by every process or thread before the execution of the parallel work. These static work scheduling approaches typically induce low overhead, since no communication at runtime is needed, after the distribution of work is done.

Dynamic Work Scheduling For problems where the computation time for different work items cannot be pre-computed, e.g. because it changes dynamically over runtime, static work scheduling should not be used because it will most likely not deliver an even distribution of work over the threads or processes. In such cases, dynamic work scheduling can deliver a better distribution of work to tasks. Dynamic work scheduling means that the computational work items are assigned to threads or processes at runtime, in a way that distributes work as evenly as possible over the available threads or processes. The well known `master-worker` approach is often used in MPI programs to achieve a dynamic work scheduling (see [Gropp et al., 2014]). In shared memory programs, often runtime systems can deliver dynamic work scheduling. The advantage over static approaches is that the distribution can be adjusted at runtime, which leads typically to a more even distribution of work. The disadvantage is, that parts of the compute resources are utilized by scheduling which introduces overhead. For example, the `master` process in a `master-worker` approach does not take part in the computation of the result and thus introduces overhead needed for the parallel execution of a program.

The OpenMP specification offers different ways for static and dynamic work scheduling. The most commonly used worksharing construct, the `loop` construct, which can be used to parallelize the execution of a `for` or `do` loop in `C/C++` or `Fortran` respectively, comes with a `schedule` clause. The `schedule` clause allows a programmer to choose how the iterations shall be distributed across all threads of the current team. The following strategies can be applied:

static[**:chunksize**] All loop iterations are divided into packages of size `chunksize` and distributed round-robin across all threads. If no `chunksize` parameter is present, the loop iterations are distributed in t evenly sized chunks for t threads.

dynamic[**:chunksize**] The loop iterations are again distributed in packages of size `chunksize`. Then, every thread executes one package. After a thread finishes a package, it gets the next available package. This strategy ensures that all threads participate in the parallel loop execution as long as work packages are available.

guided[**:chunksize**] This schedule is similar to the dynamic schedule. The difference is that the size of work packages is reduced over time. This leads to a lower scheduling overhead at the beginning of the iteration space because of the larger packages, and to a better load balancing because of the small packages at the end of the iteration space.

auto With the `auto` schedule, the compiler or runtime is free to choose any schedule.

runtime The `runtime` schedule allows users to choose one of the above schedules at runtime by setting the `OMP_SCHEDULE` environment variable.

Further OpenMP constructs which allow work distribution are the `section` construct which can be used for a static distribution of a fixed number of work packages to threads and the `task` construct. The `task` construct specifies a package of work and data, a so-called task, which can be computed independently from other tasks. The OpenMP runtime system can execute created tasks in any order and at any time before the next synchronization construct is reached. This allows a dynamic work scheduling of tasks by the OpenMP runtime system, which leads to a balanced work distribution if enough tasks are created.

In addition to standard OpenMP scheduling strategies, different researches invented improved scheduling strategies for OpenMP. S. Donfack et al. [Donfack et al., 2012] combined the static and dynamic scheduler in a similar way as we have done it before in [Berr et al., 2012], but with the difference that they need to precompute the static and dynamic fraction for the scheduler and that their scheduler is optimized only for a dense matrix factorization. Furthermore, they concentrate on cache reuse and not an NUMA issues to optimize their strategy. Kale et al. [Kale et al., 2014] implemented a similar idea where they optimized work stealing to improve the data locality for caches. The work presented here mainly differs in the point that none of the other ideas focused on large NUMA systems and data locality on a NUMA node level and they require scheduling descriptions at compile time, whereas the ideas presented here can be evaluated at run time and thus are more flexible when the dataset changes.

Architectural Trends The problem of load balancing is one of the major problems in efficient parallel programming even today, and architectural trends indicate that it will become even more important in the near future. The reason therefore are two new features of Intel's next generation micro-architecture, Haswell. Today, Haswell-EP processors already exist for up to 2-socket servers, but the Haswell-EX variant which is needed for large shared memory systems is not available. Therefore, I will only mention the features relevant for the load balancing problem discussed here, but cannot present test results for such architectures. However, the methods presented in this work are general enough to be used also on such architectures.

The first relevant feature is an independent clock frequency for the uncore part of the processor. This allows the operating system to change the clock frequency of the cores and the uncore part (memory controller, QPI interconnect, ...) independently. In [Wang et al., 2015] we showed, that this difference is very important for memory bound applications, since it allows to clock down cores for memory bound applications without reducing the memory bandwidth available. This was not the case in former microarchitectures where the memory controller was automatically clocked down with the cores. This feature gives the OS more freedom to change frequencies which potentially changes the runtime of one or more threads in a parallel program and eventually introduces load imbalance.

The second relevant feature is that the AVX vector units are clocked independently from the clock frequency of the core. So inside of one core, two clock frequencies can be chosen. If the hardware or OS chooses different frequencies here for different cores at a time, e.g. because the power budget of the chip is exceeded, this can also introduce load imbalance in a parallel execution.

4.1 Load Balancing versus Data Locality

On NUMA machines work scheduling becomes even more complicated. The overall goal is still to reduce the total runtime, but the total execution time is influenced by the size of a work package as well as by the distance to the data needed for the execution. So, the scheduling problem has two dimensions: The order of work packages to be executed, and the place where the packages are executed. To reduce the total runtime the following factors have to be optimized:

Load Balancing: The work must be distributed in a way that all threads participating in the computation are utilized from the beginning to the end of the computation. Threads finishing early have to wait in the next barrier for other threads and waste CPU cycles. So, the goal is to minimize the overall waiting time in the barrier for all threads.

Data Locality: All threads shall access only local data to minimize the data access time and to finish all work packages as fast as possible. Loading remote data leads to time consuming traffic over the interconnect and increases the computation time for individual work packages. Here, the amount of local data accesses must be maximized.

For dynamic work loads these two goals are often in conflict to each other. The data is often already distributed and maximizing the local data accesses leads to a static work distribution. Optimizing the load balancing cannot fulfill the restriction to have mostly local data accesses, if the data is not distributed evenly, so there exist setups for which it is impossible to fulfill both requirements.

4.1.1 A NUMA-aware Load Balancing Benchmark

To get a better understanding of this problem, I developed a benchmark to investigate the behavior of load balancing strategies on NUMA machines.

In the benchmark a set of work packages is created, which needs to be scheduled on a NUMA machine. A work package does a vector addition, an operation which is memory bound on current architectures for array sizes exceeding the cache size. All work packages work on different vectors which are distributed across all NUMA nodes. The number of work packages is evenly distributed across all nodes. To emulate load imbalance, in the benchmark the size of the vectors differs, leading to different computation times for the vector addition. The first work packages, on NUMA node zero, are the smallest ones and all work packages increase linearly in size. Figure 4.1 shows an example distribution of 12 work packages over 4 NUMA nodes on a 8 thread machine. For a scheduler this case is hard, since the first NUMA nodes have much less local work than the last NUMA nodes.

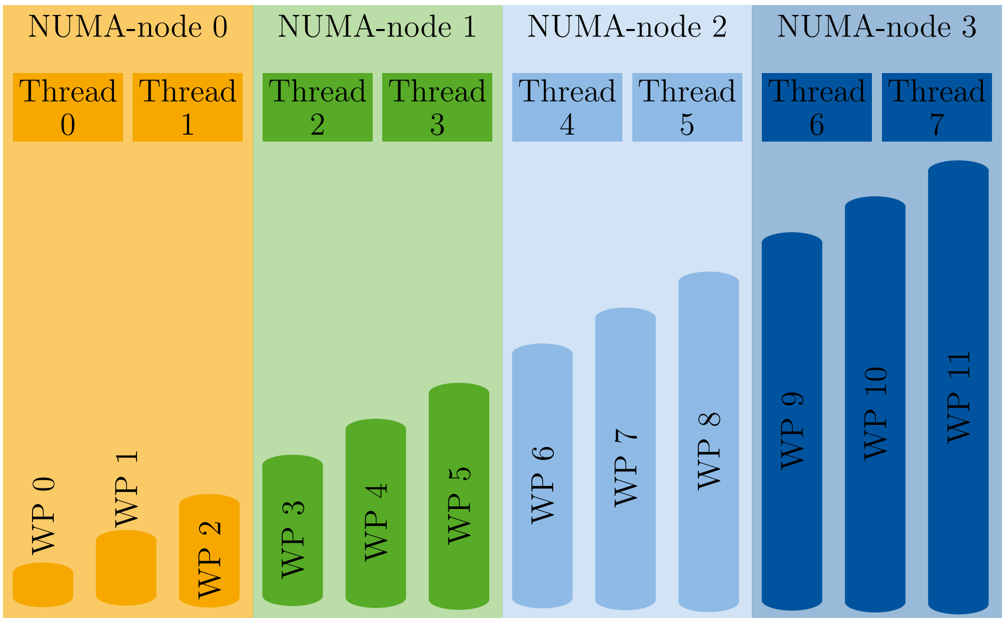


Figure 4.1: Distribution of work packages (WPs) across NUMA nodes for the load balancing benchmark. Exemplary for 12 work packages (WPs) on a machine with 8 threads and 4 NUMA nodes.

On a machine with uniform memory access, the benchmark can be used to investigate the load balancing capabilities of OpenMP's different scheduling strategies. Since only one NUMA node exists on such systems, all memory accesses are local. The benchmark reports for all threads the total execution time for work packages executed by this thread. Figure 4.2 shows the execution time for 59 threads on the Intel Xeon Phi. A total of 3840 work packages were scheduled with a total memory footprint of about 7 GB. The bars show the execution time for a `static` and a `dynamic` scheduling for a `parallel for` loop. The lines indicate the maximum execution time of a thread, since this is the time for the overall execution, which should be minimized. Both schedules work as expected on this system, the `static` schedule distributes the work packages in equal chunks to the threads. Since the vectors in the first work packages were designed much smaller, the first threads finish much faster. The last thread with the largest packages takes the longest time with 2.69 seconds. With the `dynamic` schedule, the work is distributed much more evenly over all threads. Nearly all threads finish at the same time after 1.65 seconds. The `dynamic` schedule leads to a good load balancing on UMA machines, which is no surprise since UMA architectures were common when the schedule was added to the OpenMP version 1.0 in 1997.

Today, the majority of all machines used in high-performance computing provide non uniform memory accesses. This is because 2-socket servers are the sweet spot from a price perspective on the server market today. To better investigate those

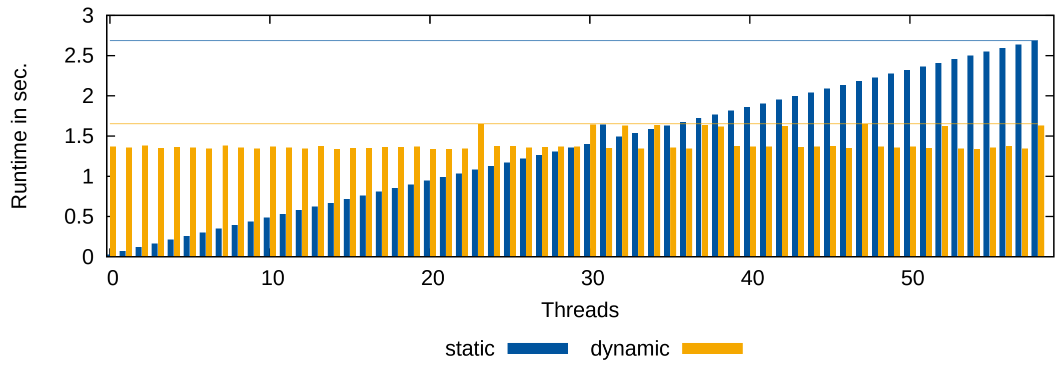
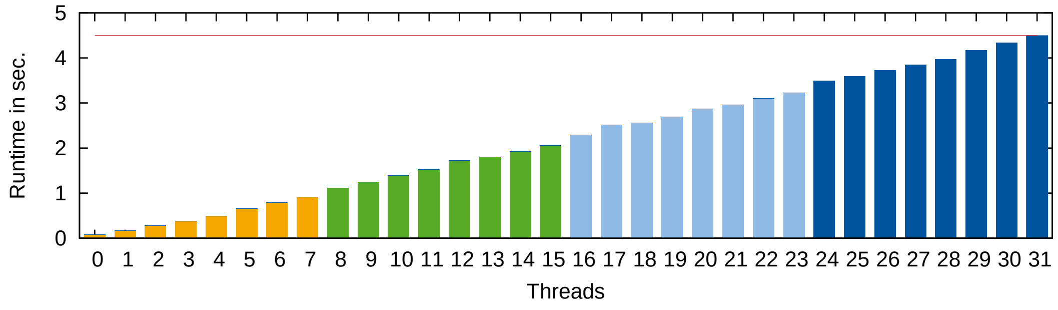


Figure 4.2: Load balancing benchmark results on a Xeon Phi system, running with 59 threads. Scheduling 3840 work packages with a `static` and `dynamic` scheduled parallel loop.

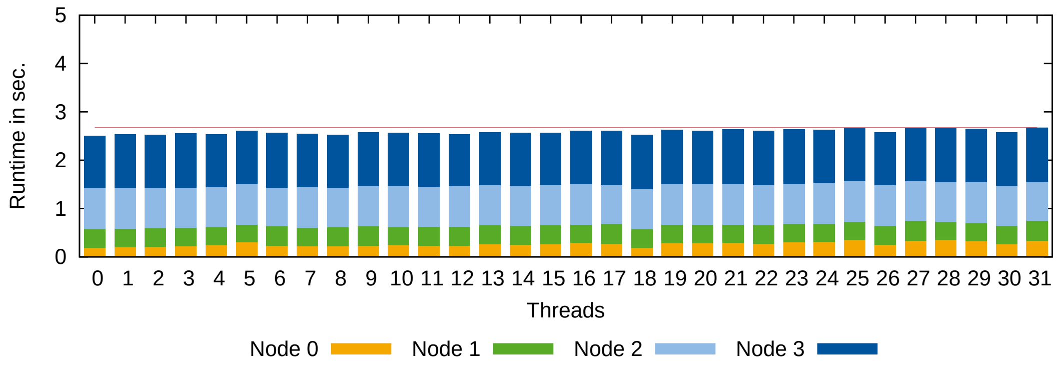
machines, the benchmark does not only report the total execution time for every thread, it also splits this time in chunks depending on the location of the data used during this time. This allows to analyze which threads worked on local and remote data. Figure 4.3 shows the results for a `parallel for` loop with a `static` and with a `dynamic` scheduling clause. The benchmark was run on a 4-socket bull s6030 system equipped with four Intel Xeon X7550 (Nehalem) eight-core processors. For each thread a bar indicates the execution time as before, in addition the color indicates the source NUMA node of the data used. Eight threads were started per socket and the placement was done in a way that the first eight threads ran on NUMA node one, the second eight threads on NUMA node two and so on. The longest running thread shows the overall execution time (red line in the diagram), since other threads have to wait in a barrier at the end of the parallel loop.

For the `static` scheduling of the parallel loop, the overall execution time for the benchmark scheduling 3840 work packages with a memory footprint of about 40 GB was 4.5 seconds. The scheduling was done in a way, that all threads were working on local data only, which is depicted by the fact that the first 8 threads have only yellow bars, the second eight threads only green bars and so on. Although this schedule is optimal for the data locality goal of the scheduling problem, obviously the load balancing is very bad. The first processor finishes all work packages after less than one second and then waits until the work is finished for about 3.5 seconds.

The `dynamic` schedule strategy delivers a much better load balancing. All threads finish nearly at the same time, which is the same behavior than on the UMA system. But, all threads worked on data located on all NUMA nodes, indicated by the yellow, green, light blue and dark blue portion in all bars in the diagram. This indicates, that the data locality goal was not achieved, since about $^3/_4$ of all data accesses were remote accesses. If we look at the overall execution



(a) static loop schedule

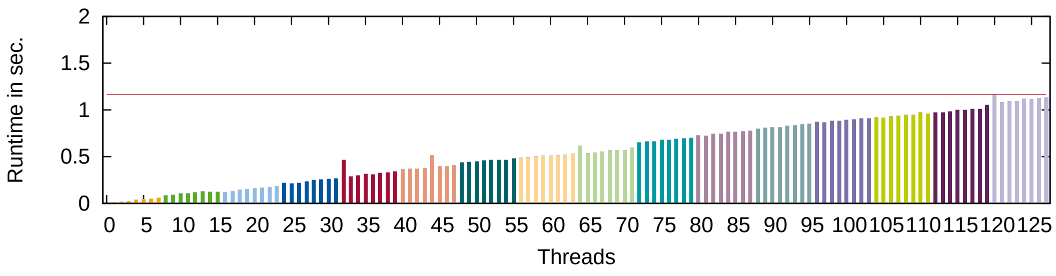


(b) dynamic loop schedule

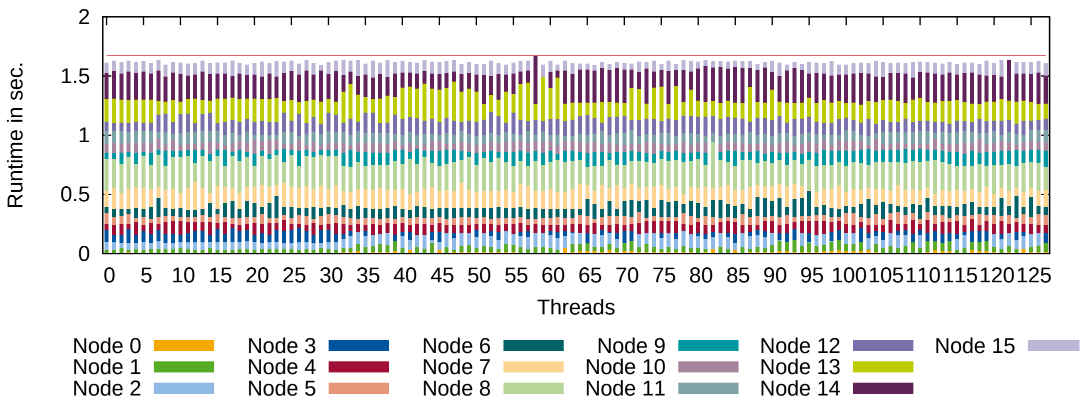
Figure 4.3: Load balancing benchmark results on a 4-socket Bull s6030 system, running with 32 threads. Scheduling 3840 work packages with a `static` and `dynamic` scheduled parallel loop.

time of 2.67 seconds we see, that the time load balancing is still beneficial since we saved about 40% of the overall runtime compared to the `static` schedule.

The need for NUMA-aware load balancing becomes much more relevant, the larger the NUMA systems gets. Figure 4.4 shows results for the same benchmark on a 16-socket BCS machine. Again, the `static` schedule delivers perfect data locality but bad load balancing and the `dynamic` schedule delivers good load balancing and bad data locality. But here, the impact of data locality is much more important than on the 4-socket machine. One reason is that the percentage of remote accesses increases with an unpredictable schedule like the `dynamic` schedule in OpenMP. Only an average of $^1/_{16}$ of all accesses is a local access, compared to $^1/_4$ on the 4-socket system. A second reason is, that the remote accesses get much more expensive, as I showed in section 2.1.1. On the 16-socket machine, the `static` scheduling finishes in 1.16 seconds, the `dynamic` schedule takes even 1.67. In contrast to the 4-socket machine, the `dynamic` scheduling does not help at all to avoid the load imbalance here. Different scheduling strategies are needed to



(a) static loop schedule



(b) dynamic loop schedule

Figure 4.4: Load balancing benchmark results on a 16-socket system, running with 128 threads. Scheduling 3840 work packages with a `static` and `dynamic` scheduled parallel loop.

better support load balancing on large NUMA systems in OpenMP programs.

4.2 A NUMA-aware Work Scheduler

The benchmark results have shown that the standard scheduling mechanisms in OpenMP are not always sufficient for work scheduling on NUMA-systems. Problems occur if

(i) the work packages are memory bound,

(ii) the used data already exists and it is distributed across the NUMA nodes of the system and

(iii) the data used inside of one work package is located on a NUMA node.

If these requirements are not met, the scheduling problem is completely different. If one work package uses data from all NUMA nodes or if the total data is only located on a single NUMA node, memory migration strategies presented in chapter 3 can be used to achieve a better data distribution, but the scheduler above can do nothing to reduce the number of remote accesses. For the rest of this section, I assume the three mentioned prerequisites are fulfilled.

4.2.1 The Scheduling Principle

As mentioned before, NUMA-aware scheduling optimizes two goals. The first one is to keep all threads busy and avoid idle time. The second goal is to work on local data. In figure 4.3 and 4.4 we can see that the static scheduling leads to good data locality at the beginning of the program run. The problem is that threads with only a small amount of the total work become idle very quickly. All threads with a large amount of work, on the other hand, can work very well on local data for a long time. The dynamically scheduled cases show that the load balancing can easily be achieved, when threads also work on remote work packages.

The scheduler presented here is a combination of both strategies. It consists of two steps:

(i) When the execution of the parallel loop starts, the same static schedule is applied to the loop which has been used to initialize the data. This leads to only local accesses for threads working in this phase.

(ii) When a thread has finished all its work packages it would wait in a barrier until all threads finished the loop. The NUMA-aware scheduler instead switches to a different strategy. The thread starts to execute work packages of other threads, in a fashion comparable to the dynamic schedule case.

This combination of static and dynamic scheduling helps to preserve data locality as much as possible. Good data locality is achieved for all threads with a sufficient amount of local work. For threads with only little local work, this goal cannot be achieved by the scheduler. In general, this goal is not achievable without losing the load balancing advantages of the scheduler, if no data is migrated. So, this scheduler aims to provide a best effort scheduling, not to optimize both load balance and data locality.

4.2.2 Work Stealing

The term "work stealing" is often used in the context of tasking to describe the situation when every thread has its own private task queue and an underutilized thread executes a task out of another threads queue. Although no tasks are used in the scheduler, the situation is very similar when a thread goes into the dynamic

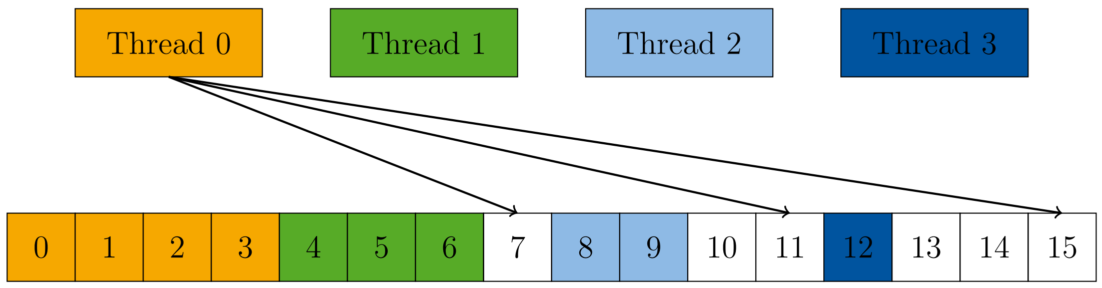


Figure 4.5: Illustration of the NUMA-aware scheduler for 16 work packages and 4 threads. Thread 0 finishes its packages first and can now steal package 7, 11 or 15.

scheduling phase, where it executes remote work packages. So, I will use the term "work stealing" here when a thread executes remote packages.

In the dynamic scheduling phase, the work stealing can be done in three different ways as described below. One thing all these ways have in common is that the stealing is done from the end of the iteration space as illustrated in figure 4.5. Here, thread 0 finishes all work packages first and now it can steal from any other thread, since there are work packages left for all threads. The advantage of stealing work packages from the end of the iteration space is that data between neighboring packages might be located on the same page or even cacheline. When the local thread works from front-to-back and the stealing is done back-to-front, there is the least amount of interference between the local thread and stealing threads.

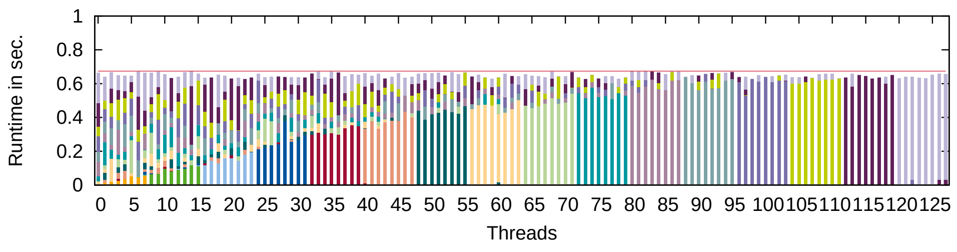
The decision, which work package should be taken, can be done in the following ways:

- Randomly one thread is chosen.
- The scheduler tries to steal from a thread as close as possible to the stealing thread in terms of NUMA distance.
- The thread with the largest number of available work packages is chosen.

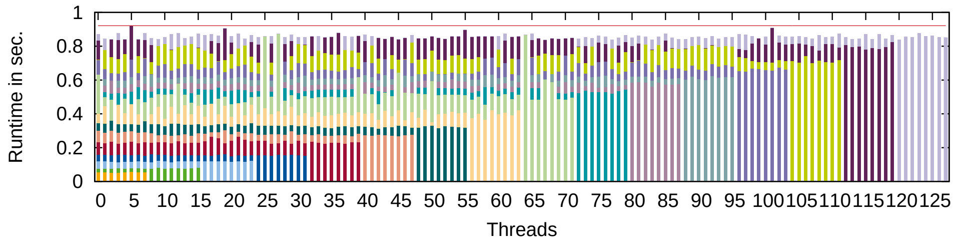
All of these approaches have different advantages and disadvantages and which one is the best highly depends on the number of work packages, their size, their distribution, and on the machine.

Choosing randomly where to steal work is a decision which can be done locally and thus is very fast to take on any machine. But, choosing randomly might lead to a situation where one thread with a lot of work remains the only one with work to do. Then all threads work on data of this thread which further slows down execution.

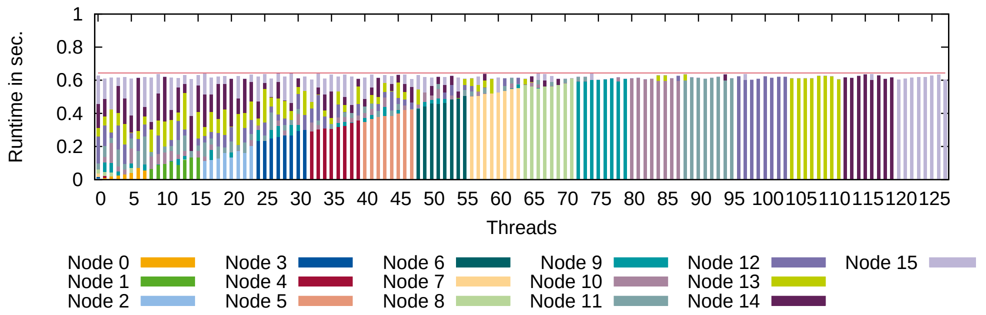
Stealing from threads which are close to the current thread can lead to the same situation, where all threads work on one NUMA node, if the data is not evenly



(a) random stealing



(b) stealing from threads with a small NUMA distance



(c) stealing from the thread with most work packages left

Figure 4.6: Load balancing benchmark results the NUMA-aware scheduler with different strategies for work stealing.

distributed across all NUMA nodes. However, the advantage of this strategy is that at the beginning threads steal from the local NUMA node, which preserves the state where all threads work on local data for a longer time. On hierarchical NUMA-machines the threads will then work on a different socket from the same node, which is still beneficial over a remote node. But at the end, threads have to steal also from remote nodes, if no more local work is available.

The last strategy, where data is taken from the thread with the most work packages left, is probably the best strategy regarding load balancing. If possible, it avoids the situation where only one thread is left with work, since the thread with the most work packages to do gets helpers from the very beginning on, but the decision requires to check for the work left of all threads. This means that on a large machine a value needs to be read from all NUMA nodes, which might be costly. Furthermore, if the work is really unevenly balanced, all threads will steal from the same thread at the same time at the beginning of phase two, and this might make the memory bus of this thread the bottleneck. This will improve when the number of work packages reaches the number of the thread with the second most packages, as then the threads get distributed between two NUMA-nodes and so on.

4.2.3 Performance Results

As shown before, traditional scheduling with the `static` or `dynamic` scheduling strategy is not efficient on large NUMA systems, like the BCS systems, for an unbalanced workload and data distribution for memory bound work packages. I implemented the NUMA aware strategies and ran the benchmark code with these strategies. Figure 4.6 shows the resulting performance. For all strategies, it can be observed that the NUMA-aware scheduling outperforms the `static` and `dynamic` scheduling. The first threads with only small local work packages steal a lot of work and the later threads with larger local work packages steal nearly no work. This leads to a lower number of remote accesses in total and thus to a better overall performance.

For the random work stealing strategy it can be observed, that the first threads steal work from all NUMA nodes. Of course, more work is stolen from the later NUMA nodes, since these nodes contain most of the work. A disadvantage of this strategy is that some of the later threads have to do work stealing, although they would have enough local work to do as too much work was stolen from those threads, leaving them without work before the execution is finished.

In case (b), where work is stolen from threads with a small NUMA distance, it can be observed that all threads from one node start stealing roughly at the same time. Then all threads steal from the next NUMA node and so on. This results in many threads stealing from the same node at the same time and thus it makes the memory controller from this node a bottleneck. This behavior, of course, is caused by the increasing load of our benchmark data set. If the work would be evenly distributed across all NUMA nodes, this pattern could not be observed. However, if there is a load imbalance in the computation between threads, it is unlikely that the load is evenly distributed across all NUMA nodes. Overall, this strategy is worse compared to the random stealing. The overall execution takes 0.92 seconds compared to 0.67 seconds in the random case.

Case (c) delivers the best performance for this benchmark data set with a run-time of 0.64 seconds. Having a closer look at the work distribution, it can be observed that the second half of the threads (64-127) performs nearly no work stealing at all. The reason is that they have enough local work to execute and the stealing scheme is balanced enough to not steal too much work from those threads. This leads to less remote accesses compared to both other schemes.

Overall, the benchmark has shown that the NUMA aware scheduler can deliver much lower remote access rates than a `static` or `dynamic` OpenMP scheduling on the BCS machine for the given data set. This leads to a better usage of the local memory bandwidth, and thus to a better overall runtime. The better scheduling characteristics of option (c) for work stealing overcome the disadvantage that a global table with the number of workpackages left for all threads is needed. Both other options for work stealing led to more remote accesses than necessary for the given dataset. However, the difference between the load aware and the random work stealing is very small with 0.03 seconds. It should be kept in mind, that for the current dataset the work increases constantly for all threads. For a more random imbalance, the random or NUMA distance optimized work stealing might be beneficial. A use case for the scheduler with a realistic data set is shown in section 6. This will further highlight the usefulness of the scheduler in real world applications.

4.3 Locality-aware Task Programming

The NUMA-aware scheduler with different strategies for work stealing implements an efficient way to schedule work packages on NUMA systems. The downside of this approach is that the user has to implement the scheduling strategy explicitly in user code. This slightly contradicts the idea of OpenMP to offer a high level programming paradigm for shared memory programming.

I have shown before that the work scheduling strategies for OpenMP's work-sharing constructs are not suitable for large NUMA systems in some cases. But in version 3.0 of the specification the `task` construct was added to OpenMP. This construct allows to specify a task, which is a bundle of code to be executed and its own data environment. Tasks can be executed immediately or the execution can be deferred. If deferred, the task is put to a queue where a thread of the current thread team can dequeue and execute it. The decisions when a task is executed and by which thread a task is executed are taken by the OpenMP runtime system. In this section, I will investigate if this model allows efficient work scheduling on large NUMA systems with standard OpenMP scheduling mechanisms.

4.3.1 Implementation of task schedulers

The OpenMP specification does not restrict OpenMP runtimes in their implementation of task queues. Even executing all tasks immediately when the `task` construct is reached without any queuing is valid according to the specification. But this would not introduce any parallelism to the program and it is not what programmers expect from an implementation. After tasking has been added to OpenMP, in general, two ways to implement task queues have become common practice.

Central task queues

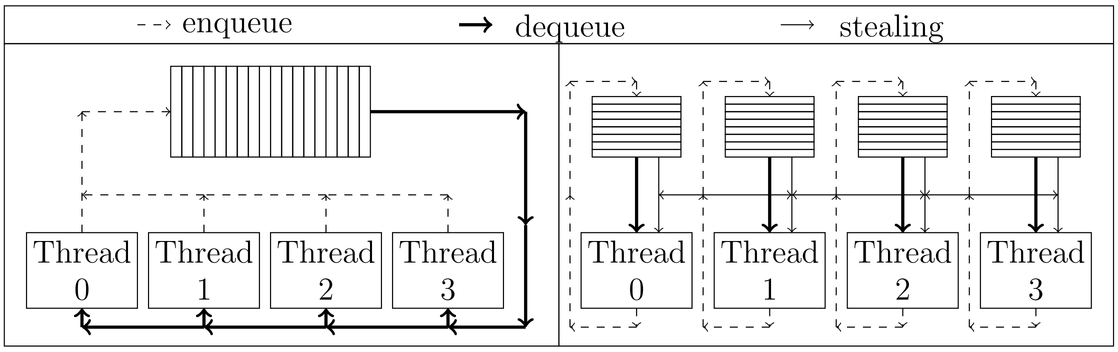


Figure 4.7: Illustration of a central task queue used by four threads (left) and thread-local task queues for four threads (right).

A simple way to implement task queues is to have one central task queue in the OpenMP runtime system. Every thread which encounters a `task` construct can enqueue the task to this central queue or immediately execute the task. The decision if the task is executed or enqueued is taken by the runtime system and might depend on certain heuristics. For example, if the task queue is full, it might lead to a better reuse of the caches to execute the current task instead of dequeueing the first task in the queue and enqueueing the current task. If a thread reaches a synchronization point, it can dequeue a task from the centralized queue and execute this task. Figure 4.7 illustrates the structure of a central task queue on the left side.

The advantage of a central task queue is that it is quite easy to implement and still delivers a good load balancing of tasks across threads. Also the runtime does not have to take any decisions where to enqueue or dequeue tasks, since only one queue exists. These decisions might also add overhead in more advanced implementations of task queues. A disadvantage of this approach has turned out to be the fact that the queue can become the bottleneck in a task parallel program.

Since all threads access both ends of the queue, frequently creating and executing tasks needs locking or atomic access to this data structure. This can easily lead to threads waiting to enqueue or dequeue tasks which is overhead that should be avoided in parallel programs. However, this approach is used in many compilers, e.g. in the GNU or Oracle Compiler.

Thread-local task queues

A different way to implement task queues is to have thread-local task queues. This approach is illustrated on the right side of figure 4.7. The main goal is to overcome the limitations of the central task queue regarding the limited scalability. Here, every thread has its own queue and so tasks can be enqueued in parallel in these thread-local queues. Furthermore, tasks can be dequeued in parallel, as long as all queues are not empty. If the task queue of a thread is empty and this thread has no more work left, task stealing is applied in this approach. Task stealing means that a thread dequeues a task from a non-local task queue. A more detailed description and implementation details can be found in [Duran et al., 2008] for the Nanos runtime and in [LaGrone et al., 2011] for the OpenUH compiler. Furthermore, this approach is also used in the Intel compiler, where the source code is publicly available[1]. The advantage of this approach is clearly that it increases the scalability if all threads enqueue and dequeue tasks. The disadvantage is that there might be overhead added to the dequeue operation when task stealing is applied. Finding a queue that is not empty can involve trying to steal from all task queues in the runtime in the worst case, when only the last one contains tasks.

4.3.2 Task creation patterns

Besides the implementation of the task queues in the OpenMP runtime, also the way programmers create tasks can have a high influence on the performance of a tasking program. In [Terboven et al., 2012a] and [Terboven et al., 2012b] we investigated two ways to create tasks.

- **Single-Producer Parallel-Executor:** In this task creation pattern all tasks are created by a single thread. All other threads of the team participate in the execution of the tasks. This pattern is often the easiest way to parallelize a program with tasks. Often a parallel region with a `single` construct is used to spawn threads and then one thread creates tasks in the `single` construct and all other threads wait in the barrier at the end of the `single` construct and execute created tasks. This construct can easily be applied in situations where normal worksharing is unsuited, like for while loops or for loops where the iteration space cannot be determined in advance.

[1]https://www.openmprtl.org/

- **Parallel-Producer Parallel-Executor:** Here, all threads participate in the creation and in the execution of tasks. To implement this pattern, a worksharing construct like a `for` construct can be used to distribute work among threads and then each thread creates tasks. The advantage of tasks is that they can lead to a better load balancing than the worksharing alone. Another way to implement this pattern is to apply tasks in a recursive way. When all tasks on a recursion level spawn new tasks, distributing these tasks across multiple threads also distributes the creation of tasks of the next level. The advantage of this approach is, that no single creator thread can become the bottleneck.

4.3.3 NUMA-aware task creation

As depicted in section 4.2, the goal for work scheduling on NUMA systems is to maximize the amount of local work done while still keeping all threads busy all the time. When the OpenMP runtime system uses thread-local task queues, the `parallel-producer multiple-executor` pattern can be used for this purpose as illustrated in figure 4.8.

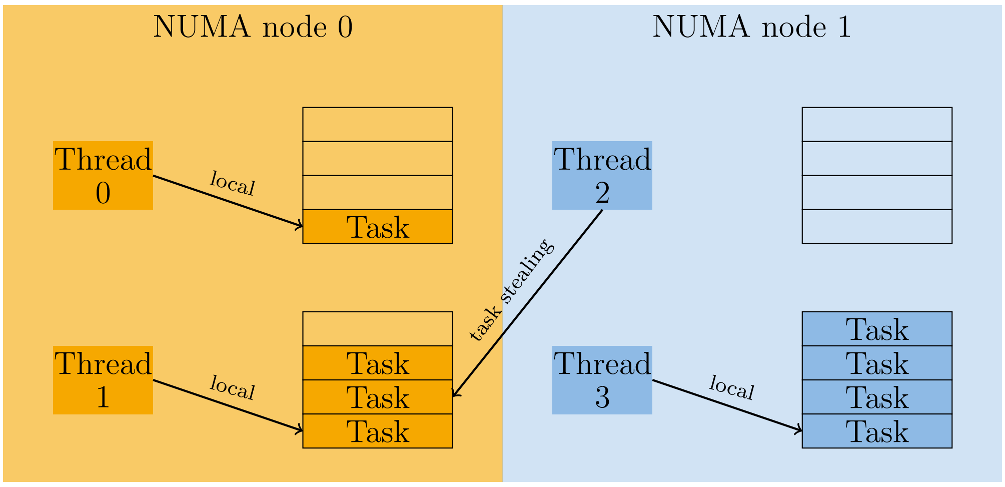


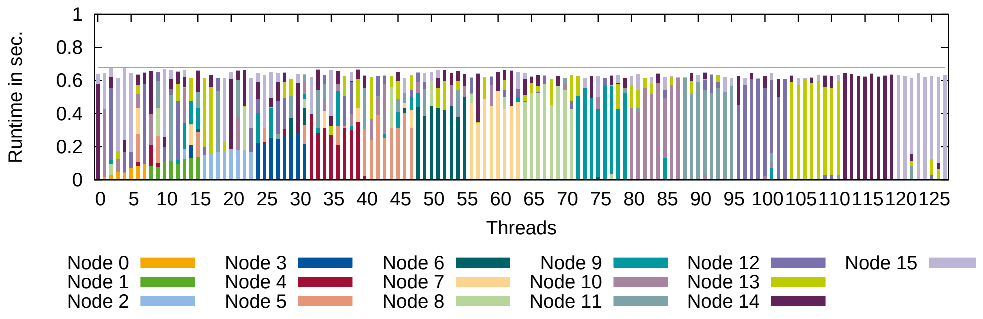
Figure 4.8: Illustration of local task queues for four threads. Thread 2 has an empty queue and applies task stealing. The other threads have local tasks to execute.

All threads start to execute tasks they created first and only steal remote tasks when no more local tasks are available. If all tasks are created by threads where the data is in the local memory, this results in a behavior similar to the NUMA-aware Scheduler with random work stealing.

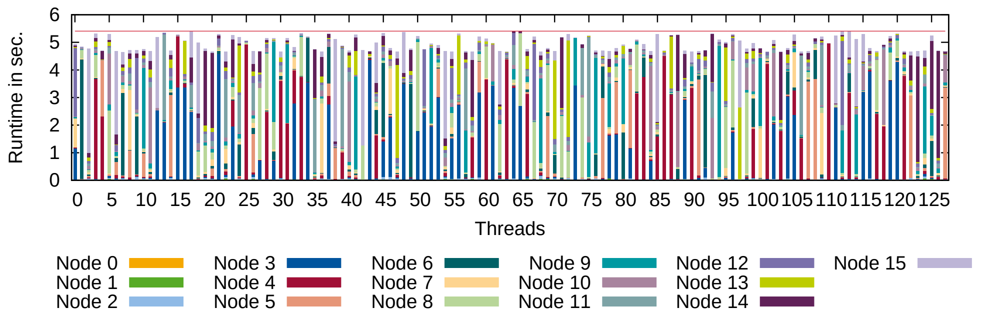
For a programmer, this situation is similar to memory initialization with the `first-touch` policy. The programmer can not influence the placement of the data directly with this policy, but by initializing the data with a thread running on a specific NUMA node the chance that the data will reside on this NUMA node increases significantly. However, the OS can still decide to use a different NUMA node if necessary, e.g. when the capacity of the desired NUMA node is not sufficient. With a NUMA-aware task creation, the programmer can create the task where he thinks it is desirable to execute the task, but the runtime might still take a different decision if needed, e.g. to achieve better load balancing.

4.3.4 Tasking Performance on NUMA systems

To evaluate performance, I modified the load balancing benchmark program to create tasks instead of using the NUMA-aware scheduler or OpenMP worksharing constructs. Hereby I created the tasks in the way described above. All threads created the data as usual and then also created the tasks working on the locally initialized data. Figure 4.9 shows the resulting performance for the Intel Compiler (v 15.0) and for the GNU Compiler (v 4.9) on the BCS system.



(a) Intel Compiler



(b) GNU Compiler

Figure 4.9: Load balancing benchmark results when tasking was applied. All tasks were created by the thread which also initialized the data used by the task, to maintain locality for thread-local task queues.

As described earlier, the Intel runtime applies thread-local task queues, whereas the GNU runtime has a centralized queue for task scheduling. It can be observed, that the performance for the Intel runtime is as desired, the total execution takes 0.68 sec. which is comparable to the NUMA-aware scheduler with random work stealing with 0.67 seconds. Looking at the diagram it can be observed, that especially the larger threads mostly work on local data, whereas the first threads apply stealing, as desired here and also in the NUMA-aware scheduler. With the GNU runtime the behavior looks completely different. The total runtime is about 8 times higher with 5.4 seconds and the distribution of colors shows that all threads execute tasks from all NUMA nodes more or less evenly distributed. The reason is, that the one centralized queue can not preserve any locality information and this leads to a lot of remote accesses and the overall slow runtime.

4.4 Summary

In this section, the problem of NUMA-aware work scheduling has been discussed. Problems occur when work packages use data which is already distributed over the NUMA nodes, since the scheduler needs to optimize the load balancing as a primary goal and also maximize the number of local memory accesses as a secondary target. With the help of the load balancing benchmark presented above, it has been shown that traditional work scheduling methods of OpenMP are not sufficient. The `static` scheduling could not fulfill the requirements to deliver a good balance of load and the `dynamic` or `guided` schedule can not maintain data locality. The presented NUMA-aware scheduler could achieve a good load balancing but still keeps as much data locality as possible. The load balancing benchmark showed a good performance for random work stealing or when stealing was done from threads with most work left to execute. The NUMA-aware work stealing led to a situation where the load balance inside of a NUMA node could be solved, but if load imbalance also exists between NUMA nodes, this strategy was slightly worse than the aforementioned once.

Finally, it has been investigated how centralized and thread local task queues work on NUMA systems and it has been shown how the latter one can be used for a NUMA-aware task creation to achieve better load balancing. The basic idea here is to create the tasks with those threads which also initialized the corresponding data for the task. This strategy could achieve performance on-par with the NUMA-aware scheduler without the need to implement a scheduler by hand for the Intel compiler which uses thread-local task queues. However, for centralized task queues as they are, for example, used in the GNU compiler, this solution performed 8 times slower than the NUMA-aware scheduler. Overall, the strategies presented here solve the scheduling problem of work on NUMA architectures better than all standard OpenMP scheduling methods. Especially on larger NUMA systems the tests have shown shortcomings in the `dynamic` scheduling strategy of OpenMP and here the presented scheduling methods are even more beneficial than on the 4-socket system investigated before.

5 Enabling NUMA-aware Task-performance Analysis

As shown in chapter 4, OpenMP tasking can be used to achieve a good NUMA-aware load balancing. However, applying the tasking feature is not trivial for the following reasons: First, tasks add a second level of parallelism on top of multi-threading and second, the task scheduling mechanisms of the OpenMP runtime influence the performance, as shown in chapter 4, but they cannot directly be observed by a user. Therefore, performance analysis tools are important to help programmers to apply the tasking parallelization in the right way. Since the tasking feature was added later (in version 3.0) to the OpenMP specification, the support in performance tools for tasks, in general, is not as advanced as for the thread level parallelism in OpenMP. In particular, regarding NUMA-aware task programming, nearly no support for tasks is present in performance tools.

In this chapter I will analyze problems which occur in task parallel programs to identify common issues which should be detected by performance tools. Then I will check the ability of state-of-the art performance tools to detect these issues and furthermore present an analysis approach for those problems which turned out to be not detected with these standard tools.

The work presented in this chapter is partially based on previous work published at IWOMP in [Schmidl et al., 2012], where non NUMA-related performance issues for tasking were investigated with the Score-P measurement tool [Knüpfer et al., 2011], [Mey et al., 2012]. Furthermore, at the Parallel Tools Workshop [Schmidl et al., 2013c], a comparison of state-of-the-art performance tools for task related performance analysis was published.

5.1 Task-related Performance Issues

In task-parallel programs, the OpenMP runtime manages a team of threads and applies scheduling of tasks to these threads. So, tasks which have to wait at a synchronization or task scheduling point do not necessarily indicate an idle thread. A performance drawback in the execution of a program only occurs when threads do not execute any tasks or when threads are utilized for overhead through task creation or scheduling. Furthermore, if the `parallel-creator multiple-executor` pattern is used for NUMA-aware task programming, as described in chapter 4, a performance penalty can occur when tasks are stolen from remote NUMA-nodes.

To identify performance problems in task parallel programs, I investigated benchmarks from the Barcelona OpenMP Task Suite (BOTS) [Duran et al., 2008] and applications developed at RWTH Aachen University. Table 5.1 lists those benchmarks and highlights if tasks are created in an iterative or recursive way and if the creation is nested or not.

Barcelona OpenMP Task Suite			**RWTH Aachen University**		
Application	task creation	nested tasks	Application	task creation	nested tasks
Alignment	iterative	no	Sudoku	recursive	yes
FFT	recursive	yes	SparseCG	iterative	no
Fib	recursive	yes	FIRE	iterative	yes
Floorplan	recursive	yes	NestedCP	iterative	yes
Health	recursive	yes	KegelSpan	recursive	yes
NQueens	recursive	yes			
Sort	recursive	yes			
SparseLU	iterative	no			
Strassen	recursive	yes			

Table 5.1: Task generation behavior of the BOTS benchmarks and several applications from RWTH Aachen University.

It can be observed, that recursive task creation occurs frequently, but also iterative creation of tasks is used in many cases. In most cases the nested creation of tasks is applied in the benchmarks and the applications investigated here. Task constructs can be directly nested inside each other which is an advantage of tasking compared to traditional worksharing constructs in OpenMP. Therefore, it is natural to apply tasking in cases where parallelism occurs in a nested way.

Depending on the datasets used for the benchmarks and applications, the following performance issues could be identified.

Too Finely Grained Task Parallelism.

If the execution time of a task is too short, the overhead to create the task or to suspend and resume it exceeds the execution time. In those cases, it is more efficient to execute the task's body directly, instead of creating the task and executing it later. The exact time to create a task and schedule it of course depends on many factors, the hardware, the OpenMP runtime, the data-sharing attributes of the task and so on. Thus, it cannot be quantified precisely when the task parallelism is too finely grained without measuring the overhead and execution time for the specific task instance. Too finely grained tasks often occur when tasks are created recursively in a divide-and-conquer type of algorithm. With every divide step, the tasks get smaller since the problem size shrinks, and at some point, the overhead exceeds the execution time of the task.

Too Coarsely Grained Task Parallelism.

When only very large tasks are created in a program, the load balancing does not work in an efficient way. At the end of the execution, when only a few tasks are left, some threads will execute the remaining tasks and the other threads will idle. If the tasks are very large, this leads to threads idling for a long time. In the worse case, the total number of tasks is less than the number of threads in the current team. Than some threads do not participate in the parallel execution at all.

Task-creation Bottleneck.

When the `single-producer parallel-executor` pattern is used, as described in section 4.3, one thread creates all the tasks. When the producer thread creates tasks slower then the team executes them, the producer will become the bottleneck and other threads will idle and wait until new tasks are created. The occurrence of this problem depends for example on the runtime system, the size of the tasks, the size of the thread team.

NUMA-unaware Task Scheduling.

If the `parallel-producer parallel-executor` pattern is used on NUMA architectures to achieve a NUMA-aware task scheduling, the scheduling decisions of the OpenMP runtime system have a high influence on the performance, as was shown with the help of the NUMA-aware load balancing benchmark in section 4.3. For example, a runtime system with a centralized task queue will not be able to preserve the NUMA-nodes where tasks were created and thus will not lead to a good NUMA-aware scheduling. Here, the overhead of task scheduling is not necessarily high, nor are any threads idling, but the execution of a task will take longer than necessary.

These four performance problems happen frequently in task-parallel OpenMP applications. Since they depend on the amount of overhead and scheduling decisions inside the OpenMP runtime, they are hard to quantify by a programmer without tool support. To analyze programs and detect those issues, the following information is necessary:

(i) the time to execute the task,

(ii) the overhead to create and schedule the task,

(iii) the creating thread of the task,

(iv) the executing thread(s) of the task,

(v) the cores where those threads are running on and

(vi) information on the topology of the system.

Presenting this information to a user can help to investigate the efficiency of a task parallel program and detect potential performance problems during execution.

5.2 Analyzing Tasks with Sampling Based Performance Tools

Now, I will investigate the ability of performance tools to detect and classify the issues mentioned above. In general, two well-established techniques exist to gather performance data for applications at runtime: `sampling` and `event-based` techniques. The ability of sampling based tools to investigate task performance in OpenMP is discussed here, before I present work done in a joint project with the event-based performance measurement system Score-P in the next section. Finally, I will present my extensions to this work to improve the abilities of Score-P for large SMP machines.

Sampling-based performance tools interrupt the application at certain points and record the status of the application at this point. Typically these sample points are taken periodically after a fix time period, e.g. every microsecond, but also other trigger mechanisms like the overflow of a hardware performance counter can be used. This technique is not accurate, but if the sampling frequency is appropriate and the number of samples is high enough it gives a good statistical overview of the application execution. Two performance tools based on this technique are the *Intel VTune Amplifier XE* and the *Oracle Solaris Studio Performance Analyzer*. Here these two tools are investigated as representatives for sampling based tools. As an example application the task-parallel Sudoku solver is used.

Sudoku

The Sudoku solver represents a typical backtracking problem. Backtracking has been identified as a relevant problem class in HPC in the classification from Berkeley, also called Berkeley dwarfs [Asanovic et al., 2006]. Since backtracking algorithms are of a recursive nature, traditional loop level parallelism in OpenMP is hard to apply in such cases, but tasking can be applied much easier. Therefore,

	6						8	11			15	14			16
15	11				16	14				12			6		
13		9	12					3	16	14		15	11	10	
2		16		11		15	10	1							
	15	11	10			16	2	13	8	9	12				
12	13			4	1	5	6	2	3					11	10
5		6	1	12		9		15	11	10	7	16			3
	2				10		11	6		5			13		9
10	7	15	11	16				12	13						6
9						1			2		16	10			11
1		4	6	9	13			7		11		3	16		
16	14			7		10	15	4	6	1				13	8
11	10		15				16	9	12	13			1	5	4
		12		1	4	6		16				11	10		
		5		8	12	13		10			11	2			14
3	16			10			7			6				12	

Algorithm 5.2.1: SUDOKU()

```
for each empty field f
    for each possible number i
        if i is already used
        in row, column or block
            then continue with $i + 1$
            else
                copy the sudoku board
                insert i in f
                create a task to check
                the new board
                continue with $i + 1$
wait for all tasks to finish
```

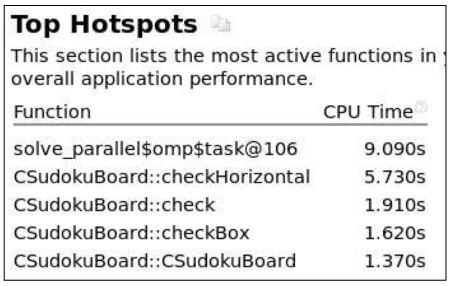
Figure 5.1: A 16x16 Sudoku board with initial entries (left) and the algorithm in pseudocode to solve the sudoku puzzle(right).

this algorithm was chosen as an example here, to study the ability of performance analysis tools in task parallel programs.

For a given Sudoku board, the solver determines all possible solutions of the Sudoku puzzle in a brute force way. Figure 5.1 shows the initial configuration of the board used in this experiment and the algorithm in pseudo-code. For every empty field, the solver tries to insert every possible number. Only if the number is not yet used in the same row, column or block it creates a task to insert the number and check the rest of the board, otherwise no task is created. In both cases the algorithm continues with the next number for the current field or with the next possible field. Every task which finds a valid number and is on the last empty field, stores the current solution as a valid solution for the puzzle. After all tasks have finished, all valid solutions have been found. Note, even if the algorithm is fairly simple, it is hard to fully understand the runtime behavior. For example, determining the number of tasks used highly depends on the initial board and even if the board is known, like the one in figure 5.1, it is difficult to determine this number a-priori. Since tasks are very useful for such dynamic algorithms this is a representative problem for task-parallel programs.

5.2.1 Intel VTune Amplifier XE

The Intel VTune Amplifier XE [Intel, 2013] is a tool produced by Intel to investigate the performance of C/C++, Fortran, Java, C# and assembly code. It

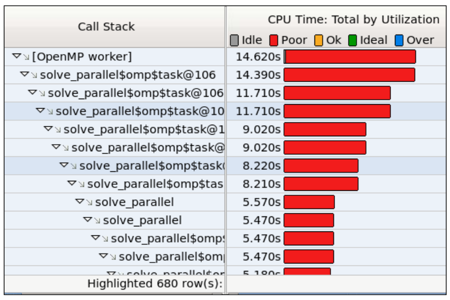


Top Hotspots

This section lists the most active functions in overall application performance.

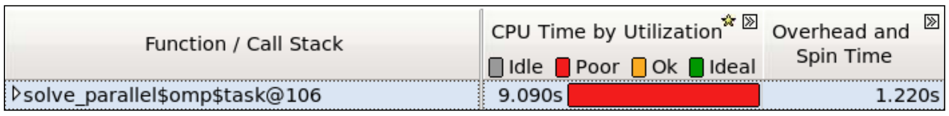
Function	CPU Time
solve_parallel$omp$task@106	9.090s
CSudokuBoard::checkHorizontal	5.730s
CSudokuBoard::check	1.910s
CSudokuBoard::checkBox	1.620s
CSudokuBoard::CSudokuBoard	1.370s

(a) Hotspots



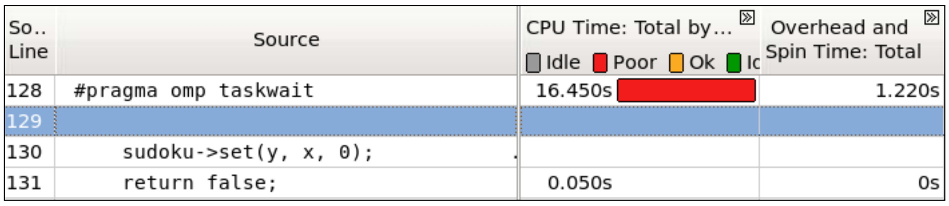
Call Stack	CPU Time: Total by Utilization Idle Poor Ok Ideal Over
[OpenMP worker]	14.620s
solve_parallel$omp$task@106	14.390s
solve_parallel$omp$task@106	11.710s
solve_parallel$omp$task@10	11.710s
solve_parallel$omp$task@1	9.020s
solve_parallel$omp$task@	9.020s
solve_parallel$omp$task	8.220s
solve_parallel$omp$tas	8.210s
solve_parallel	5.570s
solve_parallel	5.470s
solve_parallel$omp	5.470s
solve_parallel$om	5.470s
Highlighted 680 row(s):	

(b) Callstack



Function / Call Stack	CPU Time by Utilization Idle Poor Ok Ideal	Overhead and Spin Time
solve_parallel$omp$task@106	9.090s	1.220s

(c) Overhead for Functions



So.. Line	Source	CPU Time: Total by... Idle Poor Ok Ic	Overhead and Spin Time: Total
128	`#pragma omp taskwait`	16.450s	1.220s
129			
130	`    sudoku->set(y, x, 0);`		
131	`    return false;`	0.050s	0s

(d) Overhead for Sourcelines

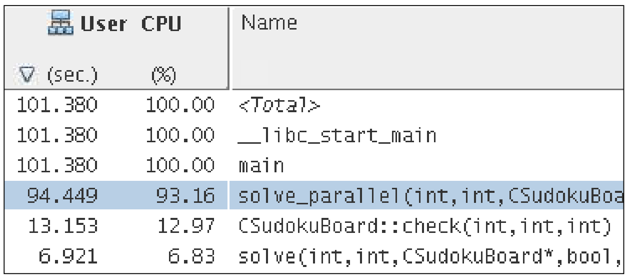
Figure 5.2: Analysis results of the Sudoku solver with the Intel VTune Amplifier XE for 32 threads on a 2-socket SandyBridge system with 32 cores in total.

supports a variety of thread-based parallel programming paradigms like OpenMP, pthreads or Intel Threading Building Blocks. Hardware Performance Counters are supported for all Intel processors.

A performance analysis of the sudoku solver was performed and the basic data related to the tasking performance is presented in figure 5.2. The overview (figure 5.2(a)) shows the top hotspots of the application. This view presents the tasking region in line 106 as major hotspot. More details of the hotspot can be found in the callstack view (figure 5.2(b)), where it is shown that the region recursively calls the `solve_parallel` region, where the task region is inside. The total runtime of all tasks on a certain level of the recursion is presented. Furthermore, the tool measures the "Overhead and Spin Time" on different levels, for functions (figure 5.2(c)) and on a source code level(figure 5.2(d)).

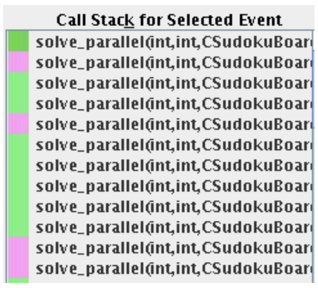
Details of what can be detected in a task-based program will be given in the next section for VTune and the Oracle Analyzer together, since both tools offer a similar amount of information.

5.2.2 Oracle Solaris Studio Performance Analyzer



User CPU (sec.)	User CPU (%)	Name
101.380	100.00	<Total>
101.380	100.00	__libc_start_main
101.380	100.00	main
94.449	93.16	solve_parallel(int,int,CSudokuBoa
13.153	12.97	CSudokuBoard::check(int,int,int)
6.921	6.83	solve(int,int,CSudokuBoard*,bool,

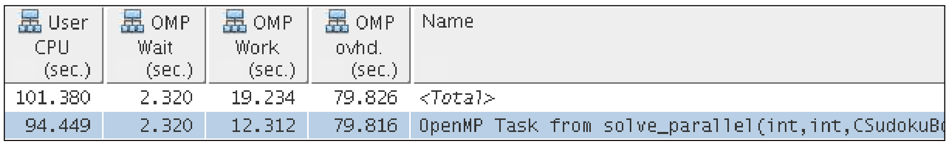
(a) Hotspots



Call Stack for Selected Event

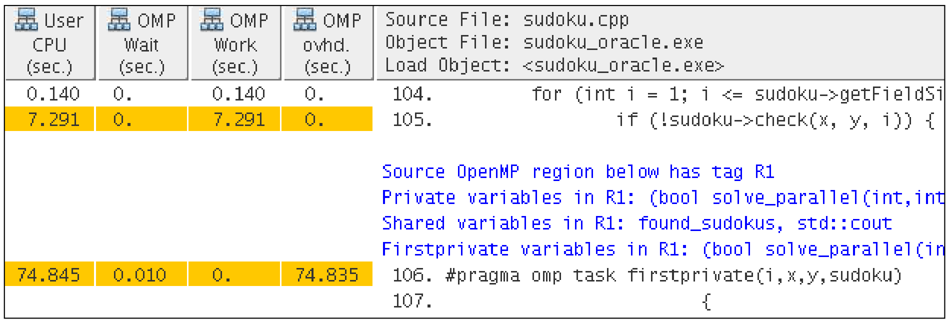
🟩	solve_parallel(int,int,CSudokuBoar
🟪	solve_parallel(int,int,CSudokuBoar
🟩	solve_parallel(int,int,CSudokuBoar
🟩	solve_parallel(int,int,CSudokuBoar
🟪	solve_parallel(int,int,CSudokuBoar
🟩	solve_parallel(int,int,CSudokuBoar
🟩	solve_parallel(int,int,CSudokuBoar
🟩	solve_parallel(int,int,CSudokuBoar
🟩	solve_parallel(int,int,CSudokuBoar
🟩	solve_parallel(int,int,CSudokuBoar
🟪	solve_parallel(int,int,CSudokuBoar
🟪	solve_parallel(int,int,CSudokuBoar

(b) Callstack



User CPU (sec.)	OMP Wait (sec.)	OMP Work (sec.)	OMP ovhd. (sec.)	Name
101.380	2.320	19.234	79.826	<Total>
94.449	2.320	12.312	79.816	OpenMP Task from solve_parallel(int,int,CSudokuBo

(c) Overhead for Functions



User CPU (sec.)	OMP Wait (sec.)	OMP Work (sec.)	OMP ovhd. (sec.)	Source File: sudoku.cpp Object File: sudoku_oracle.exe Load Object: <sudoku_oracle.exe>
0.140	0.	0.140	0.	104. for (int i = 1; i <= sudoku->getFieldSi
7.291	0.	7.291	0.	105. if (!sudoku->check(x, y, i)) {
				Source OpenMP region below has tag R1
				Private variables in R1: (bool solve_parallel(int,int
				Shared variables in R1: found_sudokus, std::cout
				Firstprivate variables in R1: (bool solve_parallel(in
74.845	0.010	0.	74.835	106. #pragma omp task firstprivate(i,x,y,sudoku)
				107. {

(d) Overhead for Sourcelines

Figure 5.3: Analysis results of the Sudoku solver with the Oracle Solaris Studio Performance Analyzer for 32 threads on a 2-socket SandyBridge system with 32 cores in total.

The Oracle Solaris Studio Performance Analyzer [Oracle, 2013] is also a sampling based tool for performance analysis. It can be used to investigate serial or multi-threaded applications written in C/C++, Fortran or Java. Furthermore, it supports hardware counters on Intel, AMD and Sparc processors under Linux and Solaris.

Figure 5.3 shows results for an analysis of the sudoku solver with the Oracle Performance Analyzer. The information is basically the same as with the Intel VTune Amplifier. The detected hotspots (figure 5.3(a)) and the corresponding callstacks (figure 5.3(b)) are shown. In contrast to the Intel tool, it does not show the runtime for different callstack levels, but only the number of callstack levels and the total runtime for a function. The tool also shows overhead spent in the OpenMP runtime system on a function (figure 5.3(c)) or source line (figure 5.3(d)) level. The overhead spent in the runtime is shown at the `task` pragma, so it can be identified as overhead through task creation or scheduling. For this test the Oracle Solaris Studio compiler and runtime system was used as the Analyzer does not support the Intel runtime system. The high overhead value shown here is due to the different tasking implementation of the Oracle runtime system.

Overall, both sampling-based tools deliver the same kind of information. Information on the time spent in all tasks created by a specific task pragma can be seen as well as overhead spent in the OpenMP runtime system for a specific task pragma. The Intel VTune Amplifier XE furthermore allows to differentiate this information for different call stack levels. Both tools, however, are able to identify neither individual task instances, nor the number of tasks executed in total. The reason is that for two samples on the same level, the tools cannot determine if they belong to the same tasks or if they belong to different tasks created in the same function. But this information would be very useful to identify too coarsely or too finely grained tasks.

5.3 Event-Based Performance Tools

Event-based performance tools gather performance data in a different way. They do not gather data at certain intervals, but when certain events occur. Such events can be the beginning and ending of a function, spawning new threads, user defined events and also the begin and end of a task execution. Since OpenMP does not provide a standardized interface to trigger such events yet, many performance tools rely on source-to-source instrumentation. This means that certain calls to a measurement library are inserted into the source code around different OpenMP pragmas. One commonly used tool for this is the ”OpenMP Pragma And Region Instrumentor” (Opari) which automatically inserts calls to the POMP interface, see [Mohr et al., 2002]. Since Opari and the POMP interface have been developed before tasking was supported in OpenMP, initially there was no support for tasks available. We extended both, Opari and POMP, to add task support as described in [Lorenz et al., 2010].

5.3.1 Gathering performance data for tasks in Score-P

Gathering performance data for tasks is more complicated than for worksharing constructs in OpenMP for the following reasons.

(i) After the creation a task can be executed or it can be pushed into a task-queue for later execution.

(ii) During the execution a task can be suspended and resumed later on at certain task scheduling points.

(iii) OpenMP does not provide an identifier for tasks. So, for the measurement system it is hard to detect which task instance is active at the moment.

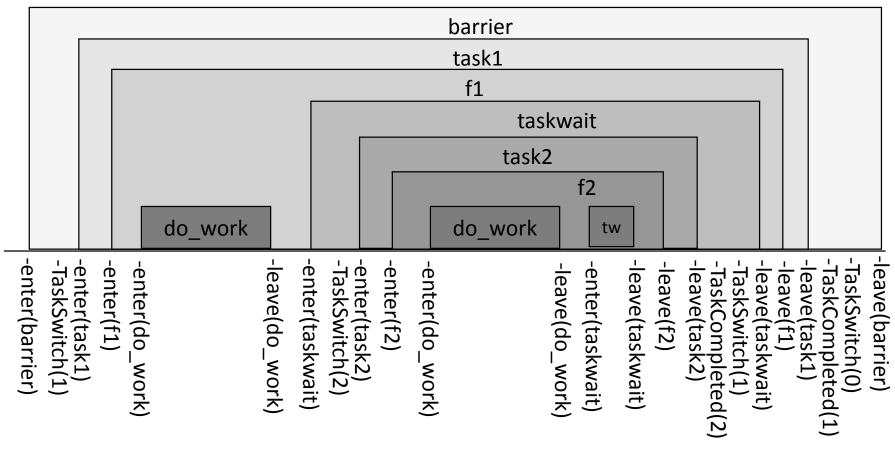
To circumvent some of these problems we implemented a way to add task identifies with Opari during the source-to-source instrumentation step. Therefore, we use a combination of `threadprivate` and task-local variables to keep the task ID. Since this is not part of this work, the reader is referred to [Lorenz et al., 2010] for details. Relevant for this work is, that those task IDs are available in the measurement system, although they are not provided by OpenMP.

Furthermore, tasks which are suspended can lead to a broken nesting of enter and exit events for functions which occur during execution. Performance tools rely on the principle of a function stack per thread, as it is typically used. This implies that an exit event must close the function on top of the stack. With tasks this can be broken, even for a single thread with tied OpenMP tasks. The reason is that a task logically contains its own function stack. When a task enters a function `foo` which contains a task scheduling point and the task is suspended at this scheduling point, no other task which is next executed by the thread can exit the function `foo`. The function `foo` can only be exited after the task has been resumed. To map this onto a thread-based execution scheme, the following steps are necessary:

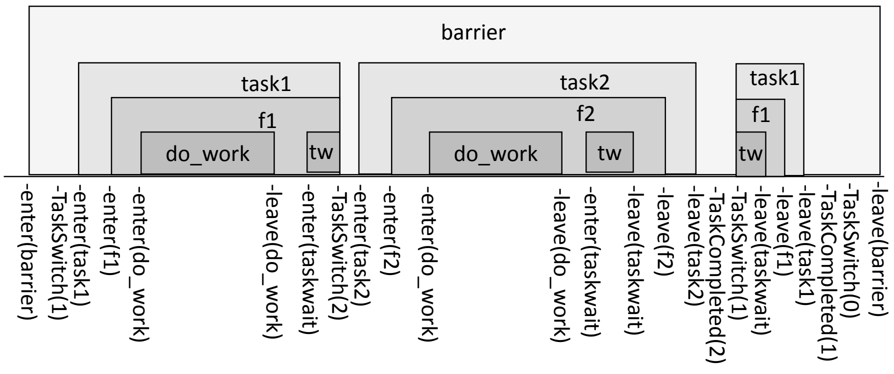
- When a task is suspended, all functions inside of the task are virtually exited, to get back to the function stack of the thread.
- The virtually exited functions are stored for every task instance.
- Whenever a task is resumed, all functions which have been stored for the task are entered again, to restore the state of the stack, when the task has been suspended.

Executing these steps at runtime would potentially add overhead to the measurement and might increase the memory requirement of the measurement system to store all the call stack information of suspended tasks. Since the enter and exit events which are normally written allow to apply these changes later, this can be done in a post-processing step afterwards. I implemented a trace parser which reads an OTF2 trace file generated by Score-P, adds the virtual exit events at task suspension and the enter events if the task is resumed and produces a modified OTF2 trace as output.

Figure 5.4 illustrates the changes on a very simple execution of two tasks. In 5.4(a) the barrier contains the execution of task 1, since the enter event of task 1 is the first event in the trace file. The task enters `f1` and executed `do_work` before it enters a `taskwait` region, where it is suspended. Task 2 is now executed. It enters



(a) original trace



(b) processed trace

Figure 5.4: Call stacks based on the present enter and exit events in the OTF2 trace, in (a) for the original trace and in (b) after the virtual exit and enter events were added.

a function `f2` and also executes `do_work`, before it reaches a `taskwait`. Then task 1 is resumed and finishes execution. Looking at the call stack, it looks like task 2 is part of task 1, which obviously was not the case. The OpenMP runtime would have been free to execute task 2 first, than task 1 and task 2 would be exchanged in the execution. So, besides the fact that problems can occur where this leads to wrong call stacks, even if the call stacks can be displayed by standard tools, this information is misleading for the user.

My post-processing tool transforms the cal stack from figure 5.4(a) into the call stack shown in figure 5.4(b). Here, the complete task 1 is suspended including function `f1` and both tasks are executed as child nodes of the barrier in a call stack view. Using this tool no wrong nesting of enter and exit events is possible and the information reflects the execution model of tasks in OpenMP.

5.3.2 Detecting task-related performance issues

In contrast to the sampling-based performance analysis tools, event-based measurements allow to identify the execution time for individual task instances. Furthermore, the tools can measure the time spent in a task pragma without the task execution time. This allows to investigate the task creation overhead for a task instance.

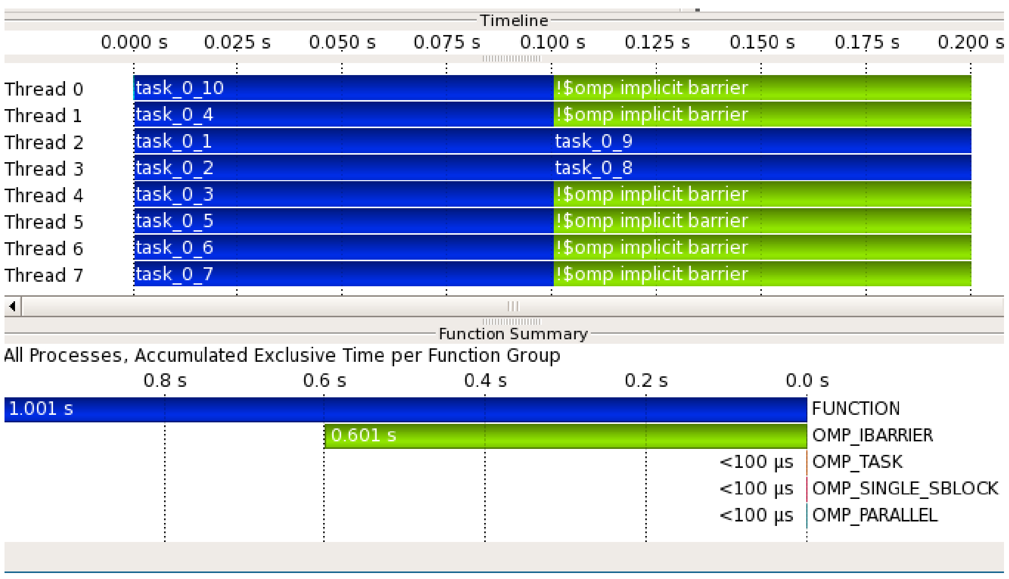


Figure 5.5: Vampir screenshot showing how too coarse grained tasks can be detected.

This measurement technique allows to directly investigate three out of the four performance issues mentioned in section 5.1. I implemented artificial test programs where too coarse grained and too fine grained tasks occur and where the task producer in a single-producer multiple-executor scheme was the bottleneck.

Figure 5.5 shows the performance results visualized in Vampir [Nagel et al., 1996] for too coarse grained tasks. In the timeline view it can be observed that all threads execute a task (blue) of the same size, then only two tasks are left, which are executed by thread 2 and 3, while all other threads are waiting in the barrier (green). When the exclusive time is shown in the `Function Summary` view, it can be observed that roughly 40% of the overall time is wasted by waiting in the barrier. If these tasks were split into more smaller tasks, all threads could be kept active for a longer time and the waiting time could be reduced, leading to a shorter overall execution time.

In figure 5.6, the performance results are shown for the test case with too fine grained tasks. Here the task execution again is shown in blue and the task creation

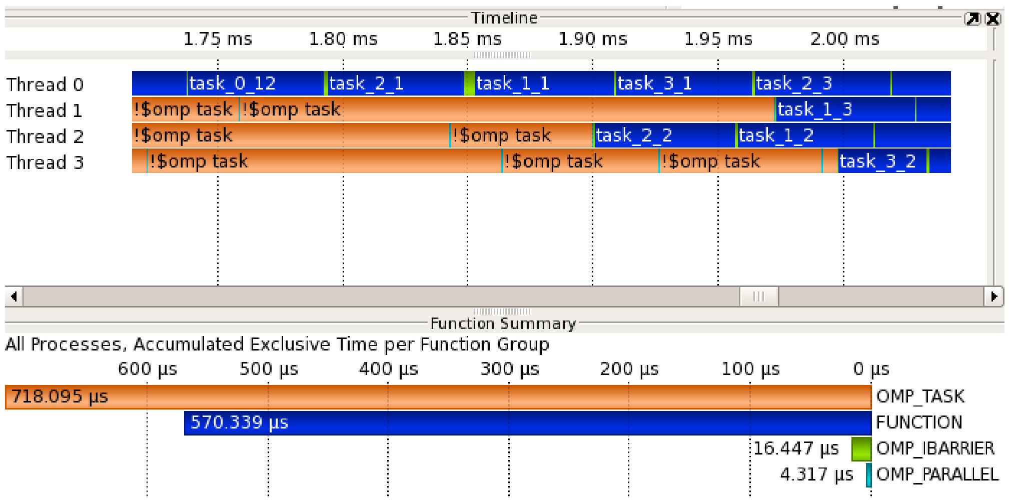
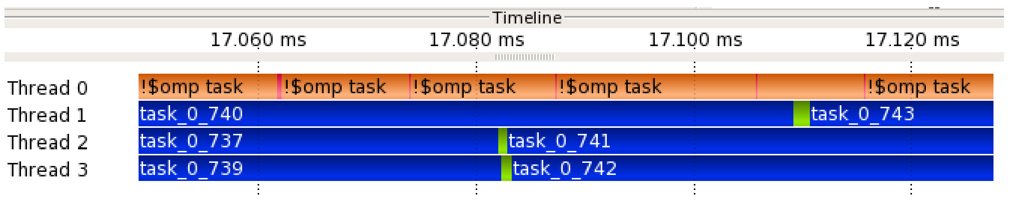


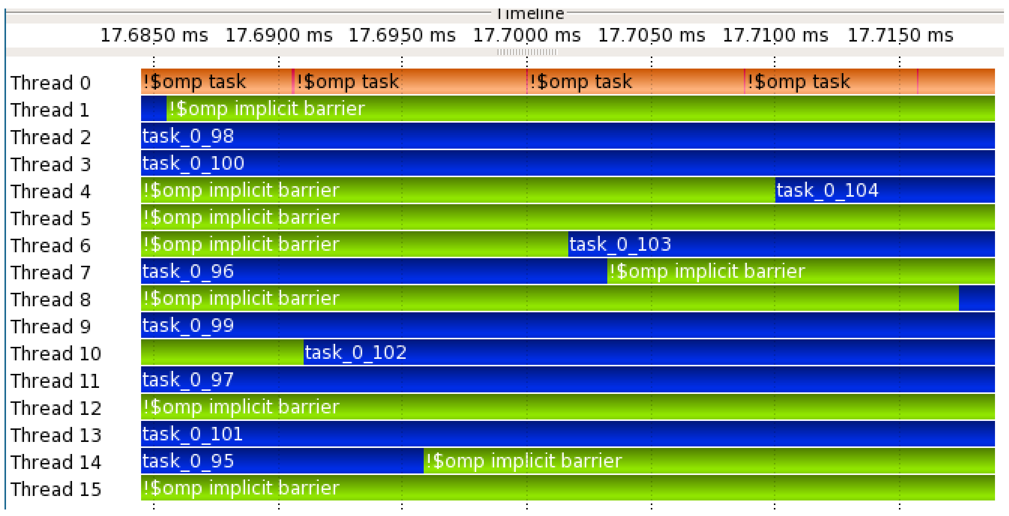
Figure 5.6: Vampir screenshot showing how too fine grained tasks can be detected.

regions are shown in orange. All threads create tasks and execute tasks. But creating a task takes longer than executing a task. This is also reflected in the `Function Summary` view. Overall more time is spent in task creation than in task execution. Here, the overall execution time can be improved if the tasks were not created. The threads could directly execute the task body and thus save the overhead. If the tasks are needed to distribute the work across threads it would be beneficial to merge the tasks into a smaller number of larger tasks. This would reduce the overhead, since a smaller number of tasks is created, but would keep the overall task execution time the same. Typically, this phenomenon occurs when tasks are created recursively for smaller and smaller partitions in a divide-and-conquer like algorithm. At some point the partitions get too small and no more tasks should be generated. This can be done by a cut-off strategy during task creation. An example of this will be shown for the Soduku solver later in this section.

Performance data of the test program where the creator was the bottleneck in a `single-creator multiple-executor` execution model are shown in figure 5.7. The same program was executed with four and 16 threads. When four threads were used, thread 0 was creating tasks all the time and threads 1-3 execute those tasks. With 16 threads again thread 0 created tasks all the time. The other threads execute tasks or wait in a barrier. Immediately after a task is created, one of the waiting threads starts to execute it, e.g. thread 10 starts task execution after the first task was created by thread 0. So, this thread was ready to execute a task, but no task was available. It can be concluded that the creator is the bottleneck here and more executor threads would not improve the overall performance of the program. The problem can be solved in two ways. First, the creator could merge some of the tasks and create larger tasks. This would



(a) 4 threads



(b) 16 threads

Figure 5.7: Vampir screenshot showing the single-creator pattern..

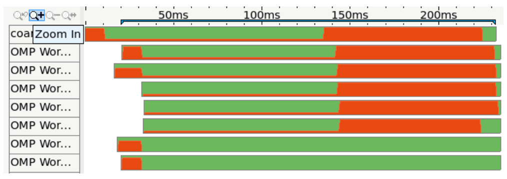
improve the ratio between task creation overhead and task execution time. The bottleneck of the creator would still exist, but the saturation is reached for a larger number of threads. Secondly, the task creation could be parallelized by switching to the `multiple-producer multiple-executor pattern`. This solution would distribute the creation overhead over all threads and no single thread would become the bottleneck.

The fourth mentioned problem, the NUMA-unaware task scheduling, cannot be detected with the given data. Improvements in the measurement tools to detect this problem are discussed later in section 5.4.

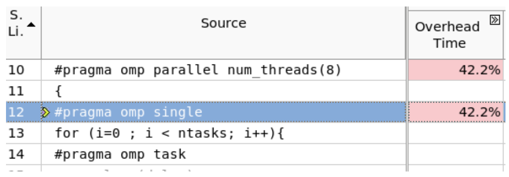
Sampling based analysis of tasking issues To highlight the differences between the information presented using the Score-P measurement system and sampling based tools, I also investigated the three performance issues presented above with the Intel VTune Amplifier XE. The Oracle Solaris Studio can also be used, but the level of information is similar to VTune, so it is omitted here.

Figure 5.8 shows VTune screenshots when the test programs were analyzed. In (a) and (b) it is shown how too coarsely grained tasks can be observed in VTune.

The timeline view shows that two threads at the bottom are busy (green bar) until the end, whereas the other tasks are waiting in a synchronization construct (orange bar). The tool correctly detects high synchronization time which is assigned to the barrier of the single construct as can be seen in the source view. So, the user gets a hint that threads are waiting in the barrier, but no connection to the task granularity can be observed, in contrast to the Score-P based analysis which was presented earlier in figure 5.5.

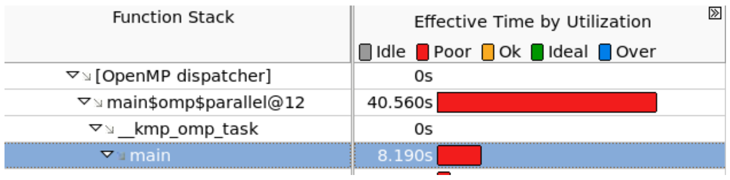


(a) coarse tasks, timeline view



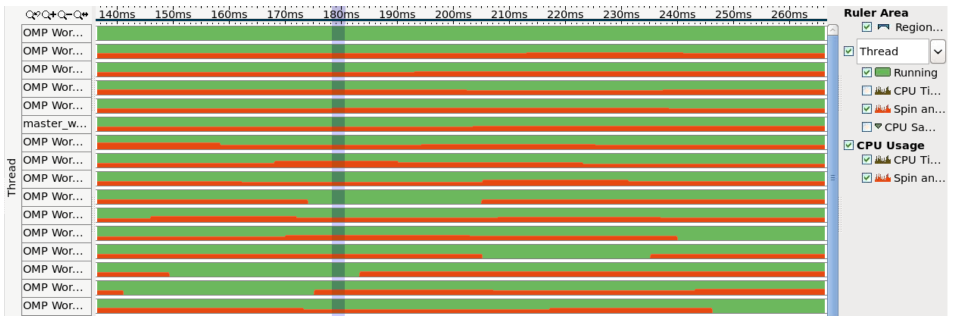
S. Li.	Source	Overhead Time
10	`#pragma omp parallel num_threads(8)`	42.2%
11	`{`	
12	`#pragma omp single`	42.2%
13	`for (i=0 ; i < ntasks; i++){`	
14	`#pragma omp task`	

(b) coarse tasks, source view

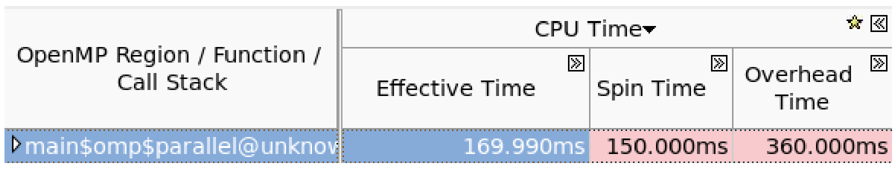


Function Stack	Effective Time by Utilization (⬜ Idle 🟥 Poor 🟧 Ok 🟩 Ideal 🟦 Over)
[OpenMP dispatcher]	0s
main$omp$parallel@12	40.560s 🟥
__kmp_omp_task	0s
main	8.190s 🟥

(c) fine tasks, function stack



(d) single-creator, timeline view



OpenMP Region / Function / Call Stack	CPU Time: Effective Time	CPU Time: Spin Time	CPU Time: Overhead Time
main$omp$parallel@unknov	169.990ms	150.000ms	360.000ms

(e) single-creator, function stack

Figure 5.8: Performance analysis with VTune for the test programs with too coarsely grained tasks, too finely grained tasks and with a bottleneck on the creator site with the single-creator pattern.

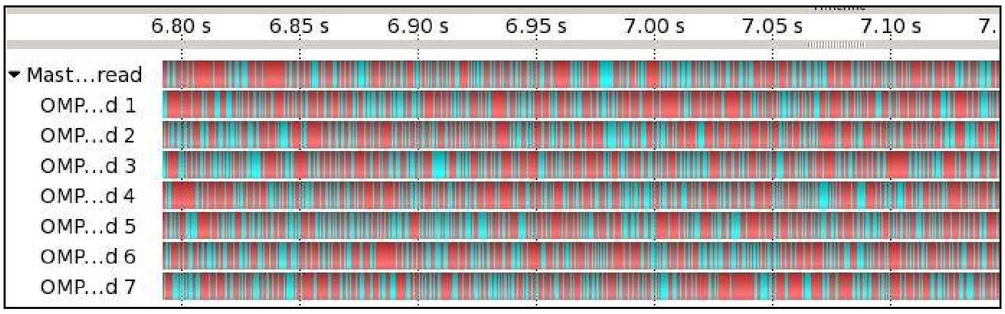
When the test program with too finely grained tasks is analyzed, most of the time is mapped to the parallel region which creates the tasks. But the tasks themselves only consume 8.2 out of 40.5 seconds as can be seen in part (c) of figure 5.8. The tool does not show synchronization time, since task creation does not synchronize. The time is also not shown as overhead time in the tool. The only hint for the programmer about wasting a lot of time is that the task body consumes much less time in comparison to the overall region.

The single-creator test was also investigated with VTune. The screenshots are shown in (d) and (e). In the timeline view it can be observed, that the first thread is busy (green) all the time, whereas the other threads have a fraction of synchronization and overhead time during the execution (orange). Since no regions or individual tasks are shown as it was the case with Score-P (fig. 5.7), no hints are given why the synchronization time shows up. The other views, e.g. the function stack, show a high overhead and spin time for the overall parallel region. Overhead and spin time can be caused by different problems, e.g. load balance or by the issue investigated here, so the information presented by VTune gives a hint which region to investigate, but the level of detail is not as high as with the event-based analysis shown before.

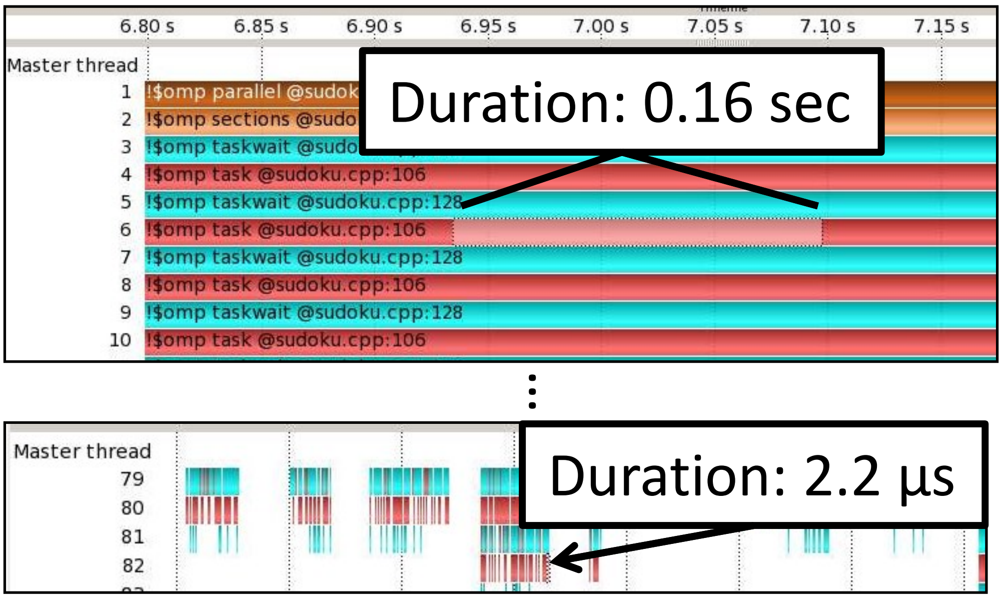
Analyzing the Sudoku solver using Score-P The event based measurement techniques can be used to get a deeper insight into the task execution of the Sudoku solver. Figure 5.9 shows the advantage of the direct measurement techniques compared to the sampling based tools mentioned above (fig. 5.3 and 5.2). In the timeline view, it can be seen that some time is spent in the taskwait regions. This is the synchronization time or the task creation and the scheduling overhead. Also the sampling based tools highlighted a high amount of overhead for the task creation or scheduling, but here we can further investigate the runtime behavior. If we zoom into the callstack view, the time for a single execution of a task instance can be seen on different levels of the call stack. On the upper levels, a task takes 0.16 seconds, but on the last levels of the call stack it only takes $2.2\mu s$. The algorithm calls the same routine and task pragma in a recursive way for all empty fields in the sudoku board. For the last empty cells the created tasks are very small since the remaining sudoku board to solve has no more empty fields. This performance problem (too finely grained tasks) occurs frequently when tasks are used in recursive algorithms and the situation can easily be analyzed with direct measurement tool like Score-P.

After the issue is detected, a well-known solution for this kind of problem is to stop creating tasks at the upper levels, where the tasks are of reasonable size. This strategy is called cut-off mechanism. I implemented a cut-off for the Sudoku solver and the performance results are shown in figure 5.10. Obviously the detected performance problem is fixed and now the program reaches a speedup of sixteen instead of four.

Although the direct measurement tools provide more detailed information than



(a) timeline


6.80 s
6.85 s
6.90 s
6.95 s
7.00 s
7.05 s
7.10 s
7.15 s
Master thread
1 !$omp parallel @sudok
2 !$omp sections @sudo
3 !$omp taskwait @sudo
4 !$omp task @sudoku.cpp:106
5 !$omp taskwait @sudoku.cpp:128
6 !$omp task @sudoku.cpp:106
7 !$omp taskwait @sudoku.cpp:128
8 !$omp task @sudoku.cpp:106
9 !$omp taskwait @sudoku.cpp:128
10 !$omp task @sudoku.cpp:106
Duration: 0.16 sec
Master thread
79
80
81
82
Duration: 2.2 µs

(b) callstack

Figure 5.9: Vampir screenshot showing the timeline and callstack view for the Sudoku solver.

the sampling based tools, there is also a disadvantage using this technique. The overhead for the sampling based tools increases linearly with the sampling frequency of the tool. Since the sampling frequency can easily be controlled, the typical overhead can be kept sufficiently low for such tools. For the direct measurement tools, the overhead is proportional to the number of events which occur, which is the number of tasks in our case. Since the number of events is controlled by the program, the tool cannot influence the overhead directly. In many cases it can lead to a massive distortion of the results. Table 5.2 shows the overhead for the measurement with the mentioned tools and also the size of the output data. Both sampling-based tools introduce a low overhead and only need a few megabyte of storage for the results. Score-P introduces 150% overhead for the Sudoku solver, which is very high, and in the tracing case, it also produces a lot of output data,

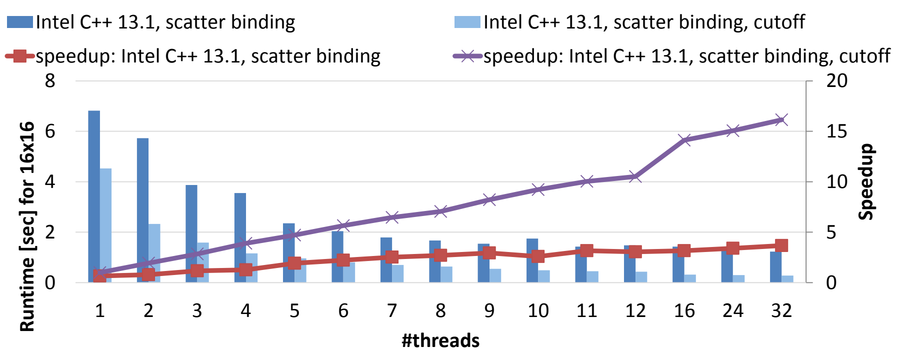


Figure 5.10: Performance of the Sudoku solver example without and with the cut-off strategy used.

16 Threads	Runtime Overhead (setup routine)	Data Volume (complete program)
Oracle Analyzer	<1%	28 MB
Intel VTune	7%	8.5 MB
Score-P (Profiling)	∼ 150%	44 KB
Score-P (Tracing)	∼ 150%	1.2 GB

Table 5.2: Overhead and generated amount of data for investigated tools for the Sudoku code.

1.2 GB. For programs running a few seconds this is very much data and for real applications the amount of data generated can easily exceed the capacity of the storage system and of the analysis tools.

In summary, sampling based tools give a good overview of the execution and they come at a lower price in terms of overhead and storage needed. If this level of detail is sufficient, these tools should be used. If details of the individual task instances are needed, which allow to detect certain task-related problems more efficiently, the overhead for the direct measurement tools has to be paid. These tools can show detailed information on individual tasks and also on the creation overhead for a task. If the same pragma is used in a recursive way, it often creates tasks of different sizes on different levels of the call stack. In such situations it is beneficial to analyze individual task instances to understand the runtime behavior and to detect the performance problem.

5.4 Improvements to Investigate Tasks on NUMA Machines

In section 4.3, I discussed how scheduling strategies of the OpenMP runtime can influence the performance of a tasking program on NUMA machines. Furthermore, I explained how a programmer can use the `parallel-producer multiple-executor` paradigm during task creation to influence the scheduling under certain conditions. As shown in chapter 4, the decisions of the runtime system to manage or steal tasks influence the performance of a program. For example, the difference between a runtime with thread-local tasks queues and a central task queue was about a factor of 8 for the load balancing benchmark on the BCS machines, as shown in figure 4.9. If the internal behavior of a runtime is essential for the execution of a program, a performance tool should be able to investigate this behavior. To the best of my knowledge, no performance tool provides this kind of information today.

5.4.1 Shortcomings in Score-P Regarding Task Analysis

At the beginning of this chapter, I mentioned information which is needed to highlight the relevant runtime decisions on NUMA machines. Currently the direct measurement tools allow to investigate the time to create and execute a task and they also show the thread which executed the task. Furthermore, the task identifiers we implemented in Score-P allow to distinguish task instances. Given the task ID when a thread creates a new task and the task ID when a thread executes a task allows detection of task stealing. Furthermore, the thread where a task was stolen from can be detected.

Missing in a Score-P measurement currently is the information on the mapping of OpenMP threads to cores and sockets. This information is essential to determine if tasks were migrated across NUMA-nodes.

5.4.2 Combining Traces with Hardware Information

As a proof-of-concept I modified the trace rewriter tool mentioned above in a way that it analyses the NUMA-related task scheduling decisions. First, I provide a mapping file which describes the hardware and the mapping of threads onto the hardware. The distribution of cores to NUMA-nodes can be easily queried from the Linux operating system. The distribution of threads onto these cores in general is not fix and it can be influenced by the operating system, as described in chapter 3. But OpenMP allows to achieve a fixed binding with the `OMP_PROC_BIND` environment variable. This mechanism was used throughout these tests to achieve the described mapping of threads to cores during execution. Then the parser tool reads the trace as well as the mapping file and investigates all executed tasks and splits them into the following groups:

- **local queue**
  Tasks in this group have been executed by the thread which created the task.
- **same socket**
  Tasks in this group have been executed by a thread different from the creator thread, but they still run on the same socket where they have been created.
- **remote socket**
  Tasks in this group have been created by a thread on a remote NUMA node.

Then the parser tool writes the same trace again to disk but it modifies all task events in a way that the groups can be seen in the resulting trace file. The workflow of the rewriter tool is sketched in figure 5.11. For production use of these improvements it would be beneficial to write the hardware and mapping information into the OTF2 trace file. Furthermore, the grouping of tasks into local and remotely executed tasks could be done directly in the performance tool. This would reduce the overhead of use, but since the Vampir tool is not publicly available, I implemented the rewriter tool as a proof of concept.

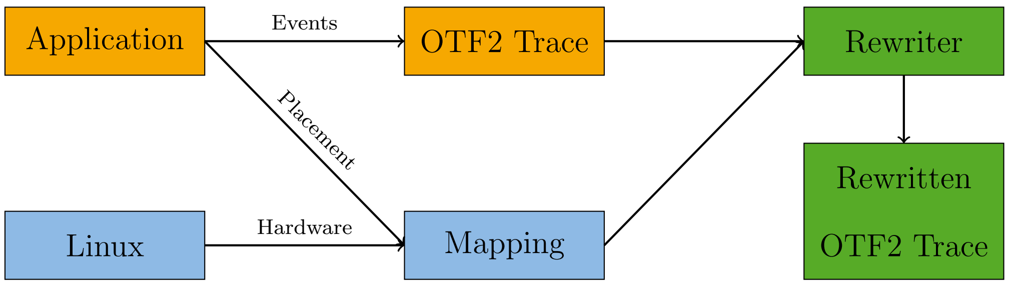


Figure 5.11: Workflow of the trace rewriter tool to use hardware information to analyze NUMA related task scheduling issues.

5.4.3 Evaluation

Since the resulting trace is still a valid OTF2 trace file, it can be visualized in Vampir as shown in figure 5.12. Here the color coding possibilities of vampir are used to show the tasks from the local queue in green, task from the same socket in yellow and tasks from a remote socket in red. The execution of the load balancing benchmark from section 4 is shown and the Intel compiler and a `parallel-producer multiple-executor` task creation scheme was used. So, this is the best case for this benchmark when tasks were used. It can be seen that a

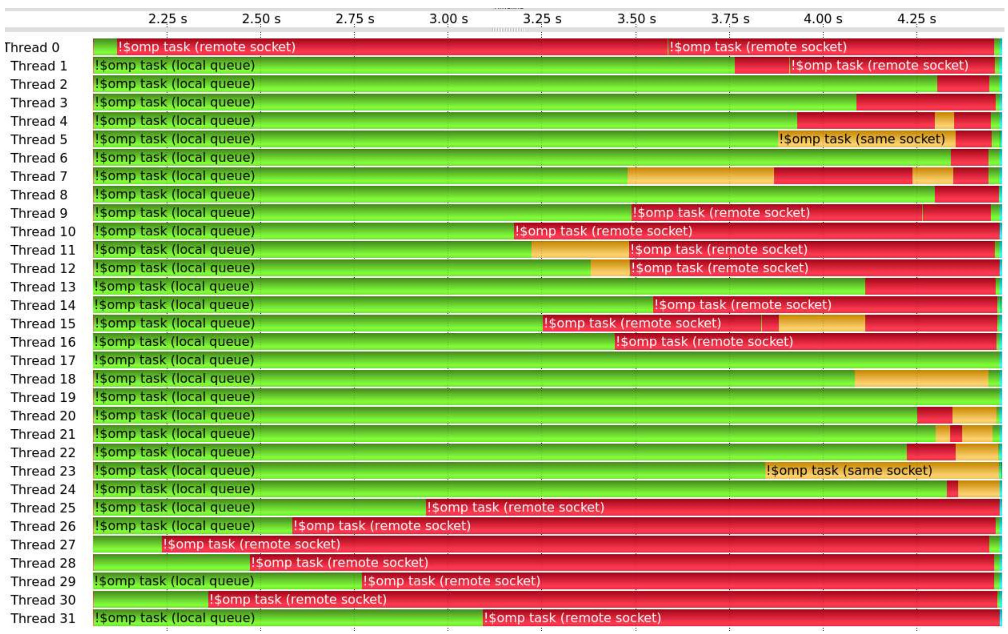

2.25 s
2.50 s
2.75 s
3.00 s
3.25 s
3.50 s
3.75 s
4.00 s
4.25 s
Thread 0 !$omp task (remote socket) !$omp task (remote socket)
Thread 1 !$omp task (local queue) !$omp task (remote socket)
Thread 2 !$omp task (local queue)
Thread 3 !$omp task (local queue)
Thread 4 !$omp task (local queue)
Thread 5 !$omp task (local queue) !$omp task (same socket)
Thread 6 !$omp task (local queue)
Thread 7 !$omp task (local queue)
Thread 8 !$omp task (local queue)
Thread 9 !$omp task (local queue) !$omp task (remote socket)
Thread 10 !$omp task (local queue) !$omp task (remote socket)
Thread 11 !$omp task (local queue) !$omp task (remote socket)
Thread 12 !$omp task (local queue) !$omp task (remote socket)
Thread 13 !$omp task (local queue)
Thread 14 !$omp task (local queue) !$omp task (remote socket)
Thread 15 !$omp task (local queue) !$omp task (remote socket)
Thread 16 !$omp task (local queue) !$omp task (remote socket)
Thread 17 !$omp task (local queue)
Thread 18 !$omp task (local queue)
Thread 19 !$omp task (local queue)
Thread 20 !$omp task (local queue)
Thread 21 !$omp task (local queue)
Thread 22 !$omp task (local queue)
Thread 23 !$omp task (local queue) !$omp task (same socket)
Thread 24 !$omp task (local queue)
Thread 25 !$omp task (local queue) !$omp task (remote socket)
Thread 26 !$omp task (local queue) !$omp task (remote socket)
Thread 27 !$omp task (remote socket)
Thread 28 !$omp task (remote socket)
Thread 29 !$omp task (local queue) !$omp task (remote socket)
Thread 30 !$omp task (remote socket)
Thread 31 !$omp task (local queue) !$omp task (remote socket)

Figure 5.12: Vampir screenshot showing the tasks during the execution of the load balancing benchmark with the Intel runtime. Tasks executed on the creating thread ar shown in green, stolen tasks from the same NUMA-node in orange and from remote NUMA-nodes in red.

lot of threads execute local tasks (green) most of the time and only a few threads execute remote tasks (red) once. Since some of the threads did not have a lot of local work, this was the best scheduling model as discussed in section 4.

If the same benchmark is executed with the GNU compiler, the performance is much worse. The reason is that the GNU runtime uses a centralized task queue which is not able to maintain any task locality. For a programmer without this knowledge it might be confusing that the performance changes that much between these compilers. With the proposed improvements for performance analysis as they are implemented in the parser tool, the analysis of the code for the GNU compiler looks like shown in figure 5.13. Here a lot of red tasks are executed which means a lot of tasks from remote sockets are executed, which leads to many remote accesses and a slower execution time. An interesting fact is that most threads execute local tasks at the beginning. This is because the task creation first puts tasks in the centralized task queue for later execution. After the queue is filled, tasks are executed immediately and not put into the queue. This of course leads to local execution. When the tasks are then executed from the task queue the chance to execute a local one is 1/32, the chance for orange is 7/32 and the chance for red is 24/32. When I zoom in to omit the phase when the task queue is filled and all tasks are executed locally at the beginning of the trace, the distribution shown in table 5.3 is observed. Obviously the measured values match pretty well with the

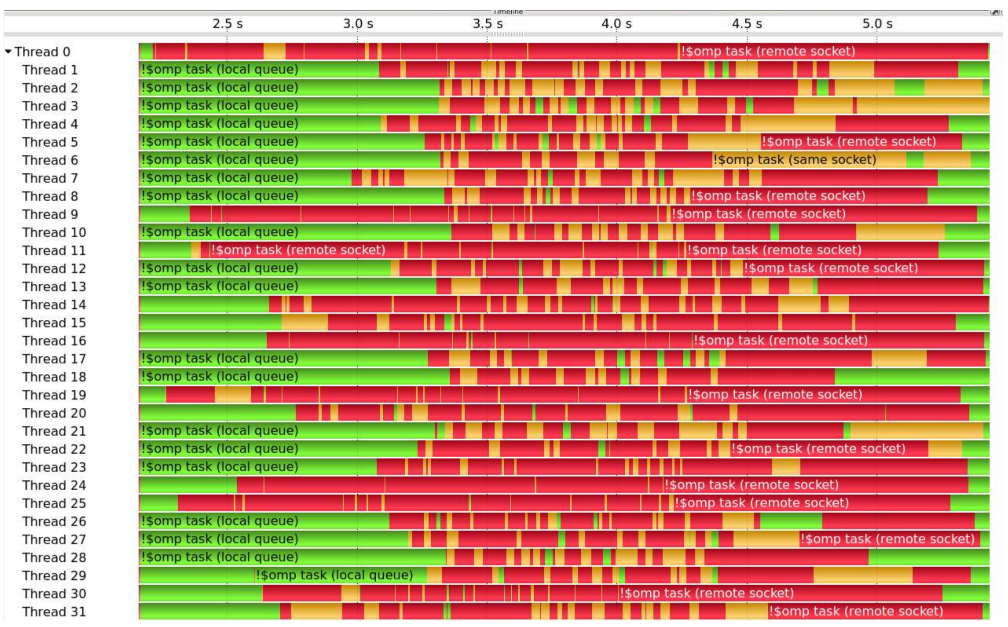


Figure 5.13: Vampir screenshot showing the tasks during the execution of the load balancing benchmark with the GNU runtime system. Tasks executed on the creating thread are shown in green, stolen tasks from the same NUMA-node in orange and from remote NUMA-nodes in red. Comparing this figure to figure 5.12, a huge difference in the way tasks are scheduled in the GNU and Intel runtime can be observed.

expectations which indicated that the assumptions on the scheduling have been correct.

access	time [sec]	fraction measured	fraction calculated
local	1.761	3.2%	3.1%
same socket	12.339	22.5%	21.9%
remote socket	40.838	74.3%	75%

Table 5.3: Fraction of tasks which have been executed locally, on the same socket or remotely for the GNU runtime system, compared to the expected ratio

This information on task scheduling can help programmers to understand the performance of a tasking program on NUMA machines, so it is beneficial if a performance tool can present the data as shown above.

5.5 Summary

In this chapter, I investigated performance issues in task parallel programs, and discussed four typical performance issues in OpenMP task-parallel programs. Too coarsely and too finely grained tasks, a bottleneck in task creation if a single thread creates all tasks, and NUMA-unaware scheduling of tasks in case of parallel task creation used with the intention to maintain NUMA locality. Sampling-based tools from Intel and Oracle were used to investigate the Sudoku solver example application. The results identified the bad efficiency of the parallelization, but they failed to highlight the problem clearly. The event-based performance analysis with Score-P, gave more detailed information which was able to detect three out of the four mentioned problems. For the last problem, the NUMA-unaware task scheduling, information about the hardware and thread placement was missing. I showed with a Score-P trace rewriting tool how this information could be used to detect the last-mentioned performance problem. Here, a separate mapping file was generated which contained the missing information about the placement and hardware. Then the rewriter combines the information from the trace and the mapping file and generates a good overview of the task scheduling done by the OpenMP runtime system.

6 A Workflow to Program Large SMP Machines

OpenMP provides an attractive approach for beginners in parallel programming as it allows for incremental parallelization of an existing application and does not require a complete rewrite of the code. Subsequent tuning can improve the performance by employing techniques taught, for example, in books like "Using OpenMP" [Chapman et al., 2007] or in an OpenMP tutorial. However, more advanced tuning steps as they are described in this thesis are often not covered.

The question of interest investigated in this chapter is if it is possible to specify a step-by-step guide that directs the programmer towards a successful shared memory parallelization of his code for large NUMA systems. Such a guide will enable performance engineering, defined as structured and methodical process of developing highly efficient HPC applications, similar to [Hager, 2013]. A structured approach to optimize applications will enable programmers, which are not experts in performance optimization, to ensure sensible performance of their codes.

This chapter contains a description of a performance engineering workflow for OpenMP programs on large NUMA systems. As part of performance engineering, a performance model is presented and all steps are accompanied with an example. Finally, the chapter is completed with two application case studies where application tuning for large NUMA systems was applied.

The work presented is partially based on former publications. A basic version of the workflow presented here was published in [Schmidl et al., 2013d] and the tool to visualize remote traffic described during the data layout section in the workflow was published in [Weyers et al., 2014]. The roofline model was applied to the CG solver used later in this chapter in [Cramer et al., 2012], but the improved performance model presented here has not been published before. Furthermore, the case studies for SHEMAT-Suite and TrajSearch have been published partially in [Schmidl et al., 2010b], [Berr et al., 2012], [Schmidl and Vesterkjær, 2014] and [Schmidl et al., 2015].

6.1 Tools-guided Performance Tuning on Big SMP Machines

6.1.1 Tuning Cycle

Performance tuning of an application is a continuous process, which aims to achieve better and better performance. Performance analysis tools can be used as part of this process. In such cases it is common practice to repeat several general steps, which are often called "tuning cycle".

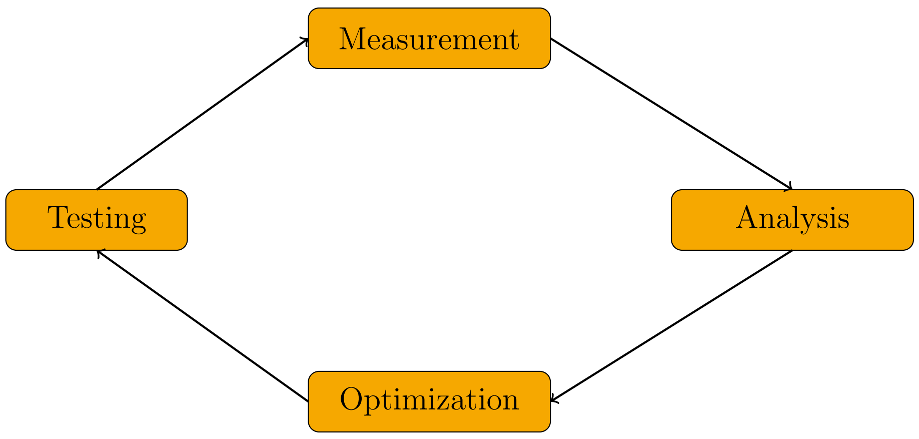


Figure 6.1: Illustration of the tuning cycle

The tuning cycle is depicted in figure 6.1 and consists of the following steps:

- **Measurement:** First, performance analysis tools are used to gather data of the execution of the program. Different kinds of data can be gathered here, ranging from execution time for specific functions to hardware counter data e.g. for TLB misses during execution. The data can be gathered in different ways, like sampling or event based, as described in section 5.2 for tasks as an example.
- **Analysis:** Second, the performance data is analyzed. Here, several tools exist to visualize the data, e.g. in a timeline browser of the Vampir GUI, showing the program execution over time, or annotated in the source code as in VTune, where e.g. the cache misses can be shown at the source lines where they occurred. Also tools exist which try to analyze the data automatically and lead the user to performance problems, like the Scalasca tool [Geimer et al., 2008]. During this step the user should be able to understand the reason of a performance loss in the program.
- **Optimization:** During the optimization step the user changes the code to circumvent the performance bottleneck. Typical optimization steps include,

for example, blocking to optimize the cache usage or rebalancing the load to achieve a better distribution of work to threads.

- **Testing:** As a last step, the changes are tested to ensure that the results are still correct and to check if the optimization really increased the code performance.

Since it is hard, most of the time even impossible, to identify all performance issues of an application all in once, these four steps are repeated in a cycle as mentioned before.

Finally, the question when to stop performance tuning is interesting, since this cycle can be repeated basically forever, but the performance gains typically get smaller and smaller. The two main reasons to stop tuning are:

(i) When an application has to reach a certain performance criteria, it is enough to optimize until this criteria is reached. A common example are real-time simulations, where an application already achieves real-time performance, there is no need to optimize and get any faster.

(ii) Many applications do not have such a limit. In many areas, getting results faster is always beneficial, e.g. because it allows more scientific results or it can speed up a development process in industrial areas. For such codes often performance modeling is used to determine the theoretical best performance of a code on an given architecture, as for example done in [Hager and Wellein, 2010]. Then performance tuning can be stopped, when a certain relative performance is reached, like 90% of the best possible performance. Where exactly this boundary is set, is up to the programmer.

The cycle mentioned can be used, as a guideline, but it does not answer the question which data shall be gathered, how it should be analyzed and how the applications can be optimized. These questions, of course, depend on the code and on the hardware architecture. Next, I will present some performance issues which frequently occur with OpenMP programs on large shared-memory machines. The tuning steps to apply are those presented in previous chapters.

6.1.2 Investigated Issues

A recipe with a list of all issues to investigate on a large NUMA machine and an order in which they shall be handled cannot be given in general. Depending on the code, certain issues are more significant than others and must be handled first, since they hide other less significant issues in the analysis. Also optimization steps which fix one issue can introduce a different one into the code by accident. Here, I will list a set of typical issues to investigate for OpenMP programs on large NUMA systems. For all of these issues an iteration of the tuning cycle is useful to investigate if the issue is relevant in the investigated source code or not. If present, the issue should be fixed and the resulting code should be investigated for other occurrences of the same issue again, until the issue does not occur anymore. The reason is that one issue can be present multiple times in a code. I advise to start checking for the issues in the order they are listed below. However, as mentioned before, it might be necessary to go back and forth in the list to find all relevant issues.

Hotspot Analysis and Parallelization Ratio

There are good reasons for the well established practice to start a performance analysis by determining the hotspots of a program. Hotspots are regions in the code where a significant amount of execution time is spent. These hotspots are then target for further optimization steps. The reason for this is grounded in Amdahl's law ([Amdahl, 1967]), which gives a limit for the speedup gained by optimization steps. Also, while Amdahl's law is often only applied for parallelization, it holds for any optimization technique applied to a code region *opt*. When the region *opt* takes a fraction T_{opt} of the overall execution time, then the speedup for any optimization applied to this region is limited by:

$$Speedup \leq \frac{1}{(1 - T_{opt})} \tag{6.1}$$

So, it is more beneficial to optimize code regions, which consume a high fraction of the execution time. Identifying those regions can be done with most performance tools and it only requires investigation of the execution time and no special hardware performance counters. However, sampling-based tools typically deliver a higher accuracy, since they can often map the execution time to source lines if debug symbols are present. For example, the "Hotspots Analysis" of the Intel VTune Amplifier XE delivers hotspots of the code on a source code level.

The hotspots analysis is often a good starting point in performance analysis, but it only delivers insights where to look in further iterations of the tuning cycle. Normally this step does not spot a performance issue which can be optimized in a straight-forward fashion.

However, the data gathered during the hotspots analysis can be used in the analysis step of the tuning cycle to spot a performance issue for OpenMP programs. The time information gathered allows to determine the parallelization ratio of the code, i.e. the fraction of the code which is parallelized with OpenMP. This fraction allows the programmer to calculate the maximum possible speedup based on Amdahl's law. If a certain speedup is desired, this analysis allows to calculate the number of hotspots which at least need to be parallelized to make this speedup possible. Otherwise, a user can start parallelizing the most relevant serial hotspot and continue this procedure to reduce the serial fraction. Since one advantage of OpenMP is that it can be incrementally added to the code, it is very suitable for such an approach.

Synchronization Overhead and Locking

As shown before, synchronization and locking can be orders of magnitude more expensive on large NUMA machines with hundreds of OpenMP threads compared to standard two socket servers. Therefore, codes which scale and run well on standard machines tend to run into this issue when hierarchical NUMA systems are used. These issues can be caused by using OpenMP synchronization constructs like critical regions or barriers. If this is the case, performance tools will show a high execution time in those regions. Here, event-based measurement tools, like Score-P, or sampling-based tools can be used. Both types will identify time spent in barriers or critical regions. For critical regions, the time spent in the overall construct and in the inner region can be measured by tools. The difference between both time spans is the time to process the critical construct and waiting time to enter the critical region. If this time is significant, a performance problem is present, since threads waste time waiting to enter the region.

Optimization steps to circumvent these problems can be to avoid barriers if they are not necessary, e.g. by using a `nowait` clause. For critical regions, an `atomic` construct can be used as a replacement sometimes. This allows an implementation to use atomic hardware operations instead of locks, which are much faster. If this is not the case, often buffers can be used to reduce the amount of critical writes and then bundle all writes of a buffer in one critical region. Although these steps are well-known optimization methods for OpenMP programs, their relevance on large NUMA systems is significantly higher compared to standard servers. This is because the overhead can be orders of magnitude higher on these systems as was shown in section 2.2.1.

Other sources of locking are the explicit use of lock routines or functions which require locking internally. One of those functions is `malloc` which is very expensive on large NUMA systems as described in section 2.1.4. If one of those functions appears in a hotspot, it should be investigated if the use of locks can be reduced here. For explicit locks, often the granularity of the locking can be adjusted, e.g. by locking elements of a data structure instead of the complete data structure. This requires more locks, but the contention typically is reduced and thus less time is spent waiting. For `malloc` calls which occur frequently, a replacement of the allocation library can help, e.g. to `kmp_malloc` as also discussed in section 2.1.4.

Data Layout

As discussed earlier, remote accesses are far more expensive on large NUMA systems compared to standard servers. Therefore, it is much more important that the data layout is done in a way that remote accesses are avoided if possible. As explained in detail in chapter 3, the data layout can be influenced in different ways. The `first-touch` policy of the OS can be used to achieve a certain data layout by parallel initialization of the data. Furthermore, different memory migration techniques have been discussed.

Before the data layout can be optimized, it must be detected where the data layout is not optimal. This is a tough task and to the best of my knowledge no tool exists which can answer this question in a simple way. A strong indication that the data layout is not optimal is the presence of many remote accesses in a code region. The amount of remote accesses can be investigated with hardware performance counters on most architectures. For example, the counter `MEM_UNCORE_RETIRED:REMOTE_DRAM` on the Westmere system delivers the number of memory accesses which have been served from a remote node. Their relevance can be better understood, if they are seen in relation to:

- (a) the local memory accesses, to get a ratio of remote accesses, or
- (b) the maximum possible number of remote accesses which can be served by the system in the given time.

Option (a) is typically easier to realize. On the Westmere system, it requires to measure also the counter `MEM_UNCORE_RETIRED:LOCAL_DRAM` to calculate the ratio. If the user then examines a high ratio of remote accesses, e.g. 50%, this indicates that the data layout is not as expected and should be optimized. Of course, for some data access patterns, e.g. random accesses in a shared array, a high number of remote accesses cannot be avoided. These issues must still be investigated by the programmer, but the hardware counters point the programmer to locations in the code which should be investigated.

For option (b), we developed a tool which was presented in [Weyers et al., 2014]. The focus of this work is on data visualization, which has been done by Weyers and Herber.

My contribution is the hardware counter analysis in relation to the possible maximum bandwidth a link can deliver. I measured the amount of transfers over all QPI links and memory links on a BCS system. Furthermore, I measured the limit of all those links, i.e. to the local memory, to a socket on the same board and to a remote board. Investigating the hardware counters then allows to determine if the usage is close to the theoretical limit of the link. If this is not the case, it indicates that this link is a performance bottleneck and optimization might be required.

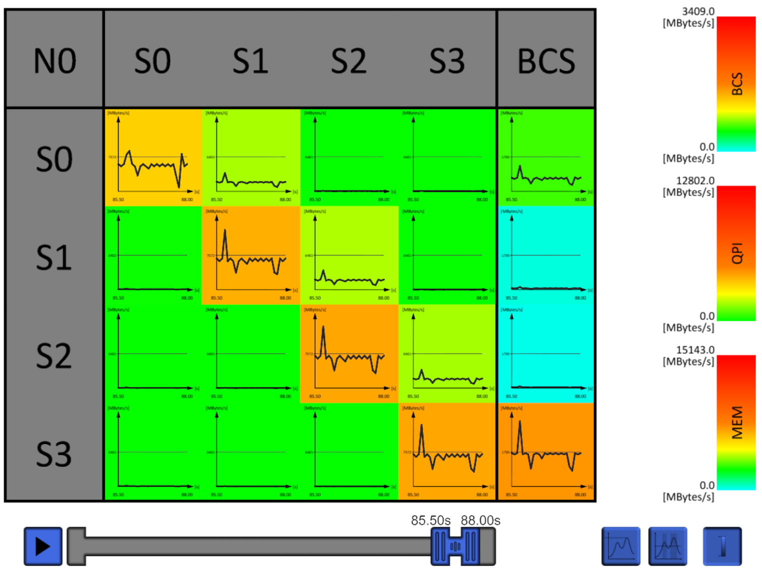


Figure 6.2: Screenshot of the visualization tool for remote accesses on the BCS system.

Figure 6.2 shows the traffic over all links of one board in a BCS system. For every link, a diagram is shown with the traffic (measured with hardware counters) shown over time. On the diagonal the memory traffic of the sockets S1, S2, S3 and S4 are shown. The elements (Sx,Sy) for $1 < x, y < 4$ and $x \neq y$ in the matrix show traffic of the QPI links on the board. The right column shows the traffic to the BCS link. All backgrounds are color coded with different scales, shown at the right. Since the memory controller can deliver about 15 GB/s, the range for those entries is 0 - 15 GB/s, whereas it is 0 - 3.4 GB/s for the BCS links, since the link cannot deliver more than this bandwidth. Overall this tool shows that remote traffic hardware counters can be analyzed in a useful way, if they are set into relation to the maximum capabilities of the system (option (b)).

If it turns out that too many remote accesses occur, the techniques presented in chapter 3 for memory placement and migration can be used to optimize the data layout.

Load Balancing

As discussed in detail in chapter 4, load balancing is a challenging task in parallel programming. If the work is not evenly distributed across all threads of a team, some threads have to wait at the next barrier for the thread with the longest execution time. Common reasons for load imbalance in OpenMP programs on NUMA systems are:

- a static distribution of work which maps more work to some threads, i.e. some threads have more instructions to execute.
- different memory access time to needed data for some threads, i.e. some threads have more remote accesses and therefore need a longer time frame to execute the same amount of instructions due to longer waiting times for the data.

In both cases load imbalance can be investigated with performance tools if the execution time of all threads is compared. If load imbalance occurs during execution, the execution time of the code region differs between threads, furthermore the waiting time in the barrier differs exactly in the opposite direction. A user therefore can compare the waiting time in a barrier or the execution time of regions for all threads in a team to detect load imbalance.

When load imbalance is detected, it can be avoided e.g. with the techniques presented in chapter 4.

Applying the tuning cycle for all these issues will optimize OpenMP applications for large NUMA systems. As mentioned before, every performance issue can occur multiple times in different parts of the application and fixing one issue might result in a different issue which was not visible before. Therefore, it usually is beneficial to repeat these steps multiple times. When to stop the tuning cycle is a hard question which can, for example, be answered with performance modeling as shown later in this chapter.

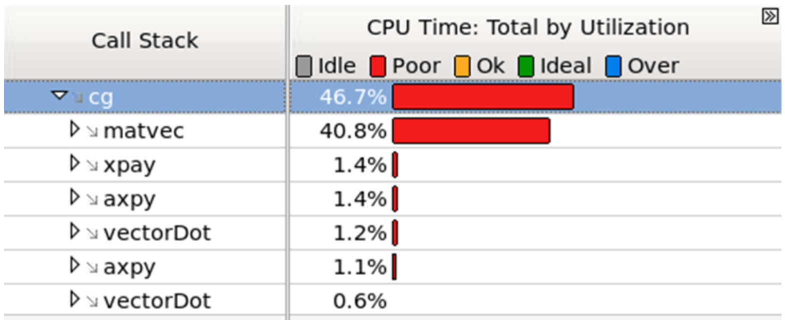
6.1.3 Conjugate Gradient Method

As an example how the tuning workflow can be applied, I use a conjugate gradient (CG) solver to optimize for a BCS system in this section. CG solvers are a standard iterative method to solve sparse linear equation systems and they are often time consuming parts of application codes. Therefore, the kernel is of high relevance in high-performance computing.

The serial version of the solver taken here was developed by Tim Cramer for an exercise in parallel programming and the example matrix was taken from the Florida Sparse Matrix Collection [1]. This collection is a set of real-world matrices which offer a realistic sparsity pattern.

As described in the workflow above, I started the first iteration of the tuning cycle with a **hotspots** analysis. I used the Intel VTune Amplifier XE here, but any other tool discussed before should deliver similar results for these tests. Figure 6.3 shows a screenshot of the main hotspots of the kernel. The fact that the CG kernel

[1]https://www.cise.ufl.edu/research/sparse/matrices/



Call Stack	CPU Time: Total by Utilization
▽ cg	46.7%
▷ matvec	40.8%
▷ xpay	1.4%
▷ axpy	1.4%
▷ vectorDot	1.2%
▷ axpy	1.1%
▷ vectorDot	0.6%

Figure 6.3: Hotspots of the CG solver investigated with Intel VTune.

only consumed 46% of the overall execution time is because the matrix must be read from disk in the kernel and the test environment contains error checks at the end which consume some time. If the CG is applied in a simulation code, those steps are of course not needed and therefore I only focus on those 46% of the test execution.

Hotspots detected in the kernel are the matrix-vector multiplication (`matvec`) and several vector additions and dot-product operations (`xpay`, `axpy` and `vectorDot`). The matrix-vector operation is by far the most dominant hotspot consuming 40.8% out of the 46.7% of the CG execution. VTune furthermore shows that the overall utilization of the machine is poor, meaning not all parallel resources are used. Since I started with a serial version, this is obvious, but this analysis detects the utilization per hotspot. So in a real application it can be used to find hotspots which need to be parallelized.

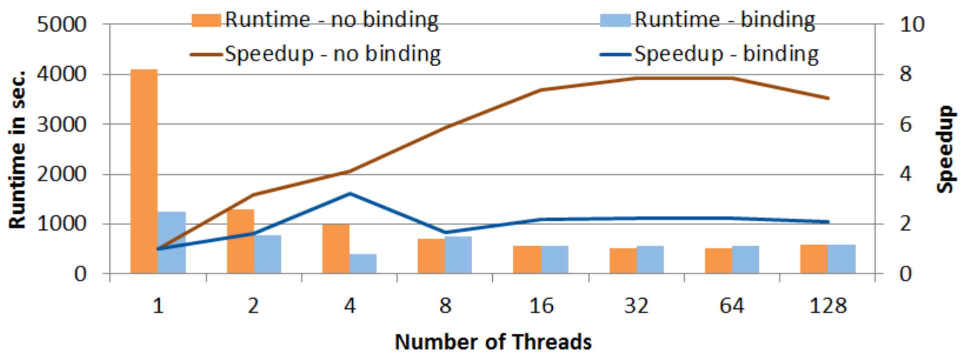
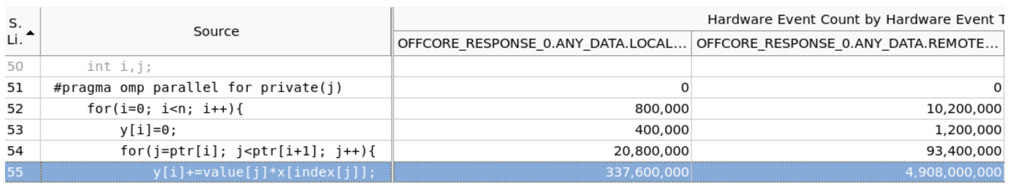


Figure 6.4: Performance reached with the CG solver on the BCS system after parallelizing hotspots with and without thread binding.

To fix this issue, I parallelized all hotspots with a parallel loop and used the reduction operation within the dot product. This led to the performance shown in figure 6.4, once measured with and without binding of threads to cores. Binding threads delivers overall better performance. But, in total, the speedup of the CG solver is very limited in both cases.

Next, I analyzed the **data layout** of the code using the VTune Amplifier. On the 16-socket BCS system, the counters `OFFCORE_RESPONSE_0.ANY_DATA.LOCAL_DRAM` and `OFFCORE_RESPONSE_0.ANY_DATA.REMOTE_DRAM` were used to measure the amount of memory accesses served from local or remote memory.



S. Li.	Source	Hardware Event Count by Hardware Event T	
		OFFCORE_RESPONSE_0.ANY_DATA.LOCAL...	OFFCORE_RESPONSE_0.ANY_DATA.REMOTE...
50	`int i,j;`		
51	`#pragma omp parallel for private(j)`	0	0
52	`for(i=0; i<n; i++){`	800,000	10,200,000
53	`y[i]=0;`	400,000	1,200,000
54	`for(j=ptr[i]; j<ptr[i+1]; j++){`	20,800,000	93,400,000
55	`y[i]+=value[j]*x[index[j]];`	337,600,000	4,908,000,000

Figure 6.5: Local and remote memory accesses of the matrix vector multiplication in the CG solver measured with Intel VTune.

Figure 6.5 shows the results for the matrix-vector multiplication. It can be observed that about 93% of all memory accesses are served from remote DRAM on the 16-socket system. The reason is that the matrix is not distributed across the NUMA nodes. Since the initialization was no hotspot it has not been parallelized and the data was placed by the OS on the NUMA node of the master thread. Some vectors involved in the CG were used the first time to store results during the execution of the CG solver and they are therefore already distributed, but for the large matrix only the eight threads on the first socket have local accesses whereas all other threads access it remotely. As mentioned in chapter 3 there exist several options to optimize the data layout, the easiest one in such a static case is to initialize the data in parallel, which has been done here for the matrix setup. Hereafter, the performance improved significantly as shown in the results in figure 6.6.

Next the **load balance** of the code is investigated. All vector operations show equal execution times on all threads. In the matrix-vector operation the overhead spent in the OpenMP runtime is noticeably high. As shown in figure 6.7, the accumulated time spent in the runtime system is about 10 seconds.

The total execution time in the parallel region is about 33 seconds, so about 25% of the overall time is overhead. The red background color in VTune indicates that this is detected as possible performance issue. Some overhead to create threads and synchronize in a parallel region is typical. But all parallel regions executed in the vector operations, which are also executed once per iteration in the solver produce less than one second of overhead and not 10 seconds. This indicates some load imbalance in the matrix vector operation. This is due to the sparsity pattern of the matrix. The rows are distributed evenly across all threads, but the load is proportional to the number of non-zero elements in the sparse matrix, not

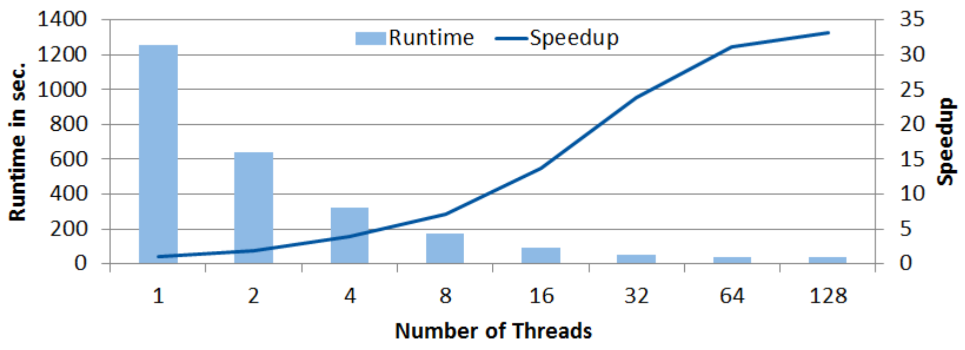
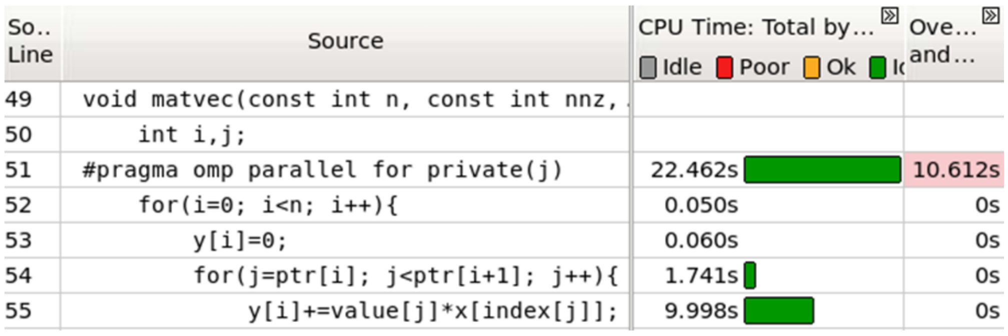


Figure 6.6: Performance reached with the CG solver on the BCS system after optimizing the memory access pattern by distributing the matrix across NUMA nodes.



So.. Line	Source	CPU Time: Total by... ⬜ Idle 🟥 Poor 🟧 Ok 🟩 I	Ove... and...
49	`void matvec(const int n, const int nnz,`		
50	`    int i,j;`		
51	`#pragma omp parallel for private(j)`	22.462s 🟩	10.612s
52	`    for(i=0; i<n; i++){`	0.050s	0s
53	`        y[i]=0;`	0.060s	0s
54	`        for(j=ptr[i]; j<ptr[i+1]; j++){`	1.741s 🟩	0s
55	`            y[i]+=value[j]*x[index[j]];`	9.998s 🟩	0s

Figure 6.7: Overhead in the sparse matrix vector operation in the CG solver, measured with Intel VTune.

proportional to the number of rows. If some rows contain more non-zero elements, threads executing those rows have more work to do. As discussed in section 4.1, `dynamic` or `guided` schedules do not work well on NUMA systems. Therefore, I computed a static distribution of rows during the matrix setup in a way that all chunks contain roughly the same amount of non zero elements. The distribution stores a start and end row for each thread. This distribution is then used during the setup and during the later computation. So, the data distribution fits to the computational access pattern in the CG solver.

Figure 6.8 shows the performance reached by the optimized version on the BCS system. The CG solver now scales well over the complete BCS system and reaches a speedup of about 45 with 128 threads. This example shows that the standard techniques presented in the optimization workflow work well for the CG kernel to optimize the code for large NUMA systems. Furthermore, it has been seen that tools can play an important role in this process since they help to identify performance problems in an easier way. Later in this chapter, in section 6.3, I will

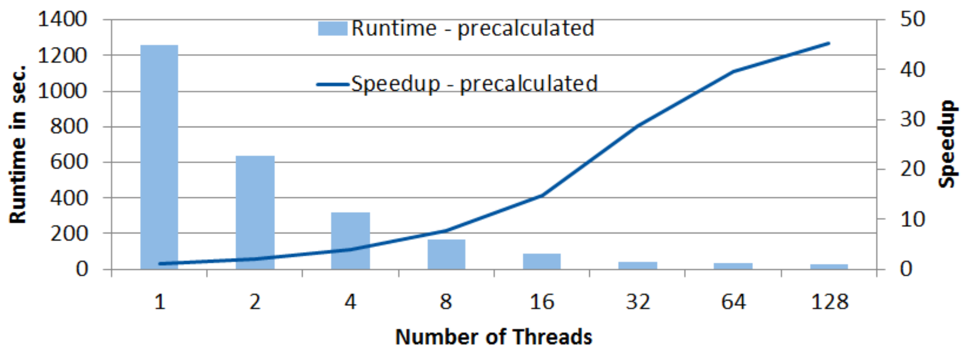


Figure 6.8: Performance reached with the CG solver on the BCS system after optimizing the memory access pattern by distributing the matrix across NUMA nodes.

present case studies that show also real world applications face similar problems and can be optimized for NUMA systems in a similar fashion.

6.2 Modeling OpenMP Performance

As already explained when the tuning cycle was described in section 6.1.1, performance modeling can be used to determine the point at which performance tuning should be stopped, because the performance is close to the optimum for a specific algorithm on a given architecture. Since modern architectures, parallel programming libraries and runtime systems, as well as parallel algorithms are complicated, a precise model, covering all details, is usually not feasible. A model should predict performance close to the reality, be simple enough to be understood by a user, and be applicable to real algorithms and applications. Here, I will first cover existing models and their shortcomings for OpenMP programs on large SMPs, and then I will present a model which overcomes these shortcomings. To conclude the discussion on performance modeling, I will apply the example to the parallel CG solver tuned above and show that the reached performance is close to the modeled performance.

6.2.1 Performance Models

Several performance models exist, which can be used for serial programs, like the roofline model presented e.g. by William et al. [Williams et al., 2009] or a model from Treibig and Hager for bandwidth-limited loop kernels [Treibig and

Hager, 2010]. The roofline model is a simple model which is based on fundamental hardware characteristics, i.e. peak performance and memory bandwidth limitations, and ignores, for example cache hierarchies. This model is relatively easy to apply, and delivers good results for serial applications where the number of memory transfers required and the floating point operations required can be determined. The model presented by Treibig and Hager is more complicated to apply, since it requires a detailed analysis of different cache levels in the system and the need to analyze the memory access in a very detailed fashion to determine, which cache level can hold the data for the algorithm. Both models are made for serial programs and Treibig and Hager write in their publication that shared memory applications would require extensive extensions of the model which are beyond the scope of their work. The roofline model can be applied also to shared-memory programs, but still it only covers bandwidth limitations and the peak performance of a system, so effects like synchronization overhead and communication are not covered.

Especially for parallel applications with MPI, models exist which cover the communication in detail, like models from [Culler et al., 1993] or [Ramos and Hoefler, 2013]. These models cover communication and network characteristics in detail. For MPI applications these models are very useful, since it is easy to identify the communication in an application. For OpenMP, no explicit communication calls exist, since the shared memory is used by all threads to exchange data. So, these models cannot be extended to also cover OpenMP parallel programs.

The model I will propose later in this chapter to cover OpenMP programs on large NUMA systems can be seen as an extension of the roofline model. Therefore, details of the roofline model are given first in the next paragraph.

Roofline Model

As already mentioned, the roofline model is based on the peak performance of a system and the memory bandwidth it can reach. Whether a kernel is limited by one or the other of these limitations depends on the number of floating point (FP) operations ($fops$) executed in the kernel and the amount of memory transfers in bytes ($memtrans$) performed in the kernel. Based on the operational intensity, defined as $OperationalIntensity = \frac{fops}{memtrans}$, the roofline model can be visualized as a diagram, as shown in figure 6.9 for the BCS system as an example. The peak floating point performance of the system is 1024 GFLOPS. This means no application will ever exceed this performance. In the diagram this is indicated by a vertical line at 1024 GFLOPS labeled with ”Peak FP Performance”. The maximum reachable memory bandwidth can be measured with a benchmark. The roofline model uses the STREAM benchmark here, which delivers a maximum bandwidth of 260 GB/s for the BCS system. The line labeled ”Max. bandwidth” covers all points in the diagram where 260 GB/s of data is transferred, dependent on the operational intensity. Both lines cannot be exceeded by an application of a given operational intensity, since the machine can either not load the data faster, if the "Max. bandwidth" line is the limit, or the machine cannot compute more floating point results, even if it could still load more data from memory.

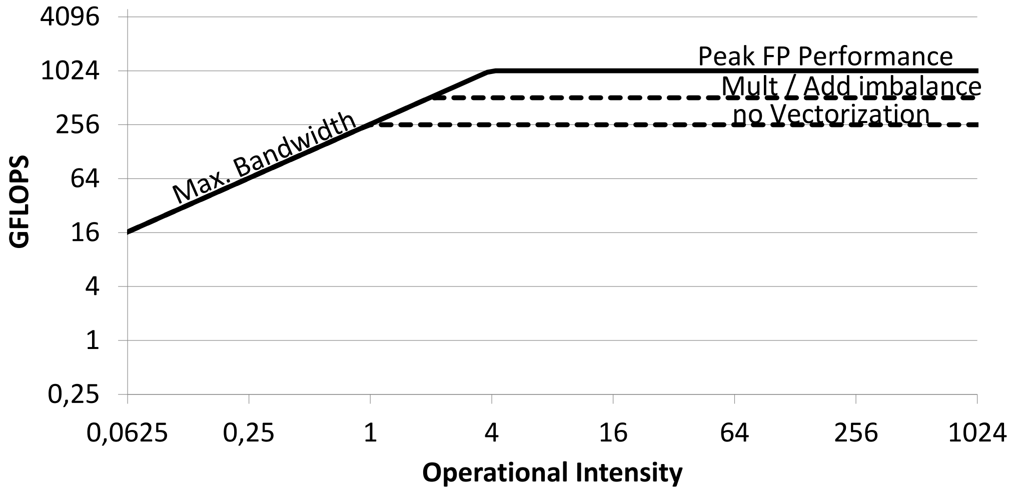


Figure 6.9: The Roofline Model applied to the BCS system.

Applications might not be able to reach one of the limits for different reasons. For example modern architectures use fused multiply-add operations, which means that two FP operations, one add and one multiply, can be done at once. If the algorithm does not require one add per multiplication, this feature cannot be used and the processor only delivers half of the peak performance (512 GFLOPS on the BCS system). Another feature which needs to be applied is SIMD vectorization. Here, a processor applies the operations on a vector of elements instead of a single element. If this feature cannot be used, e.g. because the algorithm does not work on consecutive elements in memory, another factor of two is lost in performance and the algorithm can only reach a fourth of the peak performance (256 GFLOPS on the BCS system). These limitations are illustrated by dashed lines in the diagram of the roofline model.

The diagram only describes the hardware characteristics. For a specific algorithm, a user can determine the operational intensity by counting the number of memory accesses and floating point operations, and check in the diagram if the kernel is memory or floating point limited. Furthermore, a user can compare the upper limit with the performance reached to check how close the implementation is to the optimum.

As mentioned before, the model is relatively simple to apply, but for OpenMP programs it sometimes is too simple, especially on large NUMA systems. Therefore, I will present an extended model in the next section which covers more details of the OpenMP execution, without getting too complicated to be practical.

6.2.2 A Performance Model for large SMPs

The performance model presented here is also based on the achievable memory bandwidth of a system, if the application is memory bound, but it takes a few more aspects into account. Since the roofline model was designed for serial programs, it does not take multithreading into account at all. However, it can easily be extended to reflect the memory bandwidth and peak performance which can be reached with a certain number of threads on a system.

In my model I also take the following two factors into account, which are significant on large NUMA systems:

(i) The peak memory bandwidth of a system is typically measured with the stream benchmark for large memory footprints of several GB. For smaller arrays this bandwidth typically cannot be reached, for two reasons. First, prefetching units in the core need to fill their pipeline to work efficiently and second, identical cachelines are shared between threads which then need to be transferred between the cores. For large arrays sharing these cachelines and filling the pipeline has a much smaller influence than for small arrays. Therefore, the model presented here will not take the best memory performance of the system into account but the best performance for a certain memory footprint.

(ii) In OpenMP programs, the synchronization overhead can be relevant for the overall performance. As shown in section 2.2.1 with the help of the EPCC microbenchmarks, this can be orders of magnitude more significant on large SMP machines than on standard servers.

These two factors can be used easily by measuring the STREAM bandwidths for different memory footprints and the synchronization overhead with the help of the EPCC microbenchmarks. For an application, the user then calculates the time needed to load the data and synchronize the threads. Both values are then added to calculate the predicted runtime for a memory bound application.

6.2.3 Model Description

The model can be described formally as an extension of the roofline model. In the roofline model, we can calculate the runtime T with formulas 6.2, 6.3 and 6.4.

In 6.2, *memtrans* is the amount of data which needs to be accessed in memory in the algorithm and t is the number of threads used. The time to transfer the

$$T_m = memtrans \times bw(t) \tag{6.2}$$

$$T_c = fops \times peak(t) \tag{6.3}$$

$$T = \max(T_m, T_c) \tag{6.4}$$

data for the algorithm T_m is then computed by multiplying m with the peak bandwidth which can be reached with t threads, $bw(t)$. The time to calculate all floating point operations T_c is described in formula 6.3 as the product of all floating point operations needed by the algorithm $fops$ and the peak performance of the architecture when t threads are used $peak(t)$. The time for the algorithm is then the maximum of T_m and T_c, since either the memory bandwidth or the computation time limits the algorithm's performance in the roofline model. In the roofline model the function $peak()$ is the calculated maximum performance of the machine and the function $bw()$ can be measured by using the STREAM benchmark.

The model I present in this work extends the roofline model in two ways as described in the formulas 6.5 - 6.8. The first extension is, that T_m now takes the size of the arrays into account. As shown in chapter 2, the bandwidth reached on large NUMA systems highly depends on the memory footprint, therefore it cannot be assumed, that the maximum bandwidth can be reached for all array sizes. Here, I calculate the sum over all accessed arrays in the algorithm and multiply the size of the array a with $bw(t, sizeof(a))$ which is the bandwidth which can be achieved for t threads on an array the size of a.

The second extension is that synchronization constructs are taken into account in T_s. Here, $sync(t, s)$ is the time a synchronization construct s needs to execute with t threads on the target architecture. s can be one of the following OpenMP constructs `parallel`, `parallel for`, `for`, `barrier` or `reduction`. This can be measured with the EPCC microbenchmarks as it was done in section 2.2.1. For T_m, T_c, and T_s we then obtain the following formulas:

$$\begin{gathered} T_m = \sum_{\forall a \in A} sizeof(a) \times bw(t, sizeof(a)) \\ with\ A = all\ accessed\ arrays \end{gathered} \tag{6.5}$$

$$T_c = fops \times peak(t) \tag{6.6}$$

$$\begin{gathered} T_s = \sum_{\forall s \in S} sync(t, s) \\ with\ S = all\ used\ synchronization\ constructs \end{gathered} \tag{6.7}$$

$$T = \max(T_m, T_c) + T_s \tag{6.8}$$

The overall time T to execute the algorithm is then the maximum of T_m and T_c, for the same reasons as in the roofline model, plus the synchronization time T_s, since synchronization needs to be applied in all cases, whether the algorithm is memory or compute bound.

The changes applied to the roofline model are more relevant on large NUMA systems than on other systems, because the difference in the memory bandwidth for different sized arrays is higher when more NUMA nodes are involved and the synchronization overhead is also much more relevant. To show the usefulness of the model I will next apply it to the CG kernel in the following section.

6.2.4 Example: Conjugate Gradient Method

As mentioned at the beginning of this chapter, the matrix vector multiplication is the main hotspot in the CG solver. Furthermore, it contains two dot products and three scaled vector operations per iteration. The memory footprint in the CG solver is different for the matrix-vector operation and the vector-vector operations. The sparse matrix is stored in compressed row storage (CRS) format and has a dimension of $N = 2,017,169$ and contains $nnz = 283,073,458$ non-zero elements. This results in a memory footprint of about 3.1 GB for the matrix-vector and about 45 MB for the vector-vector operations. The bandwidth which can be achieved for such arrays was measured with the STREAM benchmark. For a small number of threads the bandwidth is between 4.5 and 5 GB/s in both cases, so there is nearly no difference. With 128 threads the BCS machine can reach a transfer rate of 226 GB/s for the transfer of 3.1 GB and 155 GB/s for the transfer of 45 MB. So, the difference is really visible.

Furthermore, the CG kernel needs two reduction operations and six worksharing loops. The overhead for those operations was measured with the EPCC benchmarks for different numbers of threads.

Figure 6.10 shows the resulting predicted performance of the model and the measured performance on the BCS system. The model does not work very well for a small number of threads. The reason is that the performance for only a few threads is not really influenced by the synchronization and the bandwidth differs significantly from the stream bandwidth. A small number of threads cannot utilize the bandwidth completely. Therefore, the performance depends on many more factors like the cache reuse and the number of load and store instructions which can be issued per core and many more. For such cases, the more complicated model from Treibig and Hager [Treibig and Hager, 2010] would be more useful, since it takes for example all cache levels, and the issued instructions into account.

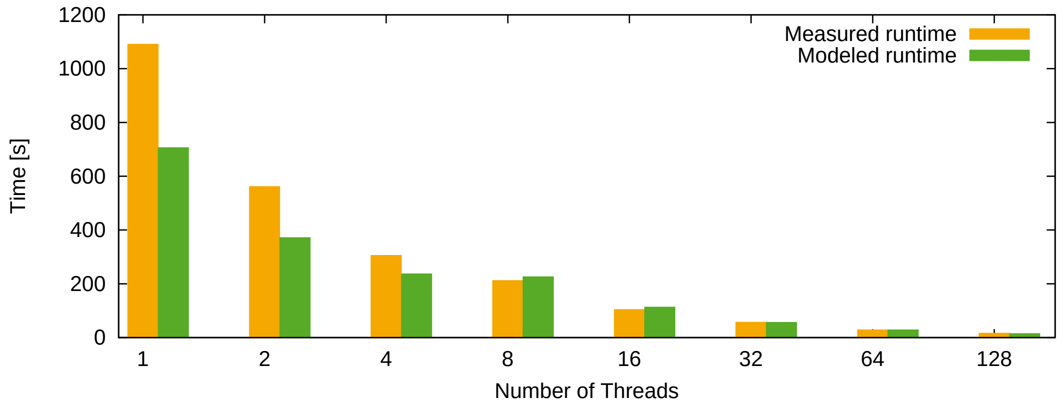


Figure 6.10: Measured and modeled runtime for the CG solver.

However, for a larger number of threads, the model presented here works well. The bandwidth measured depends more on the maximum bandwidth of the system and the data set size. Also the synchronization constructs are more relevant on large machines. For a larger number of threads (32+), the error of the presented model is between 0.7% and 10%. For comparison, the straightforward roofline model is off by 22% for 128 threads. So, the presented model is useful for a large number of threads on big SMP systems and it is comparably easy to apply for users.

6.3 Case Studies

After explaining characteristics of the large NUMA systems and techniques to optimize OpenMP applications for those machines, I will now discuss two case studies to show how real application codes were optimized to obtain good performance on large SMP machines. The first case study has been published at the IEEE Cluster conference 2010 [Schmidl et al., 2010b]. The code, called SHEMAT-Suite, was optimized mainly by Andreas Wolf for a ScaleMP machine. My contribution to the work were insights in machine characteristics and guidance with optimization techniques which were applied in the code. The second code, called TrajSearch, was at the beginning also optimized for a ScaleMP machine as part of the bachelor thesis of Nicolas Berr which I supervised, see [Berr et al., 2012] for details. Later, I added slight modifications to optimize the code further on a Numascale system, see [Schmidl and Vesterkjær, 2014] and [Schmidl et al., 2015]. Both codes show that the tuning advice presented in this thesis can lead to good performance on a large NUMA system.

6.3.1 SHEMAT-Suite

SHEMAT-Suite is a simulation package developed at the Institute for Applied Geophysics and Geothermal Energy at RWTH Aachen University. Like the predecessor SHEMAT [Clauser, 2003] (**S**immulator for **HE**at and **MA**ss **T**ransport) it solves the coupled transient equations for groundwater flow, heat transport and the transport of reactive solutes in porous media at high temperature in three space dimensions. The code version from 2010 was used in the study [Schmidl et al., 2010b] and is also the one discussed in this thesis. The code uses automatic differentiation [Wolf, 2011] to calculate directional derivatives. The OpenMP parallelization uses nested parallel regions. On the first level, different directional derivatives are computed in parallel. On the second level, each of these derivatives uses OpenMP teams to parallelize the calculation of individual derivatives.

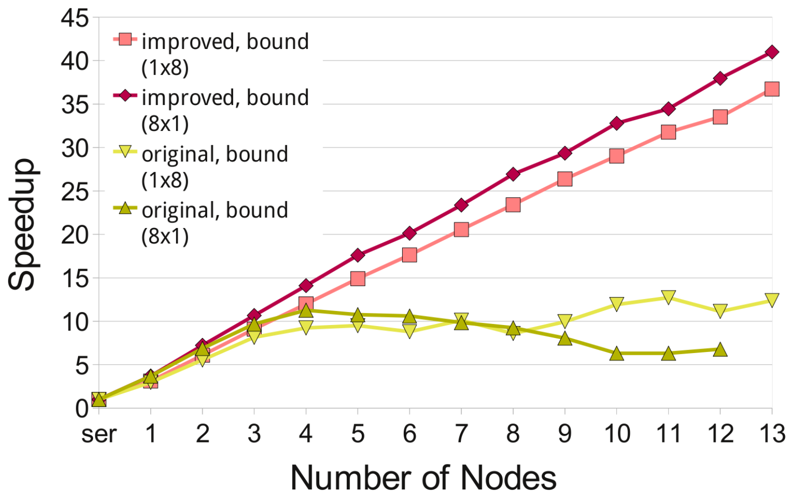


Figure 6.11: Speedup of the SHEMAT-Suite code on a 13 board ScaleMP machine for the original and optimized version.

The SHEMAT-Suite code was optimized for a ScaleMP machine based on 2 socket boards, each equipped with Intel Harpertown Quadcore processors. The code optimization contained the following steps:

- The placement of threads was optimized in a way that one derivative is computed per board of the ScaleMP machine. The inner teams then fill up all cores of a board. Since nearly no data sharing is necessary on the outer level, this reduces synchronization and data sharing between the boards.
- Performance analysis showed that frequent data allocation consumed a lot of runtime. To avoid this performance loss, arrays were allocated at the beginning of the execution which were reused during execution.
- Furthermore, a library to activate huge-pages was used to avoid long data initialization times. Nowadays, the transparent huge-pages feature of current

Linux kernels automatically uses huge-pages which makes this optimization unnecessary, since huge-pages are always used if a large array is allocated. But during the time of this optimization the library increased the overhead of data initialization a lot.

Figure 6.11 shows the speedup of the SHEMAT-Suite code without and with the optimizations applied to the code. For both versions (original-bound and improved-bound) the thread binding was enabled, since for the original code without thread binding no reliable measurements could be done. The speedup is based on a serial run with one thread. Then measurements for different numbers of boards were done, where all cores of a board were used. Running 8 threads on 1 board delivers a speedup of about 3.7. This speedup is limited due to memory limitations on one board. Adding more boards increases the available bandwidth which finally results in a speedup of about 41 on all 13 boards of the system.

Optimizing this code for large NUMA systems, in this case the ScaleMP machine, increased the performance significantly and makes the code well suited for such an architecture.

6.3.2 TrajSearch

TrajSearch is a post-processing code for dissipation element analysis developed at the Institute of Combustion Technology[2] of RWTH Aachen University by Peters and Wang [Peters and Wang, 2006]. The dissipation element analysis provides a deeper understanding of turbulence and can be employed to reconstruct important statistical properties [Gampert et al., 2011].

TrajSearch decomposes a highly-resolved three dimensional turbulent flow field obtained by Direct Numerical Simulation (DNS) into non-arbitrary, space-filling and non-overlapping geometrical elements (dissipation elements). As input data, the algorithm uses the 3D scalar field produced by the DNS. Then, starting from every grid point in this scalar field, the search processes follow the trajectory in ascending and descending gradient direction until a maximum, respectively minimum point is found. For all points in the grid a counter is incremented to store how many trajectories crossed this point. Furthermore, a list of extremal points is stored as well as the mapping of trajectories to these extremal points. All points where the trajectory ended in the same pair of minimum and maximum form a dissipation element.

All search processes can be done independently, as long as the accesses to the result arrays are synchronized. It should be mentioned here, that the compute time for the search process depends on the length of the trajectory which highly varies

[2]http://www.itv.rwth-aachen.de/

during the data set, resulting in a high load imbalance. Furthermore, the data needed for a search process depends on the ascending and descending gradients direction and thus is unknown a-priori. This complicates the data placement on a NUMA machine, since the data access cannot be predicted.

The following optimization steps have been applied to the TrajSearch code:

- **Data placement:** As mentioned before, the data access pattern is unpredictable, so it is impossible to reach 100 % of local memory accesses. But, since the starting point of every search in the scalar field is known, it is possible to achieve local accesses for the start of every trajectory search and the local neighborhood. The scalar field is therefore distributed in equal chunks over the machine in a static way.
- **Local buffering:** As a result of the analysis, the extremal points need to be stored in a global list. Locking the list for every new extremal point produced too much overhead, so every thread keeps a local list in its thread local storage during the computation and at the end these lists are merged. Some of the extremal points might be found by several threads and are thus stored in several private lists, but during the merge process this can be eliminated. Since the memory consumption per thread is low, there is no need to merge double entries earlier. Furthermore, for every grid point the number of crossing trajectories is stored. To avoid extensive locking here, also thread local buffers are implemented, but the memory consumption is much higher. Potentially some GB need to be stored per thread, thus the buffers need to be flushed during computation when a certain threshold is reached into the shared result array.
- **Data allocation:** Since a lot of data allocation calls negatively influenced the performance of the code, an optimized memory allocation call (`kmp_malloc`) was used, as described in chapter 2.
- **NUMA-aware scheduling:** The time needed to calculate a search process for a trajectory depends on different factors, one is the length of the trajectory. So, the processes potentially need extremely different computation times, which results in load imbalance. Since the data locality when such a search process is started should be preserved, dynamic scheduling cannot be applied for the reasons explained in detail in chapter 4. The NUMA-aware scheduler using load-aware stealing is used in the TrajSearch code to avoid these imbalances as much as possible.

Figure 6.12 shows the runtime and speedup achieved with the TrajSearch code on a Numascale machine. The used machine is operated by the University of Oslo's Center for Information Technology, USIT. The systems is a PRACE prototype that it is financially supported by the PRACE-1IP project. As a PRACE prototype

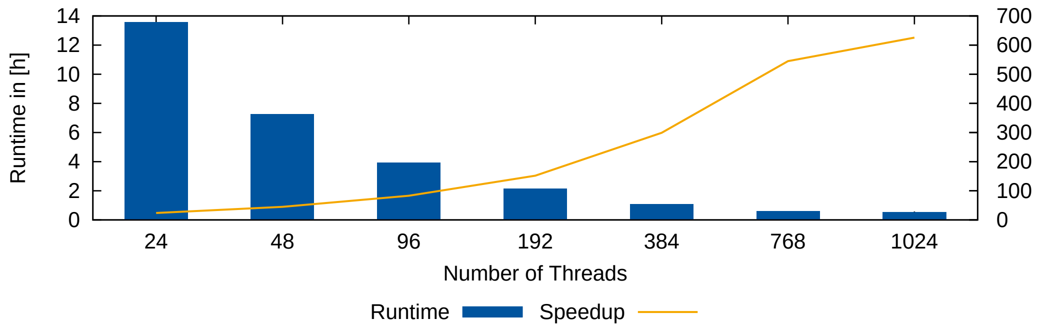


Figure 6.12: Runtime and speedup of the TrajSearch code on a Numascale machine with up to 1024 threads.

system, this machine is representative for the class of large shared memory machines. Since these prototypes are identified as architectures with a promising future by PRACE, evaluating tuning steps on this architecture will have a high relevance on future systems as well.

The configuration of the system is as follows:

- 72 IBM x3755 M3 nodes
- 144 AMD 6174 CPUs
- 1728 Cores
- 4.6 TB Memory
- 3D Torus Interconnect (3x6x4)

The performance results are presented for up to 1024 cores only, for the following reason. The Oracle compiler provided best performance on the system, but it only supports thread placement up to 1024 threads. A serial test run to calculate the speedup was not possible due to the memory requirements of the code. Therefore, the speedup is calculated with the 24 threads run as reference under the assumption that a speedup of 24 is achieved for this reference point. In total, the code scales well over the machine and a speedup of about 625 is achieved with 1024 threads. A performance drop can be observed in the diagram after 768 threads. The reason is that the AMD 6174 CPUs always share floating-point units between two cores. This means the system only has 864 FPUs and so for 1024 threads some FPUs are shared between two threads.

6.4 Summary

In this chapter, a workflow was presented to optimize applications for large NUMA systems. The workflow described for typical performance problems of shared memory programs on large NUMA systems how performance issues can be identified with tools and how they can be avoided. For more difficult issues, it is referred to the appropriate former chapter where several problems have been discussed in depth. Furthermore, the workflow ends with a performance model taking more factors into account than the simple roofline model, without getting too complicated. Finally I presented two case studies, SHEMAT-Suite and TrajSearch, where optimizations have been done to enable application codes to profit from large NUMA systems. Here, the largest tests were done with with TrajSearch on a Numascale system with 1024 threads, where a speedup of about 625 was reached.

7 Summary and Conclusion

In the work presented in this thesis, different relevant aspects of OpenMP programming on large NUMA systems have been covered. First, I investigated different machine characteristics in chapter 2. Here it was shown, that hierarchical NUMA machines, which employ different levels of interconnect technology throughout the system, have a noticeable difference in bandwidth and latency when the different interconnects are used. Also the synchronization time of OpenMP constructs increased by orders of magnitude, when multiple levels of interconnects needed to be used for synchronization. The relevance of this became clear for the SHEMAT-Suite case study, where the binding of threads on different levels of nested parallel regions could use these characteristics to minimize synchronization on the inner most, i.e. second, OpenMP level. Regarding the representation of these NUMA distances to the user, I discussed problems with the System Locality Distance Information Table (SLIT) used in Linux and proposed a measured distance matrix as more accurate representation which could be used in OpenMP runtimes and filled at initialization time with simple tests.

Next, I presented different optimization techniques which can and need to be applied to OpenMP programs to achieve a good performance on large NUMA systems. Basically, the main goal is to reach as many local memory accesses as possible on a NUMA system, this includes the problems of thread placement, memory placement and work distribution. For thread placement, OpenMP already offers binding strategies but leaves the generation of place-lists to the programmer. Here, I proposed a greedy TSP algorithm to approximate the problem, since an optimal place-list cannot be achieved in reasonable time. The algorithm was evaluated on large test systems and it was equal to or nearly equal to the optimal solution with respect to the total distance during neighbors in the list compared to an exponential algorithm which computes the optimal list. The advantage was, that it runs in sub-second timeranges instead of hours or days for the optimal solution, which makes it usable during initialization of the OpenMP runtime. Also the placement of data was targeted in chapter 3 in the OMPX library which in addition to memory migration functionality presented a next_touch implementation for the Linux operating system. Also a model was presented to determine if migration of data in a program is useful or not, depending on the number of times the data is used on the remote node.

The distribution of work is pretty easy in a static case, but on NUMA systems no good scheduling scheme exists for dynamic cases. Therefore, I investigated load balancing on NUMA systems in chapter 4. I looked at standard scheduling techniques and proposed a NUMA-aware scheduling scheme which outperformed all available schedules, i.e. `static`, `dynamic` and `guided` scheduling. Furthermore, I investigated the behavior of different OpenMP runtime systems to handle tasks on NUMA machines and gave advice to programmers, how tasks can be used for work distribution in a good way, under some circumstances, i.e. if the OpenMP runtime uses thread local task queues and task stealing. Additionally, performance tools and their ability to investigate the performance of tasks were examined and compared in chapter 5. All presented tools had weaknesses to investigate the performance of tasks on NUMA systems for the presented NUMA aware task creation. The main reason was, that the tools were hardware unaware, which means they analyze threads, but have no notion where these threads were executed. I presented a tool which merges hardware information into a Score-P trace to allow a better investigation of NUMA related scheduling decisions in the trace.

Finally, I presented a workflow in chapter 6 which puts all the technique together and combines them with standard performance analysis and optimization techniques. The result is a structured procedure which can be applied to do performance engineering of OpenMP codes on large NUMA systems. The presented case studies with the SHEMAT-Suite and the TrajSearch code give evidence, that it is possible to optimize codes for such system sand to achieve a good performance and scaling with OpenMP.

I have presented methods to scale OpenMP programs to more than a thousand threads on large systems, but some of the techniques used are not part of the OpenMP specification, thus I implemented optimization techniques like the NUMA-aware scheduler or the OMPX library. In conclusion, OpenMP is missing features for memory placement and the scheduling in a NUMA-aware fashion is needed. Furthermore, tasks can be programmed NUMA-aware with some runtime systems, but not with all. Here, the standard also needs to improve to offer a vendor-independent way in the spirit of an open specification like OpenMP.

Future hardware is expected to become even more complicated with respect to memory hierarchies. Processors like Intel's Knights Landing have been announced with different types of memory attached to a single processor: Standard DDR4 RAM and on-chip memory called MCDRAM. MCDRAM will provide a higher bandwidth but less capacity than the standard DDR4 memory. Furthermore accelerators will emerge which allow to share memory between a host and a device processor. Future work in the context of OpenMP will have to extend the techniques for memory placement and migration as presented in this dissertation to be applicable on future architectures. An integration into the OpenMP specification is highly desirable,as this is a requirement for portable support of those architectural features.

Bibliography

[Amdahl, 1967] Amdahl, G. M. (1967). Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA. ACM.

[Asanovic et al., 2006] Asanovic, K., Bodik, R., Catanzaro, B. C., Gebis, J. J., Husbands, P., Keutzer, K., Patterson, D. A., Plishker, W. L., Shalf, J., Williams, S. W., and Yelick, K. A. (2006). The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, TECHNICAL REPORT, UC BERKELEY.

[Aslot et al., 2001] Aslot, V., Domeika, M., Eigenmann, R., Gaertner, G., Jones, W., and Parady, B. (2001). SPEComp: A New Benchmark Suite for Measuring Parallel Computer Performance. In Eigenmann, R. and Voss, M., editors, *OpenMP Shared Memory Parallel Programming*, volume 2104 of *Lecture Notes in Computer Science*, pages 1–10. Springer Berlin Heidelberg.

[Ayguadé et al., 1999] Ayguadé, E., Martorell, X., Labarta, J., GonzÃ¡lez, M., and Navarro, N. (1999). Exploiting Multiple Levels of Parallelism in OpenMP: A Case Study. In *Proc. Of the 1999 International Conference on Parallel Processing, Ajzu*, pages 172–180.

[Bailey et al., 1991] Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. (1991). The NAS parallel benchmarks. Technical report, NASA Ames Research Center.

[Barak et al., 1993] Barak, A., Guday, S., and Wheeler, R. G. (1993). *The MOSIX Distributed Operating System: Load Balancing for UNIX*. Springer, Secaucus, NJ, USA.

[Berr et al., 2012] Berr, N., Schmidl, D., Göbbert, J. H., Lankes, S., an Mey, D., Bemmerl, T., and Bischof, C. (2012). Trajectory-Search on ScaleMP's vSMP Architecture. In *Applications, Tools and Techniques on the Road to Exascale Computing : proceedings of the 14th biennial ParCo conference ; ParCo2011 ; held in Ghent, Belgium*, Advances in Parallel Computing ; 22, New York, NY. IOS Press.

[Bircsak et al., 2000] Bircsak, J., Craig, P., Crowell, R., Cvetanovic, Z., Harris, J., Nelson, C. A., and Offner, C. D. (2000). Extending OpenMP for NUMA Machines. In *Proceedings of the 2000 ACM/IEEE Conference on Supercomputing*, SC '00, Washington, DC, USA. IEEE Computer Society.

[Broquedis et al., 2010] Broquedis, F., Clet Ortega, J., Moreaud, S., Furmento, N., Goglin, B., Mercier, G., Thibault, S., and Namyst, R. (2010). hwloc: a Generic Framework for Managing Hardware Affinities in HPC Applications. In IEEE, editor, *PDP 2010 - The 18th Euromicro International Conference on Parallel, Distributed and Network-Based Computing*, Pisa Italie.

[Broquedis et al., 2009] Broquedis, F., Furmento, N., Goglin, B., Namyst, R., and Wacrenier, P.-A. (2009). Dynamic Task and Data Placement over NUMA Architectures: An OpenMP Runtime Perspective. In *Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, IWOMP '09, pages 79–92, Berlin, Heidelberg. Springer-Verlag.

[Bull, 1999] Bull, J. M. (1999). Measuring Synchronisation and Scheduling Overheads in OpenMP. In *In Proc. of 1st European Workshop on OpenMP (EWOMP '99)*, pages 99–105.

[Bull and O'Neill, 2001] Bull, J. M. and O'Neill, D. (2001). A Microbenchmark Suite for OpenMP 2.0. *SIGARCH Comput. Archit. News*, 29(5):41–48.

[Bull et al., 2012] Bull, J. M., Reid, F., and McDonnell, N. (2012). A Microbenchmark Suite for OpenMP Tasks. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 271–274, Berlin, Heidelberg. Springer-Verlag.

[Calotoiu et al., 2013] Calotoiu, A., Hoefler, T., Poke, M., and Wolf, F. (2013). Using Automated Performance Modeling to Find Scalability Bugs in Complex Codes. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 45:1–45:12, New York, NY, USA. ACM.

[Chapman et al., 2007] Chapman, B., Jost, G., and Pas, R. v. d. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming.* The MIT Press.

[Christofides and GROUP., 1976] Christofides, N. and GROUP., C.-M. U. P. P. M. S. R. (1976). *Worst-Case Analysis of a New Heuristic for the Travelling Salesman Problem.* Management sciences research report. Defense Technical Information Center.

[Clauser, 2003] Clauser, C., editor (2003). *Numerical Simulation of Reactive Flow in Hot Aquifers. SHEMAT and Processing SHEMAT.* Springer, New York, NY, USA.

[Cramer et al., 2012] Cramer, T., Schmidl, D., Klemm, M., and an Mey, D. (2012). OpenMP Programming on Intel Xeon Phi Coprocessors: An Early Performance Comparison. In *Proceedings of the Many-core Applications Research Community (MARC) Symposium at RWTH Aachen University*, pages 38–44.

[Culler et al., 1993] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R., and von Eicken, T. (1993). LogP: Towards a Realistic Model of Parallel Computation. In *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '93, pages 1–12, New York, NY, USA. ACM.

[Donfack et al., 2012] Donfack, S., Grigori, L., Gropp, W. D., and Kale, V. (2012). Hybrid Static/Dynamic Scheduling for Already Optimized Dense Matrix Factorization. In *Proceedings of the 2012 IEEE 26th International Parallel and Distributed Processing Symposium*, IPDPS '12, pages 496–507, Washington, DC, USA. IEEE Computer Society.

[Duran et al., 2008] Duran, A., Corbalán, J., and Ayguadé, E. (2008). Evaluation of OpenMP Task Scheduling Strategies. In *Proceedings of the 4th International Conference on OpenMP in a New Era of Parallelism*, IWOMP'08, pages 100–110, Berlin, Heidelberg. Springer-Verlag.

[Duran et al., 2009] Duran, A., Teruel, X., Ferrer, R., Martorell, X., and Ayguade, E. (2009). Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In *Parallel Processing, 2009. ICPP '09. International Conference on*, pages 124–131.

[Eichenberger et al., 2012] Eichenberger, A., Terboven, C., Wong, M., and an Mey, D. (2012). The Design of OpenMP Thread Affinity. In Chapman, B., Massaioli, F., Müller, M., and Rorro, M., editors, *OpenMP in a Heterogeneous World*, volume 7312 of *Lecture Notes in Computer Science*, pages 15–28. Springer Berlin Heidelberg.

[Gampert et al., 2011] Gampert, M., Göbbert, J. H., Gauding, M., Schäfer, P., and Peters, N. (2011). Extensive strain along gradient trajectories in the turbulent kinetic energy field. *New Journal of Physics*.

[Geimer et al., 2008] Geimer, M., Wolf, F., Wylie, B. J. N., Ábrahám, E., Becker, D., and Mohr, B. (2008). The SCALASCA Performance Toolset Architecture. In *Proc. of the International Workshop on Scalable Tools for High-End Computing (STHEC), Kos, Greece*, pages 51–65.

[Goglin and Furmento, 2009] Goglin, B. and Furmento, N. (2009). Enabling high-performance memory migration for multithreaded applications on LINUX. In *Parallel Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–9.

[Gropp et al., 2014] Gropp, W., Lusk, E., and Skjellum, A. (2014). *Using MPI: Portable Parallel Programming with the Message-Passing Interface,* 3rd edition. MIT Press.

[Hager, 2013] Hager, G. (2013). Performance engineering: from numbers to insight. In *Euro-Par 2012: Parallel Processing Workshops*, pages 393–394. Springer.

[Hager and Wellein, 2010] Hager, G. and Wellein, G. (2010). *Introduction to High Performance Computing for Scientists and Engineers.* CRC Press, Inc., Boca Raton, FL, USA, 1st edition.

[Hewlett-Packard et al., 2011] Hewlett-Packard, Intel, Microsoft, Phoenix, and Toshiba (2011). Advanced Configuration and Power Interface. http://www.acpi.info/.

[Hoeflinger, 2006] Hoeflinger, J. P. (2006). Extending OpenMP to Clusters.

[Huang et al., 2006] Huang, L., Chapman, B., and Liao, C. (2006). An Implementation and Evaluation of Thread Subteam for OpenMP Extensions. Workshop on Programming Models for Ubiquitous Parallelism (PMUP 06), Seattle.

[Intel, 2013] Intel (2013). Intel VTune Amplifier XE. Last accessed on March 24, 2016.

[Jonathan Corbet, 2014] Jonathan Corbet (2014). AutoNUMA: the other approach to NUMA scheduling. `http://lwn.net/Articles/488709/`. Last visited on 09/05/2014.

[Jost et al., 2003] Jost, G., Jin, H., Mey, D. A., and Hatay, F. F. (2003). Comparing the OpenMP, MPI, and Hybrid Programming Paradigms on an SMP Cluster 1.

[Kale et al., 2014] Kale, V., Randles, A., and Gropp, W. D. (2014). Locality-Optimized Mixed Static/Dynamic Scheduling for Improving Load Balancing on SMPs. In *Proceedings of the 21st European MPI Users' Group Meeting*, EuroMPI/ASIA '14, pages 115:115–115:116, New York, NY, USA. ACM.

[Knüpfer et al., 2011] Knüpfer, A., Rössel, C., an Mey, D., Biersdorff, S., Diethelm, K., Eschweiler, D., Geimer, M., Gerndt, M., Lorenz, D., Malony, A. D., Nagel, W. E., Oleynik, Y., Philippen, P., Saviankou, P., Schmidl, D., Shende, S. S., Tschüter, R., Wagner, M., Wesarg, B., and Wolf, F. (2011). Score-P – A Joint Performance Measurement Run-Time Infrastructure for Periscope, Scalasca, TAU, and Vampir. In *Proc. of 5th Parallel Tools Workshop, Dresden, Germany.*

[LaGrone et al., 2011] LaGrone, J., Aribuki, A., Addison, C., and Chapman, B. (2011). A Runtime Implementation of OpenMP Tasks. In Chapman, B., Gropp, W., Kumaran, K., and Müller, M., editors, *OpenMP in the Petascale Era*, volume 6665 of *Lecture Notes in Computer Science*, pages 165–178. Springer Berlin Heidelberg.

[Lankes et al., 2010] Lankes, S., Bierbaum, B., and Bemmerl, T. (2010). Affinity-on-next-touch: An Extension to the Linux Kernel for NUMA Architectures. In *Proceedings of the 8th International Conference on Parallel Processing and Applied Mathematics: Part I*, PPAM'09, pages 576–585, Berlin, Heidelberg. Springer-Verlag.

[Laudon and Lenoski, 1997] Laudon, J. and Lenoski, D. (1997). The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, pages 241–251, New York, NY, USA. ACM.

[Löf and Holmgren, 2005] Löf, H. and Holmgren, S. (2005). Affinity-on-next-touch: Increasing the Performance of an Industrial PDE Solver on a cc-NUMA System. In *Proceedings of the 19th Annual International Conference on Supercomputing*, ICS '05, pages 387–392, New York, NY, USA. ACM.

[Lorenz et al., 2010] Lorenz, D., Mohr, B., Rössel, C., Schmidl, D., and Wolf, F. (2010). How to reconcile event-based performance analysis with tasking in OpenMP. In *Proceedings of the 6th international conference on Beyond Loop Level Parallelism in OpenMP: accelerators, Tasking and more*, IWOMP'10, pages 109–121, Berlin, Heidelberg. Springer-Verlag.

[Lu et al., 1998] Lu, H., Hu, Y. C., and Zwaenepoel, W. (1998). OpenMP on networks of workstations. In *Supercomputing '98: Proceedings of the 1998 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–15, Washington, DC, USA. IEEE Computer Society.

[Margery et al., 2003] Margery, D., Vallee, G., Lottiaux, R., Morin, C., and yves Berthou, J. (2003). Kerrighed: A SSI Cluster OS Running OpenMP. In *In Proc. 5th European Workshop on OpenMP (EWOMP03*.

[Mattson et al., 2004] Mattson, T., Sanders, B., and Massingill, B. (2004). *Patterns for Parallel Programming.* Addison-Wesley Professional, first edition.

[McCalpin, 1995] McCalpin, J. D. (1995). STREAM: Sustainable Memory Bandwidth in High Performance Computers. Last accessed on March 24, 2016.

[McVoy and Staelin, 1996] McVoy, L. and Staelin, C. (1996). lmbench: Portable Tools for Performance Analysis. In *Proceedings of the 1996 annual conference on USENIX Annual Technical Conference*, ATEC '96, pages 23–23, Berkeley, CA, USA. USENIX Association.

[Mey et al., 2012] Mey, D. a., Biersdorf, S., Bischof, C., Diethelm, K., Eschweiler, D., Gerndt, M., Knüpfer, A., Lorenz, D., Malony, A., Nagel, W. E., Oleynik, Y., Rössel, C., Saviankou, P., Schmidl, D., Shende, S., Wagner, M., Wesarg, B., and Wolf, F. (2012). Score-P: A Unified Performance Measurement System for Petascale Applications. In Bischof, C., Hegering, H.-G., Nagel, W. E., and Wittum, G., editors, *Competence in High Performance Computing 2010*, pages 85–97. Springer Berlin Heidelberg. 10.1007/978-3-642-24025-6_8.

[Mohr et al., 2002] Mohr, B., Malony, A. D., Shende, S., and Wolf, F. (2002). Design and Prototype of a Performance Tool Interface for OpenMP. *J. Supercomput.*, 23(1):105–128.

[Morin et al., 2004] Morin, C., Gallard, P., Lottiaux, R., and Vallée, G. (2004). Towards an efficient single system image cluster operating system. *Future Gener. Comput. Syst.*, 20(4):505–521.

[Müller et al., 2012] Müller, M. S., Baron, J., Brantley, W. C., Feng, H., Hackenberg, D., Henschel, R., Jost, G., Molka, D., Parrott, C., Robichaux, J., Shelepugin, P., van Waveren, M., Whitney, B., and Kumaran, K. (2012). SPEC OMP2012 – an Application Benchmark Suite for Parallel Systems Using OpenMP. In *Proceedings of the 8th International Conference on OpenMP in a Heterogeneous World*, IWOMP'12, pages 223–236, Berlin, Heidelberg. Springer-Verlag.

[Nagel et al., 1996] Nagel, W., Weber, M., Hoppe, H.-C., and Solchenbach, K. (1996). VAMPIR: Visualization and Analysis of MPI Resources. *Supercomputer*, 12(1):69–80.

[Nikolopoulos et al., 2000] Nikolopoulos, D., Papatheodorou, T., Polychronopoulos, C., Labarta, J., and Ayguade, E. (2000). Leveraging Transparent Data Distribution in OpenMP via User-Level Dynamic Page Migration. In Valero, M., Joe, K., Kitsuregawa, M., and Tanaka, H., editors, *High Performance Computing*, volume 1940 of *Lecture Notes in Computer Science*, pages 415–427. Springer Berlin Heidelberg.

[Noordergraaf and van der Pas, 1999] Noordergraaf, L. and van der Pas, R. (1999). Performance Experiences on Sun's Wildfire Prototype. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing*, SC '99, New York, NY, USA. ACM.

[OpenMP ARB, 2013] OpenMP ARB (2013). OpenMP Application Program Interface, v. 4.0.

[Oracle, 2013] Oracle (2013). Oracle Solaris Studio 12.2: Performance Analyzer. Last accessed on March 24, 2016.

[Pacheco, 1996] Pacheco, P. S. (1996). *Parallel Programming with MPI*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[Peters and Wang, 2006] Peters, N. and Wang, L. (2006). Dissipation element analysis of scalar fields in turbulence. *C. R. Mechanique*, 334:493–506.

[Ramos and Hoefler, 2013] Ramos, S. and Hoefler, T. (2013). Modeling Communication in Cache-coherent SMP Systems: A Case-study with Xeon Phi. In *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, HPDC '13, pages 97–108, New York, NY, USA. ACM.

[Sato et al., 2001] Sato, M., Harada, H., Hasegawa, A., and Ishikawa, Y. (2001). Cluster-enabled OpenMP: An OpenMP compiler for the SCASH software distributed shared memory system. *Scientific Programming*, 9(2,3):123–130.

[Schmidl, 2009] Schmidl, D. (2009). Platzierung von OpenMP-Programmen auf hierarchischen Speicherarchitekturen. (Diplomarbeit, RWTH Aachen University).

[Schmidl et al., 2013a] Schmidl, D., an Mey, D., and Müller, M. (2013a). Performance Characteristics of Large SMP Machines. In Rendell, A., Chapman, B., and Müller, M., editors, *OpenMP in the Era of Low Power Devices and Accelerators*, volume 8122 of *Lecture Notes in Computer Science*, pages 58–70. Springer Berlin Heidelberg.

[Schmidl et al., 2014] Schmidl, D., Cramer, T., Terboven, C., Mey, D., and Müller, M. (2014). An OpenMP Extension Library for Memory Affinity. In DeRose, L., de Supinski, B., Olivier, S., Chapman, B., and Müller, M., editors, *Using and Improving OpenMP for Devices, Tasks, and More*, volume 8766 of *Lecture Notes in Computer Science*, pages 103–114. Springer International Publishing.

[Schmidl et al., 2013b] Schmidl, D., Cramer, T., Wienke, S., Terboven, C., and Müller, M. (2013b). Assessing the Performance of OpenMP Programs on the Intel Xeon Phi. In Wolf, F., Mohr, B., and an Mey, D., editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 547–558. Springer Berlin Heidelberg.

[Schmidl et al., 2012] Schmidl, D., Philippen, P., Lorenz, D., Rössel, C., Geimer, M., an Mey, D., Mohr, B., and Wolf, F. (2012). Performance Analysis Techniques for Task-Based OpenMP Applications. In Chapman, B., Massaioli, F., Müller, M., and Rorro, M., editors, *OpenMP in a Heterogeneous World*, volume 7312 of *Lecture Notes in Computer Science*, pages 196–209. Springer Berlin / Heidelberg. 10.1007/978-3-642-30961-8_15.

[Schmidl et al., 2010a] Schmidl, D., Terboven, C., an Mey, D., and Bücker, M. (2010a). Binding Nested OpenMP Programs on Hierarchical Memory Architectures. In Sato, M., Hanawa, T., Müller, M., Chapman, B., and de Supinski, B., editors, *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking* *and More*, volume 6132 of *Lecture Notes in Computer Science*, pages 29–42. Springer Berlin / Heidelberg. 10.1007/978-3-642-13217-9_3.

[Schmidl et al., 2013c] Schmidl, D., Terboven, C., an Mey, D., and Müller, M. S. (2013c). Suitability of Performance Tools for OpenMP Task-Parallel Programs. In Knüpfer, A., Gracia, J., Nagel, W. E., and Resch, M. M., editors, *Tools for High Performance Computing 2013*, pages 25–37. Springer International Publishing.

[Schmidl et al., 2013d] Schmidl, D., Terboven, C., Iwainsky, C., Bischof, C., and Müller, M. S. (2013d). Towards a Performance Engineering Workflow for OpenMP 4.0. In *Proc. of ParCo, Minisymposium Modeling and Engineering*, Munich, Germany.

[Schmidl et al., 2010b] Schmidl, D., Terboven, C., Wolf, A., Mey, D. a., and Bischof, C. (2010b). How to Scale Nested OpenMP Applications on the ScaleMP vSMP Architecture. In *Proceedings of the 2010 IEEE International Conference on Cluster Computing*, CLUSTER '10, pages 29–37, Washington, DC, USA. IEEE Computer Society.

[Schmidl and Vesterkjær, 2014] Schmidl, D. and Vesterkjær, A. (2014). Scaling OpenMP Programs to Thousand Cores on the Numascale Architecture. Poster Session at SC14, `http://sc14.supercomputing.org/sites/all/themes/sc14/files/archive/tech_poster/tech_poster_pages/post108.html`.

[Schmidl et al., 2015] Schmidl, D., Vesterkjær, A., and Müller, M. S. (2015). Evaluating OpenMP Performance on Thousands of Cores on the Numascale Architecture. In *Proceedings of ParCo2015*.

[Terboven et al., 2008a] Terboven, C., an Mey, D., Schmidl, D., Jin, H., and Reichstein, T. (2008a). Data and thread affinity in openmp programs. In *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?*, MAW '08, pages 377–384, New York, NY, USA. ACM.

[Terboven et al., 2008b] Terboven, C., Mey, D., Schmidl, D., and Wagner, M. (2008b). First Experiences with Intel Cluster OpenMP. In Eigenmann, R. and de Supinski, B., editors, *OpenMP in a New Era of Parallelism*, volume 5004 of *Lecture Notes in Computer Science*, pages 48–59. Springer Berlin / Heidelberg. 10.1007/978-3-540-79561-2_5.

[Terboven et al., 2012a] Terboven, C., Schmidl, D., Cramer, T., and an Mey, D. (2012a). Assessing OpenMP Tasking Implementations on NUMA Architectures. In Chapman, B., Massaioli, F., Müller, M., and Rorro, M., editors, *OpenMP in a Heterogeneous World*, volume 7312 of *Lecture Notes in Computer Science*, pages 182–195. Springer Berlin / Heidelberg. 10.1007/978-3-642-30961-8_14.

[Terboven et al., 2012b] Terboven, C., Schmidl, D., Cramer, T., and an Mey, D. (2012b). Task-Parallel Programming on NUMA Architectures. In Kaklamanis, C., Papatheodorou, T., and Spirakis, P., editors, *Euro-Par 2012 Parallel Processing*, volume 7484 of *Lecture Notes in Computer Science*, pages 638–649. Springer Berlin / Heidelberg. 10.1007/978-3-642-32820-6_63.

[Treibig and Hager, 2010] Treibig, J. and Hager, G. (2010). Introducing a Performance Model for Bandwidth-Limited Loop Kernels. In Wyrzykowski, R., Dongarra, J., Karczewski, K., and Wasniewski, J., editors, *Parallel Processing and Applied Mathematics*, volume 6067 of *Lecture Notes in Computer Science*, pages 615–624. Springer Berlin Heidelberg.

[Vallée et al., 2003] Vallée, G., Lottiaux, R., Rilling, L., Berthou, J.-Y., Malhen, I. D., and Morin, C. (2003). A Case for Single System Image Cluster Operating Systems: The Kerrighed Approach. *Parallel Processing Letters*, 13(2):95–122.

[Wang et al., 2015] Wang, B., Schmidl, D., and Müller, M. (2015). Evaluating the Energy Consumption of OpenMP Applications on Haswell Processors. In Terboven, C., de Supinski, B. R., Reble, P., Chapman, B. M., and Müller, M. S., editors, *OpenMP: Heterogenous Execution and Data Movements*, volume 9342 of *Lecture Notes in Computer Science*, pages 233–246. Springer International Publishing.

[Weyers et al., 2014] Weyers, B., Terboven, C., Schmidl, D., Herber, J., Kuhlen, T. W., Müller, M. S., and Hentschel, B. (2014). Visualization of Memory Access Behavior on Hierarchical NUMA Architectures. In *Proceedings of the First Workshop on Visual Performance Analysis*, VPA '14, pages 42–49, Piscataway, NJ, USA. IEEE Press.

[Williams et al., 2009] Williams, S., Waterman, A., and Patterson, D. (2009). Roofline: An Insightful Visual Performance Model for Multicore Architectures. *Commun. ACM*, 52(4):65–76.

[Wolf, 2011] Wolf, A. (2011). *Ein Softwarekonzept zur hierarchischen Parallelisierung von stochastischen und deterministischen Inversionsproblemen auf modernen ccNUMA-Plattformen unter Nutzung automatischer Programmtransformation.* PhD thesis, Aachen. Zusammenfassung in dt. und engl. Sprache; Aachen, Techn. Hochsch., Diss., 2011.

[Zhang, 2008] Zhang, G. (2008). Extending the OpenMP Standard for Thread Mapping and Grouping . In *OpenMP Shared Memory Parallel Programming*, volume 4315/2008, pages 435–446.