Job-Scheduling in Main-Memory Based Parallel Database Systems

Diploma Thesis

Jochen Seidel

May 8, 2011

Supervisors:
Dipl.-Inform. Jonathan Dees
Dipl.-Inform. Dipl.-Math. Jochen Speck

Responsible Supervisor:
Prof. Dr. rer. nat. Peter Sanders

Abstract

In systems with Non-Uniform Memory Access (NUMA), memory bandwidth depends upon the memory location relative to the accessing processor. Each processing node can have its own local memory, but accesses foreign memory transparently in the same address space. Our approach is to minimize memory access time by scheduling jobs on the processing node where most memory access happens. We have implemented a working user-land scheduler favoring local memory access with a user interface similar to the one of Intel Thread Building Blocks (TBB). Furthermore, we devise a model to predict the effects of concurrently running jobs. We then show theoretically and in experiments, how efficiency can be gained by executing two different jobs in parallel rather than sequentially. This implies the concurrent execution of more than one query in a DBMS can be advantageous.

Declaration

I hereby confirm that the thesis at hand is my own work, and that all literal quotations and other author's ideas have been acknowledged in notes to the text or the bibliography.

Erklärung

Ich erkläre hiermit, die vorliegende Arbeit selbständig verfasst zu haben und keine außer den angegebenen Quellen verwendet zu haben.

Karlsruhe, May 8, 2011

Jochen Seidel

Deutsche Zusammenfassung German Abstract

Systeme mit nichtuniformem Speicherzugriff (Non-Uniform Memory Access, NUMA) bestehen aus miteinander verbundenen Rechenknoten, an die jeweils lokaler Arbeitsspeicher angebunden ist. Sämtliche lokalen Speicher erscheinen in solchen Systemen transparent als ein zusammenhängender physikalischer Speicherbereich. Die einem Job zur Verfügung stehende Speicherbandbreite hängt daher von der *Entfernung* zu der angefragten Speicherstelle ab.

In dieser Arbeit wird ein Ansatz verfolgt, Speicherzugriffe für parallele Jobs, deren Daten über mehrere Knoten verteilt liegen, zu beschleunigen. Dies soll dadurch geschehen, dass die Ausführung des Jobs auf demjenigen Rechenknoten stattfindet, an dem die Daten lokal im Speicher liegen.

Zunächst wird analysiert, wie sich parallel ausgeführte Programme verhalten, die den gleichen Speicherkanal belegen. Dabei stellt sich heraus, dass lokaler Speicherzugriff eine etwa doppelt so große Bandbreite erreicht wie Speicherzugriff auf fremde Knoten. Aufgrund dieser Feststellung wird das C++ Scheduling-Framework *NUMA Building Blocks* (**NBB**) entwickelt. In diesem können Jobs festlegen, auf welchem Knoten die benötigten Daten vorhanden sind. Die Scheduling-Entscheidungen werden dabei innerhalb der Anwendungsschicht getroffen, da nur dort Wissen über Aufteilung und Verwendung des genutzten Hauptspeichers auf verschiedene Knoten vorhanden ist.
Das Framework ist so gestaltet, dass der Scheduler austauschbar ist. Dies soll zukünftig ermöglichen, auch andere Scheduling-Algorithmen und -Konzepte experimentell zu testen. Die Programmierschnittstelle der Bibliothek orientiert sich dabei in groben Zügen an derjenigen der TBB-Bibliothek von Intel.

Eine bereits existierende Implementierung des TPC-H Datenbank-Benchmarks setzt TBB zur Parallelisierung ein. Durch die beschriebenen Vorarbeiten ist es mit geringem Aufwand möglich, diese auch mit NBB zu übersetzen. Anhand des Benchmarks wird die Praxistauglichkeit des entwickelten Frameworks demonstriert. Bei der **Evaluation** können in Extremfällen *Laufzeitgewinne bis zu 67%* für diese realitätsnahe Anwendung beobachtet werden. Die Anwendung liefert außerdem weitere Anhaltspunkte über das Verhalten von parallel ausgeführten bandbreitenintensiven Jobs.

Um diese Effekte zum Treffen von Scheduling-Entscheidungen vorhersagen zu können, wird schließlich ein **Modell** entwickelt. In dem Modell werden sowohl *Beschleunigungseffekte* durch Zuteilung von mehr als einen Prozessor zu einem Job, als auch *Verlangsamung* durch beschränkte Speicherbandbreite abgebildet. Dies führt zur Charakterisierung eines Jobs durch zwei Kennzahlen: Laufzeit auf einem Prozessor mit unbeschränkter Speicherbandbreite und demjenigen Anteil der Laufzeit, in dem auf Speicherzugriffe gewartet wird. Trotz der Tatsache, dass viele Rechnereigenschaften wie z.B. Caches bei der Modellierung außer Acht gelassen wurden, liefert das Modell gute Vorhersagen über die Programmlaufzeit.
Außerdem sind mit Hilfe des Modells Aussagen darüber möglich, ob die parallele Ausführung von zwei Jobs den Rechner effizienter auslasten kann als die sequenzielle Ausführung. Dies kann dann der Fall sein, wenn einer der Jobs während seiner Laufzeit nur wenig Speicherbandbreite benötigt, ein anderer jedoch sehr viel. In diesem Fall profitiert der speicherintensive Job aufgrund der beschränkten Bandbreite kaum von zusätzlichen Prozessoren. Daher ist es sinnvoller, diese zur Ausführung des rechenintensiven Jobs zu nuzten, der den anderen wegen seines

geringen Bandbreitenbedarfs kaum verlangsamt.

Experimente ergaben jedoch, dass der auftretende Parallelisierungsoverhead einen weitaus größeren Einfluss auf die Laufzeit hat. Demgegenüber ist die mit Hilfe des Modells vorhergesagte Effizienzsteigerung vernachlässigbar. Wie diese Effekte besser abgebildet und vorhergesagt werden können bleibt daher zu beantworten.

Contents

1. Introduction

This diploma thesis addresses the problem arising from the discrepancy between processing speed and memory bandwidth of current processors. We will analyze how job execution time is influenced by available memory bandwidth. Eventually we devise a new model for job scheduling in order to predict these effects. In order to accomplish that we first develop a new user-land scheduling library. We port an existing benchmark program to the new scheduling library and observe how scheduling decisions affect job run-time.

Our approach differs from others in the fact that the two modeled resources *CPU time* and *memory bandwidth* are *interacting* with each other. Before presenting our results we will now briefly introduce some terminology and discuss how our work relates to job scheduling and database systems.

1.1. NUMA Architecture

The term Non-Uniform Memory Access (NUMA) is used to describe multi-processing environments where not all memory can be accessed at the same speed, although it is available in one continuous (physical) address space. This stems from the fact that in shared-memory systems with more than one processor socket, each socket has *local memory* attached to it. In order to access memory attached to another processor socket, the processors are connected by a communication network. Accessing non-local memory is called *foreign memory access*. Not only does it exhibit higher access latency because it is not attached to the local memory controller, but also the bandwidth is smaller as it is for local memory access.

NUMA systems were designed to overcome the memory bottleneck that arises in Simultaneous Multiprocessing (SMP) systems, where multiple processor cores share a common memory bus. In fact, current NUMA systems are a combination of the two architectures: Multiple processor cores are attached to one socket with a shared memory bus, and multiple interconnected sockets form a NUMA computer. Some current processors also support Simultaneous Multithreading (SMT), which is called "Hyperthreading" by Intel. Here, one CPU core presents itself as multiple logical CPUs to the operating system. This allows the CPU to execute more than one thread on one core, which can be beneficial if for example a thread is waiting for data to be fetched from memory. In such cases, the processor core can execute arithmetic instructions from another thread running on the logical CPU belonging to that core.

Figure 1.1 on the following page gives a schematic view of a typical NUMA system. A *NUMA node* consists of a processor with attached local memory. Each processor can be composed of multiple SMP cores which in turn may support SMT. It is not necessary that the network connecting the nodes forms a complete graph.

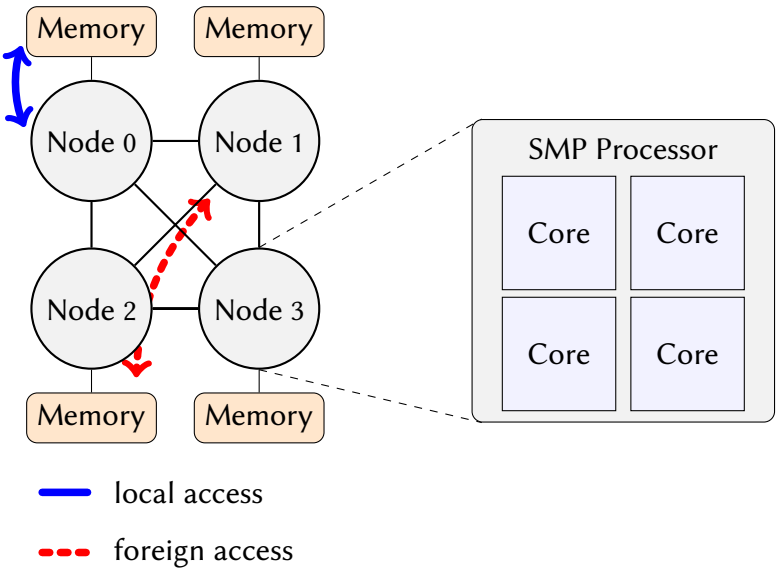


Figure 1.1.: Schematic view of a NUMA system consisting of SMP processors.

1.2. Job Scheduling

In the classic job scheduling problem k jobs or tasks with run-times t_i have to be scheduled on p processors. In the standard case the goal is to minimize overall time needed to execute all jobs, which is called *makespan*.

Many variants of this problem have been studied. For example scheduling of jobs that require a fixed amount of processors at the same time (*multiprocessor jobs*), jobs that have varying run-times depending on a fixed number of processors available to that job (*moldable jobs*) or jobs that can vary the number of used processors during run-time (*malleable jobs*). Jobs may also be *preemptible*, probably at a certain cost, have a *dependency relation*, an *arrival time* before which the job cannot be scheduled, or the requirement to meet a specific *deadline*.

Different optimization goals apart from *makespan* include *average response time* (time between job arrival and completion), which is also called flow time, and *throughput* (jobs processed per time). Variants with additional independent resource constraints were investigated as well. The problems have been studied both as on-line and off-line variants.

This thesis will concentrate on *moldable* and partly on *malleable jobs*. We analyze the effects of an implicitly shared resource, namely memory-bandwidth, on job run-time. As for the optimization criterion we will discuss the consequences of our findings on *makespan* and *throughput*. Scheduling jobs as they appear in database systems is an own field of research, which often focuses on maximizing *throughput* for longer running jobs and/or minimizing *average response time* for interactive jobs.

1.3. Database Systems

Traditional relational database systems store structured data that can be accessed and modified using a standardized language for relational database systems, usually referred to as "Structured Query Language" (SQL). All data is stored on disk and required parts are loaded into main memory on demand. A lot of research was conducted concerning inter- and intra-query parallelism. To execute a query in parallel, so-called *query execution plans* are arranged from basic database operations and optimized for anticipated execution time. These optimizations may also consider disk access scheduling to improve this main bottleneck of traditional database systems.

However, continuous declines in RAM pricing have led to a new concept for databases: Instead of keeping data on disk, all data can be kept in memory. Furthermore, the traditional data layout has changed from row-wise storage to columnar data storage. The reason for that lies in the fact that most database queries refer to only few columns of a table. Because not only data from disk, but also main memory is transferred block-wise only (one sector or cache line respectively), many unneeded data transfers occur when data is stored by row. MonetDB[1] is a popular example of a main-memory based columnar database.

A standardized database benchmark suite was designed by the Transaction Processing Council (TPC), a "non-profit corporation founded to define transaction processing and database benchmarks"[2]. Its focus is to model data and queries as they appear in real-world business applications. In this thesis, we will use the TPC Benchmark Specification H (TPC-H) as a collection of database jobs for scheduling. We use an experimental in-memory implementation by Jonathan Dees[3] to measure the effects of bandwidth-sharing on NUMA hardware.

In the following section we present the macro-structure of this thesis.

1.4. Outline

The remainder of this thesis is structured as follows: After reviewing related work in section 1.5, we will describe the different hardware setups that were used in our experiments in chapter 2 at first. After describing the basic setup, we present memory bandwidth and SPECint_rate2006[4] benchmark results in section 2.1 in order to quantitatively characterize the setup. Analysis of these results lets us expect NUMA-aware scheduling to be beneficial to certain workloads.

In chapter 3, we then present our newly developed library for NUMA-aware user-land scheduling. After giving a broad overview in section 3.1 and simple examples of usage, the details of our implementation are described in section 3.3.

We then use this scheduling framework to schedule the execution of TPC-H database benchmark queries in chapter 4. Analysis of execution times reveal our approach in some cases is as much as 67% faster compared to NUMA-agnostic scheduling.

[1] http://monetdb.cwi.nl/
[2] http://www.tpc.org/information/about/about.asp
[3] This work is not published yet.
[4] http://www.spec.org/

Collecting further measurements of the memory-bandwidth dependent run-time of those queries, we devise a new model for parallel job execution in chapter 5. It allows not only to model moldable and malleable jobs, but also takes effects of memory bandwidth sharing into account. In section 5.4 we then show, why the model suggests that certain pairs of jobs can be executed with higher efficiency when running them in parallel rather than sequentially. We then try to verify the predicted effect in experiments and discuss arising problems.

Finally, we summarize our results in chapter 6 and point out possible future work on these topics.

1.5. Related Work

Job scheduling is a broad field in computer science. The Handbook of Scheduling edited by Leung et al. [13] gives an overview on the topic. A good overview of many theoretical results concerning *multiprocessor task scheduling* can be found in [5]. The authors of [18] describe an polynomial 2-approximation for non-preemptive moldable job scheduling. In [2] a classification for resource constrained scheduling problems is introduced and relations between classes identified.

The gap between theory and practice in job scheduling is the topic of [7] by Feitelson et al. Frachtenberg and Feitelson also point out common "Pitfalls in parallel job scheduling evaluation" in [8]. In [4] Cirne and Berman show how using job moldability can improve turn-around time of jobs running on a supercomputer system.

Many experimental studies focus on locality of scheduling to minimize cache misses or communication overhead. Arora [1] and Meng et al. [15] are examples for this in multi-programmed (SMP and SMT) systems. Consequently, Brecht [3] and Koita et al. [11] concentrate on NUMA-scheduling in terms of system-wide scheduling and focus on scheduling related jobs on nearby cores. Philbin et al. [16] showed that splitting problems into fine-grained threads can even improve performance in sequential programs due to fewer cache misses.
The Sequoia Programming Language [6] takes a different approach where the programmer has to deal with the memory hierarchy (CPU cache levels as well as per-node memory) explicitly.

Garofalakis investigated scheduling of parallel query execution plans with respect to time- and space-shared resources (CPUs or disks and memory, respectively) in [9]. Manegold's Ph.D. Thesis [14] introduces a sophisticated modeling technique to develop cache-conscious database algorithms. Lee et al. show how to reduce cache misses due to concurrently running queries that compete for a shared cache by a technique called *page colouring* in [12].

2. Hardware Setup

The experiments were conducted on two different hardware setups. One was composed of four Intel Xeon processors, the other of four AMD Opteron processors. Basically, these processor chips (NUMA nodes) consist of a number of processor cores supporting SMP. The number of links each node can have to other nodes is limited, thus the interconnections need not necessarily form a fully connected graph (though this is the case for the Xeon system). Note that due to virtual memory management, an application cannot tell whether it is accessing local or foreign memory without help from the operating system.

The Xeon system additionally has support for SMT, visible to the user as two logical CPUs with shared L1- and L2-Cache. The third level cache is shared among all cores on the chip, as depicted in figure 2.1.

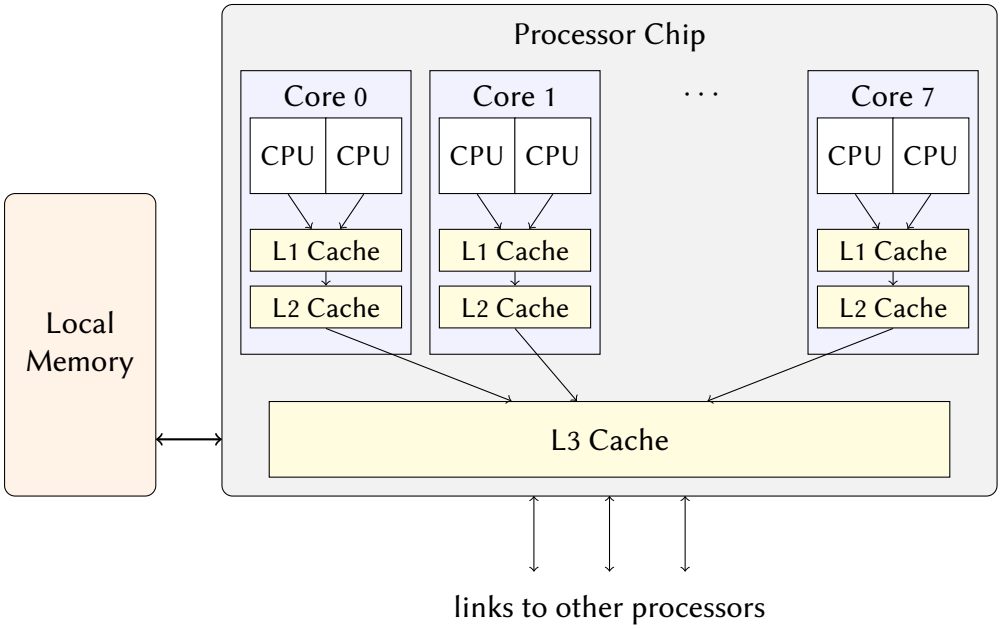


Figure 2.1.: Cache hierarchy of Xeon processors, using the X7560 model as example.

The Xeon system has four pairwise connected Intel Xeon X7560 processors running at 2.27 GHz. One X7560 processor is divided into eight cores with two SMT threads each, totalling in 16 logical CPUs per socket and 64 in the whole system. The eight cores in one socket share 24 MB of level 3 cache. Each socket has 64 GB of main memory attached, summing up to a total of 256 GB main memory available.

The Opteron system differs from the Xeon system in several aspects. First, the Opteron 6168 processor which we used for our experiments does not support SMT, so only one thread can

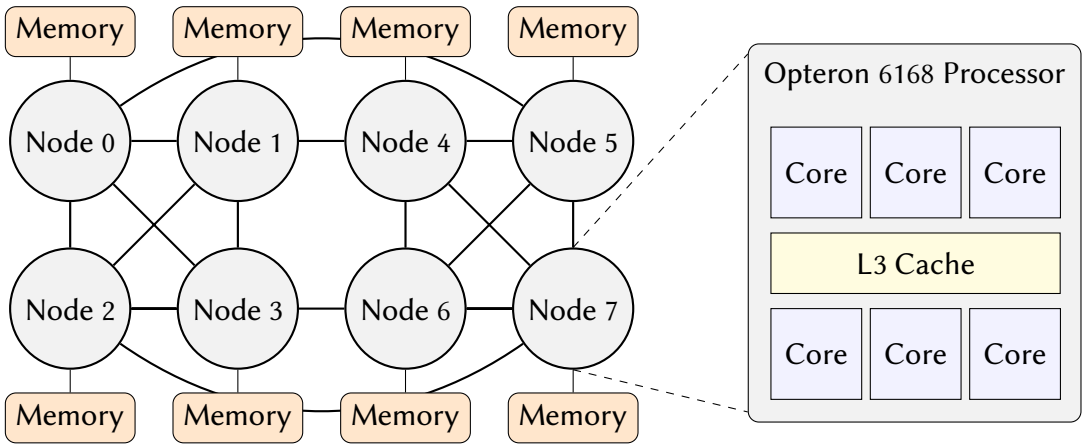


Figure 2.2.: Schematic view of our Opteron system.

be scheduled per processor core. Second, there are two processor chips per socket. Each chip consists of six SMP processing cores with 6 MB of shared L3-Cache and has 32 GB local main memory attached to it. One such chip with local memory represents one NUMA node, which means that with four sockets, eight NUMA nodes are available. As each processor chip can have four connections to other nodes, the connections do not form a complete graph. The graph is shown in figure 2.2.

An overview of specifications and SPECint_rate2006[1] benchmark results for both systems is given in table 2.1 below.

			Cores		logical CPUs		Memory (GB)		SPECint
System	Sockets	L3 (MB)	Socket	Total	Socket	Total	Socket	Total	rate2006
Xeon	4	24	8	32	16	64	64	256	1150
Opteron	4	12	12	48	12	48	64	256	706

The Opteron system has 2×6 cores with 2×6 MB L3-Cache and 2×32 GB main memory per socket. SPECint_rate2006 values taken from http://www.spec.org/.

Table 2.1.: Systems used in our experiments.

We now further investigate memory performance of both systems and compare bandwidth of local and foreign memory access.

2.1. Memory Benchmark

A widely accepted measure for memory bandwidth is the well-known STREAM[2] memory benchmark. We executed this benchmark on our test systems to get a first impression of available bandwidth. To minimize caching effects, we executed the tests with 200,000,000 elements,

[1] http://www.spec.org/
[2] http://www.cs.virginia.edu/stream/

i. e. 4.6 GB of memory were used. Table 2.2 shows the results when executing the benchmark with a single thread. As customary for the STREAM benchmark we only list the highest observed bandwidth.

	Rate, single threaded (MB/s)			
System	Copy	Scale	Add	Triad
Xeon	6,252	6,012	6,613	6,445
Opteron	5,929	5,939	6,234	6,237

Benchmark with 200,000,000 elements (4,578 MB of memory).

Table 2.2.: STREAM memory benchmark results with one single thread.

One thread does not suffice to utilize all available bandwidth. Executing the benchmark on all available CPUs thus yields 6–10 times higher results on the Xeon system, as listed in table 2.3. The STREAM benchmark uses the OpenMP library for parallelization. To simulate a more complex memory layout, we also tried striping (see section 3.1.1 on page 13) the memory over all available nodes in chunks of 3 MiB. As there are no means to tell OpenMP about our memory layout, in such cases performance degrades by up to 40% on the Xeon system.

The Opteron system has a lower single-threaded throughput. Yet, as it consists of twice as many nodes as the Xeon system, maximum bandwidth is higher by a factor of 1.4-1.5 compared to the Xeon system. However, it suffers considerably more from suboptimal memory layout, because the communication graph is not complete. As table 2.3 shows, performance can drop by 60-65% in these cases.

		Default Rate (MB/s)				Striped Rate (MB/s)			
System	T	Copy	Scale	Add	Triad	Copy	Scale	Add	Triad
Xeon	64	47,492	47,465	53,002	53,408	27,781	25,738	34,476	31,989
Opteron	48	70,396	69,988	76,848	76,938	26,888	26,862	27,904	28,385

Benchmark with 200,000,000 elements (4,578 MB of memory). *Default* means no special memory partitioning took place, whereas in the *striped* test memory was spread over all available sockets in chunks of 3 MB.
Column T denotes the number of threads used.

Table 2.3.: STREAM memory benchmark results with different memory allocations when using all available processors.

We have seen that varying memory distribution can have an impact on performance of the STREAM benchmark. While the benchmark is a good indicator for overall system performacne, it does not describe how a single node of the NUMA system performs. Hence we further investigate properties of local and foreign memory access.

Table 2.4 on the following page shows local memory write throughput for an increasing number of threads running on a single node while the rest of the system is idle. Maximum available

	Xeon		Opteron	
Threads	GiB/s	per Thread	GiB/s	per Thread
1	4.270	1.000	3.932	1.000
2	7.523	1.762	5.274	1.341
3	8.946	2.095	5.959	1.515
4	9.611	2.251	6.317	1.607
5	9.778	2.290	6.351	1.615
6	9.820	2.300	6.311	1.605
7	9.839	2.304	—	—
8	9.861	2.309	—	—

Table 2.4.: Local memory bandwidth.

bandwidth when using all CPUs on the Opteron system is only 1.6 times the bandwidth of one thread, while on the Xeon system it is 2.3 times as large.

Bandwidth of foreign memory access is shown in table 2.5. As discussed before, not all nodes can be reached in one hop on the Opteron system. We therefore measured both memory bandwidth for one and two hops on that system, while on the Xeon system all foreign nodes can be reached in one hop.

Foreign memory bandwidth when accessed across one hop is 25-40% less than local memory access on both systems. While on the Opteron system foreign memory bandwidth fluctuates between 70-75% of local memory bandwidth, on the Xeon system it systematically decreases from 76% to 62% as the number of threads increases.

Accessing memory across two hops on the Opteron system yields only 43% of local memory bandwidth from one thread. It decreases to 29% when using all six threads.

These findings suggest a significant decrease of run-time for memory-bound jobs when scheduling takes the memory layout of the application into account. Therefore the goal of this thesis is to maximize usable memory bandwidth by scheduling threads local to the required data. To

	Xeon		Opteron 1 Hop		Opteron 2 Hops	
Threads	GiB/s	per Thread	GiB/s	per Thread	GiB/s	per Thread
1	3.267	1.000	2.778	1.000	1.700	1.000
2	5.430	1.662	3.966	1.428	1.806	1.062
3	6.187	1.894	4.415	1.589	1.812	1.066
4	6.499	1.989	4.492	1.617	1.810	1.064
5	6.371	1.950	4.506	1.622	1.808	1.064
6	6.125	1.875	4.490	1.617	1.806	1.062
7	6.043	1.849	—	—	—	—
8	5.923	1.813	—	—	—	—

Table 2.5.: Foreign memory bandwidth.

that end we first develop a new user-land scheduling library, which is presented in the following chapter. It eases programming of NUMA-aware applications and enables us to measure effects of NUMA-aware scheduling.

3. NUMA-Aware User-Land Scheduling

As we saw in experiments in the preceding section 2.1, access to local memory on NUMA computers yields higher throughput than non-local memory access. Thus, execution speed of memory-bound jobs depends on the choice of the executing NUMA node. We therefore expect to increase processing speed by scheduling jobs so most of their memory access will be local.

Exact knowledge of memory layout and usage is inherently only available to the application itself. While modern operating systems offer methods to reserve memory on specific nodes, the process scheduler cannot have a-priori knowledge about which thread will access which memory block next.

Operating system schedulers can hence migrate threads to different nodes based solely on heuristics, which may incorporate page faults resulting from foreign memory access. This implies a thread has to run a certain amount of time before it can be considered for migration. Furthermore, migration can result in penalties from cache loss.

To counteract these effects, operating systems offer low-level system calls to set a thread's affinity for a given node, core or logical processor, and to allocate memory on a specific node. In Linux they are defined in the header files `sched.h` and `numa.h`. This leaves responsibility for optimal memory and thread placement to the application programmer.

Scheduling strategies known as gang scheduling and coscheduling have been implemented at operating system level. These favour scheduling a specified set or subset of related threads on neighbouring cores or nodes, in order to reduce cost of communication via shared memory blocks and to allow cache reuse between cooperating processes.

Parallelization libraries like ForestGOMP[1] (an GNU OpenMP[2] fork), MPI[3] (Message Passing Interface) or Intel's Threading Building Blocks, a parallelization library by Intel (TBB),[4] can also incorporate coscheduling or gang scheduling on the thread level.

To the best of our knowledge, none of them allow to take the application's internal memory layout into account when making scheduling decisions. This might not be a problem if the application does not lay out memory in a specific way, or if either memory layout or thread granularity is coarse enough, so that cost for thread migration is negligible. While these assumptions can be true for HPC (High Performance Computing) applications (as indicated by the STREAM benchmark results in section 2.1 on page 6), database applications are a different

[1] `http://runtime.bordeaux.inria.fr/forestgomp/`

[2] `http://gcc.gnu.org/projects/gomp/` is an open implementation of the OpenMP specification, which can be found at `http://openmp.org/`

[3] `http://www.mpi-forum.org/`

[4] `http://threadingbuildingblocks.org/`

matter. This is due to the fact that unlike in HPC most memory cells are only read once and not read and written repeatedly. It is thus important the first memory access is carried out with maximal performance already.

Before describing our approach to NUMA-Aware User-Land Scheduling in detail, we will give an overview of the library concepts and design and show a simple example of usage.

3.1. NBB Library Overview

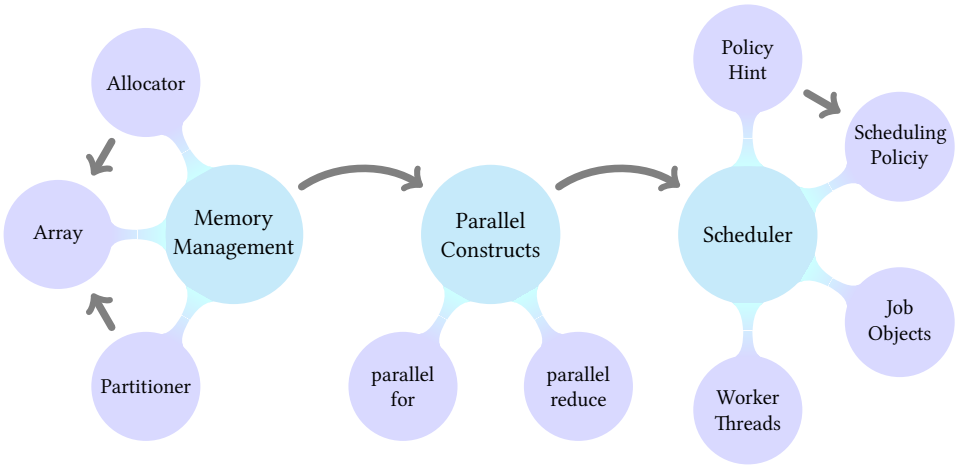


Figure 3.1.: NBB Library Components.

We have argued why it is beneficial to take knowledge about memory layout into account for scheduling. NUMA-aware scheduling leads to higher memory throughput, which is essential for database applications.

Jonathan Dees' to-be-published implementation of the TPC-H database benchmark queries use Intel's TBB library for inter-query parallelization. This library offers a convenient and platform-independent way to iterate over ranges of memory and perform non-trivial reduction operations. However there is no way to tell the scheduler which memory block to be processed is present at which node. Preliminary tests revealed the result of this ignorance is varying memory bandwidth, and one gets the best execution time only if jobs are scheduled at the right nodes by chance. See section 4.1 on page 32 for a quantification of this effect. As the TPC-H benchmark provides a collection of real-world database jobs for evaluation, one goal when writing the library was to minimize migration efforts.

To achieve this we tried to stick to the TBB interface as closely as possible. Consequently, we implemented functions like `parallel_for` and `parallel_reduce`, which appear in similar form in TBB. Admittedly, in our case one has to pass additional knowledge of the application's memory layout. We named the library NUMA Building Blocks (NBB) and implemented it in C++. As pictured in figure 3.1, NBB can be separated in roughly three parts: Memory management, scheduler and parallel constructs, which convey information about memory allocation to the scheduler. We will now give a brief conceptual overview of each component, starting with memory management.

3.1.1. Memory Management

As mentioned earlier, the NBB scheduler was designed to schedule jobs on specific nodes, given knowledge of the job's node-affinity. For this reason, users can allocate memory that is striped across given nodes in a defined manner. This definition can later be used to advise in scheduling decisions. An illustration of a striped memory region is given in figure 3.2.

Defining the affinity of a (memory) range is done by means of `partitioner` objects. The knowledge stored in the partitioner class is represented by the arrows in figure 3.2.

One versatile partitioner class `striping_partitioner` is available. It can be parametrized by stripe-size, grain-size and a vector of nodes. For memory allocation only the stripe-size and the node vector parameter are used. The *stripe-size* parameter defines how many elements should be taken to form one stripe, i.e. the largest continuous range of memory to be allocated on a single node. The memory for stripe number i will be allocated on the node given by the element at position i in the *vector of nodes*.

There may be more stripes to allocate than nodes listed in the vector. Assume the vector holds k elements. In that case the nth stripe will be allocated on the node given by the element at position $i \pmod k$ in the vector, i.e. the vector will be used cyclically.

These parameters allow for different memory layouts, the most obvious case being regular striping over all or a subset of all nodes. Note that the stripe-size could also be set to (required memory size/node count) so memory gets divided into only one stripe per node. Random striping[5] can be achieved by shuffling the vector of nodes to use. Beyond that it is of course possible to implement a custom partitioner.

The grain-size parameter is used by the scheduler to determine the minimal amount of data that should be processed as one work package, i.e. *job object*. Partitioner objects carry only

[5]Of course the resulting striping will only be random if the vector contains as many elements as there are stripes to distribute.

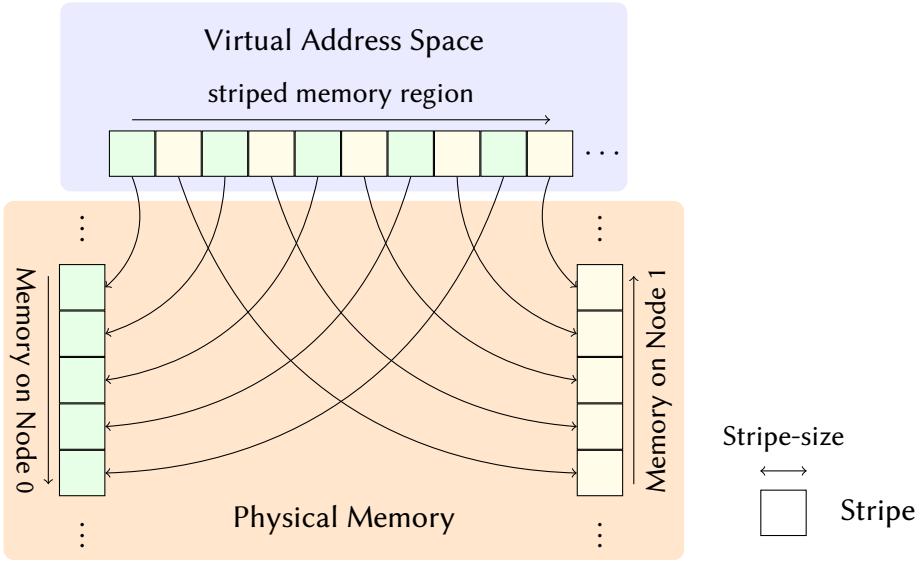


Figure 3.2.: Striping memory over two nodes.

information about memory striping (i.e. the arrows in figure 3.2 on the preceding page). Actual memory *allocation* is done by an allocator class that knows about a specific partitioner. After allocation the memory region is accessible as one contiguous block in the virtual address space of the application.

A simple container class taking advantage of this is `nbb::array`, which is basically a fixed-size vector class. When information about the striping used for allocation is required, e. g. to pass it to `nbb::parallel_for`, it can be retrieved from the array instance.

Note that due to the fact that the physically distributed memory is mapped to a continuous region in the virtual address space, the striping itself is transparent to the programmer. Thus, accessing an element using the array is as simple as with standard containers by using **operator**`[]`, e.g. `my_array[5]` and directly maps to standard pointer arithmetics.

We will now describe how the NBB scheduler was designed so it could take knowledge about memory layout into account.

3.1.2. Scheduler

Upon initialization, NBB starts one worker thread on every CPU in the system. Each worker thread then tries to retrieve a single *job object* from the scheduler.

Job objects have virtual methods `affinity()` and `execute()` (among others), the former returning on which node the job shall be executed, and the latter performing actual work. The job is responsible for returning the most suitable value for affinity, taking knowledge about memory layout into consideration. An example of how this can be done will be shown in the following section about parallel constructs.

A scheduler in NBB is a C++ template taking a "scheduling policy" class as parameter. The scheduler retrieves requests for job objects from workers and forwards them to an instance of its scheduling policy class. The policy in turn is responsible for executing a job with a worker thread on the desired node. Furthermore, some basic job management functions like waiting for a job's children are implemented in the scheduler.

Scheduling policies can define "policy hints". These may give additional information about a scheduled job, e.g. priority, precedence or additional resource constraints. Besides a very simple policy that does nothing special but to execute jobs on the desired node and only takes an *empty* policy hint there is the `static_policy` class. `static_policy` defines a set of CPUs as hint

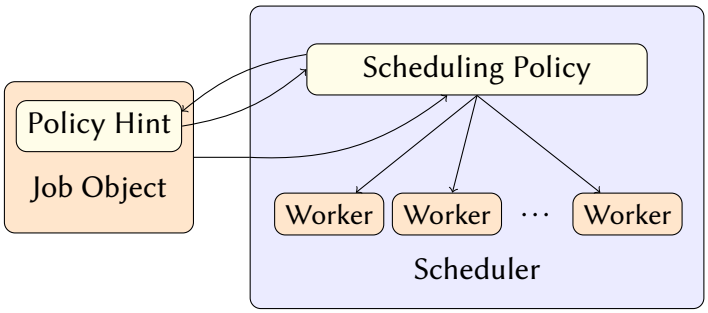


Figure 3.3.: Conceptual overview of scheduler.

and schedules the job on these CPUs only. It is helpful for evaluation of statically precomputed schedules for moldable jobs.

The design of our scheduling framework allows to test and evaluate different scheduling algorithms and concepts in practice. To that purpose one simply needs to replace the scheduling policy and use the appropriate policy hints inside the application.

The higher level parallel constructs `parallel_for` and `parallel_reduce` provide an easy way to convey information about memory striping to the scheduler. Both rely on deriving from the abstract `job` class to accomplish that goal.

3.1.3. Parallel Constructs

Apart from basic job objects described in the previous paragraph, NBB also offers `parallel_for` and `parallel_reduce` as higher level parallel constructs. `parallel_for` simply iterates over a given range of values, while `parallel_reduce` additionally performs a reduction after iteration. Both take the same arguments

- a range of values over which to iterate (usually `my_array.range()`, i.e. the range of an `array`),
- a partitioner that knows about the values affinities (usually `my_array.partitioner()`),
- a callable body object to execute,
- a scheduler and
- a policy hint that matches the policy of the scheduler.

The difference between the two lies in the required body objects (functors). The one used by `parallel_reduce` additionally needs to implement means to split an instance and join two previously split instances, similar to TBB. In contrast to that only one body instance will be used for iteration with the `parallel_for` function.

3.2. Usage Examples

To demonstrate how the library can be used for parallel programming, we will now give a simple example. We will allocate an array of size `n` and fill it with numbers from `0` to `(n-1)` in parallel. Then, we demonstrate how to sum up all elements in the array using parallel reduction.

Before we can use the parallel constructs described above we need to initialize the NBB scheduler, set up a partitioner to our needs and allocate memory.

Initialization. NBB does not automatically initialize a global scheduler, because it cannot know which policy should be used. Therefore the scheduler has to be initialized with the desired scheduling policy manually:

```
#include <nbb/scheduler.hpp>
#include <nbb/static_policy.hpp>

nbb::static_policy policy;
nbb::scheduler<static_policy> scheduler(policy);
```

Listing 3.1: Initializing `nbb::scheduler`.

First, one needs to initialize a policy object. `static_policy` does not take any parameters upon construction, but a policy supporting priorities could for example be initialized with a maximal priority value. Then a scheduler object using the desired policy can be constructed.

Memory Partitioning. Before allocating memory, a partitioner must be configured. Partitioners are used to partition ranges of indices, and (typed) ranges describe a range of indexed values of a specific type.

```
#include <nbb/partitioner.hpp>
#include <nbb/range.hpp>

typedef nbb::typed_range<size_t, int> my_range_t;

nbb::striping_partitioner<my_range_t>
  custom_partitioner(stripesize, grainsize, vector_of_nodes);

nbb::striping_partitioner<my_range_t>
  default_partitioner(range, vector_of_nodes);
```

Listing 3.2: Configuring `nbb::striping_partitioner`.

These two partitioner instances partition ranges of **int**s which are addressed by values of type `size_t`. When constructed like `custom_partitioner`, stripe-size and grain-size can be set to custom values. Used in the second form *reasonable defaults* are chosen for both parameters. At this point, those reasonable defaults are merely constant values we determined in run-time experiments. In both cases, the last parameter `vector_of_nodes` can be omitted, which means striping should use all available nodes.

Now, one can allocate memory for an array of integers with the defined striping:

```
#include <nbb/array.hpp>
typedef nbb::array<int, nbb::alloc, nbb::striping_partitioner>
       my_array_t;
my_array_t
  array(desired_number_of_elements, default_partitioner);
```

Listing 3.3: Allocating Memory using `nbb::array`.

The first template parameter defines the type of elements to be stored inside the array, the second which allocator to use (currently, there is only one available). The last parameter is

the partitioner template to use for memory striping. If no partitioner instance is passed to the constructor, a default-constructed one will be used.

Now we will can iterate over this range of memory using the aforementioned functions `parallel_for` and `parallel_reduce`.

Parallel Iteration and Reduction. To iterate over a range of partitioned values, use the `parallel_for` loop. As mentioned earlier, a callable body object needs to be passed to the `parallel_for` function. It is called `for_body_inst` in the following listing:

```
#include <nbb/parallel_for.hpp>

struct for_body {
  my_array_t &a;

  for_body(my_arary_t &a) : a(a) { }

  template <class range>
  void operator()(const range &r) {
    for (typename range::const_iterator
           end = r.end(),
           it = r.begin();
         it != end;
         ++it)
      a[it] = it;
} for_body_inst(array);

nbb::static_policy::policy_hint hint;

nbb::parallel_for(array.range(), array.partitioner(),
                  for_body_inst, scheduler, hint);
```

Listing 3.4: Using `nbb::parallel_for`.

`nbb::array` offers some convenience methods to retrieve the associated partitioner and range instances. Those are used in the listing above to set up the parallel iteration using `parallel_for`.

Before scheduling a job, one needs a scheduling hint matching the scheduling policy. After executing the loop, `array[i]` will hold the value i. This could be verified using `parallel_reduce`:

```
#include <nbb/parallel_reduce.hpp>

struct reduce_body {
  my_array_t &a;
  int sum;
  reduce_body(my_arary_t &a)
    : a(a), sum(0) { }
```

```
  reduce_body(reduce_body const &other, nbb::split)
    : a(other.a), sum(0) { }
  template <class range>
  void operator()(const range &r) {
    int local_sum = 0; // allow to allocate register
    for (typename range::const_iterator
           end = r.end(),
           it = r.begin();
         it != end;
         ++it)
      local_sum += a[it];
    sum += local_sum;
  }
  void join(reduce_body const &other) {
    sum += other.sum;
  }
} reduce_body_inst(array);

nbb::static_policy::policy_hint hint;

nbb::parallel_reduce(array.range(), array.partitioner(),
                     reduce_body_inst, scheduler, hint);

size_t n = array.size() - 1; // array holds values from 0..n
assert(body_instance.sum == (n*(n+1))/2);
```

Listing 3.5: Using `nbb::parallel_reduce`.

Note the implementations of an additional splitting constructor and a `join()` method. For the exact semantics of those please refer to section 3.3.7 on page 29. New instances of the functor are constructed by the library using the *splitting constructor*. They usually need to be initialized as identities to the `join` operation (here the sum is set to 0 for plus). It is well known that after execution the assertion on the last line holds.

3.3. Implementation Details

As the implementation of NBB makes use of C++ templates, most functionality is provided in header-only libraries. Some convenience functionality does not depend on template parameters, and is compiled as shared library. The library consists of roughly 3,500 lines of code[6], about 1,000 thereof being tests.

Various boost[7] libraries are used for thread management, synchronization and memory pool

[6] Determined using David A. Wheeler's 'SLOCCount'.
[7] http://www.boost.org/

management. Unit tests were written using the boost test framework. SCons[8] is used as build tool, Doxygen[9] to generate reference documentation from comments in the source code, and git[10] for version control.

We start discussion of the implementation details with memory allocation.

3.3.1. Memory Allocation

Mapping from virtual to physical memory addresses is done page-wise, and the operating system offers system calls to pin pages of memory to a certain node. This implies memory striping cannot be arbitrarily fine-grained.

The region of memory to manage is represented by a *range* class, while knowledge about the striping across NUMA nodes is encapsulated in a *partitioner* class. A NUMA-aware *allocator* class can use this knowledge to request the desired mapping from virtual to physical memory from the operating system. An example of how these classes can be used in conjunction is the *array* container class. We will now describe those concepts in more detail.

Range. The `blocked_range<integral>` class encapsulates a one-dimensional range $[a, b)$ of integral values. It offers methods `begin()` and `end()` to iterate over these values as well as a range-splitting constructor `blocked_range(blocked_range &o, integral split_index)`. The right half $[\text{split\_index}, b)$ of `o` is returned in the new range, while `o` is updated to contain the left half $[a, \text{split\_index})$.

Another class `blocked_type_range<integral, T>` is used to convey information about the type `T` indexed by the range. This information is needed for the *partitioner* class to determine how many elements of `T` fit on a single page.

Partitioner. Knowledge about how elements indexed by a range should be distributed across available NUMA nodes is encapsulated in a *partitioner* class called `striping_partitioner`. It can be parametrised by two scalar values: *stripe-size* and *grain-size*. For memory management only the first is used, the latter will be used when scheduling jobs that iterate over a given range. Both values are given to the partitioner in *numbers of elements*, not in bytes. Furthermore, a `std::vector<int>` enumerating the nodes to use for striping can be passed to the constructor.

The *grain-size* parameter defines the minimal number of consecutive elements to be processed as one job when using `parallel_for` or `parallel_reduce` (see sections 3.3.6 and 3.3.7 on pages 27 and 29 respectively). It is up to the scheduling policy to decide the exact range that will be processed as a single job. The `static_policy` described in section 3.3.4 on page 26 however will always split the ranges as many times as possible, i.e. until they are not larger than the grain-size.

The *stripe-size* parameter defines how many consecutive elements will be allocated on a single node. As memory can only be pinned to specific nodes in whole pages, the stripe-size will be

[8] http://www.scons.org/
[9] http://www.doxygen.org/
[10] http://git-scm.com/

adjusted so that one stripe is at least the size of one memory page. More precisely, the size of one stripe will be the least common multiple of page size, and the amount of memory required to hold the desired number of elements to avoid page thrashing.

If the vector of nodes to use for striping is not passed to the constructor, all available nodes will be used. This default vector can be accessed by the `ALL_NODES()` function. If there are more stripes to allocate than nodes listed in the vector, it will be re-used cyclically, i.e. the i-th stripe will be allocated on node `vector[i] % vector.size()`.[11]

Overall performance depends heavily on the setting of the stripe- and grain-size parameters. This is because a scheduling policy would always split a range until the size does not exceed the stripe-size to ensure local memory access. If the resulting ranges are too small, the overhead required to schedule each range for execution becomes a dominant factor in overall run-time. On the other hand the ranges should not be too large to allow homogeneous load sharing. For the default stripe-size we found 1 MiB to be a good choice in many cases, the default grain-size we found works best was 256 kiB. It is possible though to tweak both parameters manually using the `stripesize(size_t n)` and `grainsize(size_t n)` methods.

Interface of `striping_partitioner` template class.

`bool has_affinity(typed_range const &r) const` Determine whether the given range `r` resides on a single node.

`int affinity(typed_range const &r) const` If `r` resides on a single node, return which node it is.

`bool should_split(typed_range const &r) const` Return true if the range `r` spans over more than one node. Splitting the range should eventually lead to a sub-range residing on a single node. Essentially, this method returns true iff `r.size()<stripesize`.

`bool could_split(typed_range const &r) const` Return true if the range can be split into smaller sub-ranges. Essentially, this method returns true iff `r.size()<grainsize`.

`typed_range split(typed_range &r) const` Split `r` by using the range-splitting constructor.

`void nodewise_action(function &f, typed_range const &r)` This template method is used by the allocator to quickly determine the affinity of sub-ranges. `f` is expected to take three parameters. The function call `f(left, right, node)` means that the elements from the range `[left,right)` are allocated on the given node. It is ensured that the union of ranges from all function calls is equal to `r` and that the ranges do not overlap.

Those methods (except `nodewise_action`) will be used by the job objects implementing the `parallel_for` and `parallel_reduce` functionality to determine affinity of range to be processed. Available constructors and methods to set stripe- and grain-size are described in the accompanying Doxygen documentation.

Allocator. The NUMA-aware allocator class `alloc<T, partitioner>` tries to follow the C++ standard for allocators [10] as closely as possible. By an additional template parameter one can choose which type of partitioner should be used. The main difference is the `allocate`

[11] % denotes the modulus operator in C++.

method, which takes an additional parameter—the partitioner object that should be used for memory striping. If left out, a default-constructed partitioner will be used. To acquire page-aligned regions of memory, the glibc[12] library function `posix_memalign` is used. After memory allocation, the `nodewise_action` method of the partitioner object to pin the memory pages to the desired nodes using the `numa_tonode_memory` library call provided by libnuma[13].

Array. A simple container class that takes advantage of the concepts and classes introduced above is the `array<T, allocator, partitioner>` template class. It offers a straightforward interface to construct an array holding a given number of elements that are spread over the nodes using a specific partitioner.

As with standard containers it can be accessed using iterators or the **operator**`[]`. In addition to that, the two methods `range()` and `partitioner()` are available. These offer a convenient way to iterate over the array using `parallel_for` and `parallel_reduce`.

3.3.2. Job Class

```
template <class Scheduler>
class job<Scheduler> {
 protected:
  job(policy_hint_t &policy_hint); // construct a root job with
      policy hint
  job(job *parent); // construct a child job with policy hint from
      parent

 public:
  virtual bool has_affinity() const = 0;
  virtual int affinity() const = 0;
  virtual bool should_split() const = 0;
  virtual bool could_split() const = 0;
  virtual job* split() = 0;
  virtual void execute() = 0;
...
```

Listing 3.6: Constructors and pure virtual methods of `job` class.

Schedulable entities in NBB are called *job* and are derived from the abstract template class `job`. Jobs have a reference to a designated policy hint which determines how this job and its children should be scheduled. As the scheduling policy class affects the type of the scheduler (see section 3.3.3), the job base class is implemented as a template class.

Constructors of `job` template class.

[12] http://www.gnu.org/software/libc/
[13] http://oss.sgi.com/projects/libnuma/

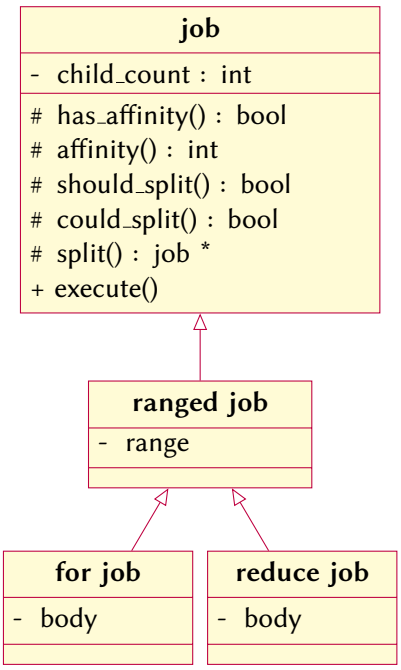


Figure 3.4.: Hierarchy of job classes.

`job(policy_hint_t &policy_hint)` Constructor to create root jobs. `policy_hint` parameter is stored as a reference and made available to scheduling policy.

`job(job *parent)` Constructor to create child jobs, which will use the same `policy_hint` as the `parent` job. Child count of the `parent` job is increased.

Virtual interface of `job` template class.

`bool has_affinity()` Return true iff calling `affinity()` returns the node id on which this job should be scheduled.

`int affinity()` Return the node id on which this job should be executed, if `has_affinity()` returns true. If it returns false, any valid node id (e.g. `0`) must be returned.

`bool should_split()` Determine whether calling `split()` will eventually return job objects that have affinity for a specific node, i.e. which return true when `has_affinity()` is called.

`bool could_split()` Determine whether this job object could be split into smaller jobs even if `has_affinity()` already returned true. Implementations of this method should honor the implication `should_split()` $\Rightarrow$ `could_split()`.

`job_t *split()` If `could_split()` returns true, split the job into two smaller jobs. The returned job should become a child of **this** and use the same scheduling hint. This can easily be accomplished by using the child job constructor mentioned above. Calling `execute()` on both jobs (in any order or concurrently) after calling `split()` must yield the same result as executing the original job object.
If `could_split()` returns false, behaviour of this method is not defined. Thus, the actual implementation may be empty (i.e. **return** `NULL`) for jobs that cannot be split.

`void execute()` Actual action to be performed. When a job is scheduled for execution by a worker thread, this method will be called once. Because all scheduling takes place in user-space, blocked worker threads cannot retrieve another job object to work on. In that case the operating system scheduler will likely free the allocated CPU. Hence, this method should not make any potentially blocking system calls.

All virtual methods of the abstract `job` class are declared as *pure virtual* methods, i.e. no default implementation is defined in the base class. Therefore all virtual methods of the job class must be overridden.

Ranged Job Class

The `ranged_job` template class provides an implementation for some of the virtual methods declared in the `job` class. It works as a simple example on how the virtual methods could be implemented. `parallel_for` (section 3.3.6 on page 27) and `parallel_reduce` (section 3.3.7 on page 29) both rely on deriving from the `ranged_job` class.

```
template <class Range, class Partitioner, class Scheduler>
class ranged_job : public job<Scheduler>
{
  ranged_job(); // hide default constructor
 protected:
  Range _range;
  Partitioner const &_partitioner;

  // construct a root job with policy_hint
  ranged_job(Range const &r, Partitioner const &p, typename
      Scheduler::policy_hint_t &policy_hint);
    : job<Scheduler>(policy_hint), _range(r), _partitioner(p) { }

  // construct a child job which uses policy_hint from parent
  ranged_job(Range const &r, Partitioner const &p, ranged_job *parent);
    : job<Scheduler>(parent), _range(r), _partitioner(p) { }

 public:
  bool has_affinity() const { return _partitioner.has_affinity(_range); }
  int affinity() const { return _partitioner.affinity(_range); }
  bool should_split() const { return _partitioner.should_split(_range); }
  bool could_split() const { return _partitioner.could_split(_range); }
};
```

Listing 3.7: The `ranged_job` class.

Note that this is still an abstract class, because `execute()` has no implementation yet. This is why the `split()` method cannot be implemented at this point. It must be implemented by classes that derive from this class. Implementing this method usually involves calling the

child-job constructor. A simple example how the method could be implemented is shown in listing 3.8 below, which is an excerpt of the `for_job` class used to implement the `parallel_for` construct. It relies on the `split()` method of a `partitioner` object to divide the given range. The child-job constructor leaves most initialization to the corresponding constructor of `ranged_job`.

For more details about the implementation of the `parallel_for` loop also see section 3.3.6 on page 27.

```
template <class Range, class Partitioner, class Body, class Scheduler>
class for_job : public ranged_job<Range, Partitioner, Scheduler> {
  Body &_body;
  // constructs a child for job with given range
  for_job(Range const &r, Partitioner const &p, Body &b, for_job *parent)
      : ranged_job<Range, Partitioner, Scheduler>(r, p, parent), _body(b)
  { }
 public:
  for_job *split() {
    return new for_job<Range, Partitioner, Body, Scheduler>(
        this->_partitioner.split(this->_range), // split range and return one part
        this->_partitioner,
        _body, // shared reference of body object
        this); // parent job
  } ...
```

Listing 3.8: Excerpt from `for_job` class.

3.3.3. Scheduler Class

As mentioned before, one design goal was to provide a framework that facilitates testing different scheduling strategies. For this purpose the scheduler is implemented as a C++ template class taking a scheduling policy class as template parameter. The scheduler template class manages

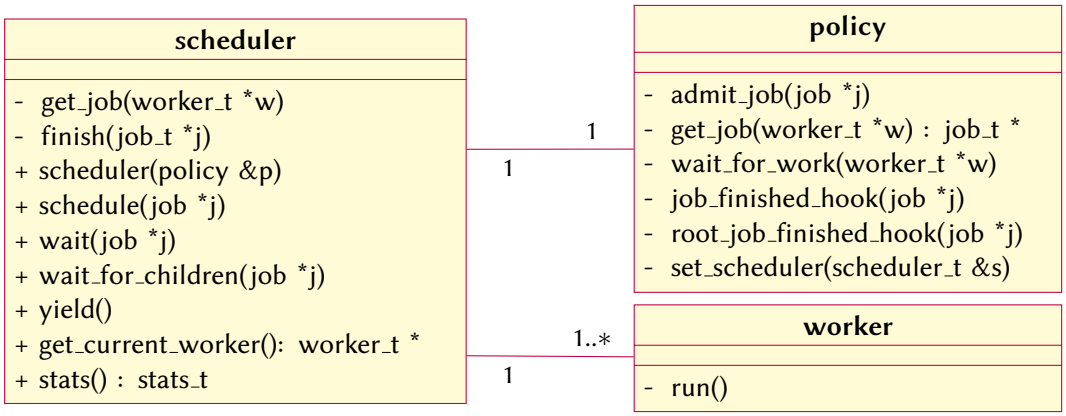


Figure 3.5.: (Incomplete) class diagrams of scheduler, policy and worker class.

the worker threads and provides common functionality, like waiting for a job to finish. Upon construction the worker threads are started.

The constructor of the scheduler class takes an instance of the scheduling policy class (which itself might require some run-time initialization) as parameter. All requests requiring scheduling decisions are forwarded to the specified scheduling policy instance. Those requests include workers asking for job objects to process or user requests to schedule jobs.

Methods implemented by `scheduler<policy>` template class.

`scheduler(policy &p)` Constructor to create a scheduler object with the given policy. Starts up worker threads and initializes the policy object by calling the `policy.set_scheduler` method (see below).

`void schedule(job_t *j)` Schedule a job for execution. The call is forwarded to the scheduling policy.

`void wait(job_t *j)` Wait for a job `j` to finish. Returns when all children of `j` and `j` itself have been processed. Only one call to this (*or the* `wait_for_children`) method per job is allowed. If called from a worker thread, other job objects may be processed while waiting.

`void wait_for_children(job_t *j)` Wait for children of job `j` to disappear. Returns when all children of j have been processed. Only one call to this (*or the* `wait`) method per job is allowed. If called from a worker thread, other job objects may be processed while waiting.

`void yield()` Yield the current worker thread to another job object. Tries to process another job using the current worker thread and returns later.

`worker_t *get_current_worker()` Return worker class associated with calling thread, or `NULL` if the calling thread is not managed by the NBB scheduler object.

`stats_t &stats()` Return collected statistics if enabled via the `NBB_COLLECT_STATS` compile time parameter.

The remaining methods (e.g. to get a worker class by its id) are documented in the provided Doxygen documentation. The `finish` method used by workers to mark a job as finished is declared as private. It wakes up threads waiting for the job to finish and forwards the call to the scheduling policy in case further actions are required. The methods `get_job` and `wait_for_work` (also marked private) will be called by worker threads and are forwarded to the scheduling policy. The interface of a scheduling policy class required by the scheduler is described below.

Methods required from scheduling policies.

`void admit_job(job_t *j)` Admit a job for scheduling. This method is called by the scheduler when a new job `j` is scheduled for execution.

`job_t *get_job(worker_t *w)` Return a job for worker `w`. Called by worker threads (through the scheduler class) to retrieve a job object to execute. Threads will pass the associated worker class which carries information about the assigned node and CPU. The `execute()`

method of the returned job object will be called. If there is no work available, the scheduling policy may return `NULL`.

`void wait_for_work(worker_t *w)` Wait for new work to arrive for worker `w`. Will be called if `get_job` returned `NULL`.

`void job_finished_hook(job_t *j)` Called by the scheduler when job `j` is finished. If no special action is required in such cases, the implementation may be empty.

`void root_job_finished_hook(job_t *j)` Called by scheduler when a root job `j` is finished. If no special action is required in such cases, the implementation may be empty.

`void set_scheduler(scheduler_t *s)` Called once before scheduling starts, but after all threads have been started. Useful if the policy needs to initialize data structures depending on the scheduler's configuration (e.g. the number of available threads).

The static scheduling policy is a basic example of how such policies can be implemented. It is discussed in the following section.

3.3.4. Static Scheduling Policy

```
class static_policy : boost::noncopyable {
 public:
  // default constructor
  static_policy();

  class policy_hint {
   public:
    // default constructor (job allowed to run on any unused CPU)
    policy_hint()
        : _cpus(EMPTY_VECTOR()), _strict(false)
    { init_queues(); }

    // restriction constructor (job runs on only given CPUs, if strict or on
        given and unused CPUs if not strict)
    policy_hint(std::vector<int> const &cpus, bool strict)
        : _cpus(cpus), _strict(strict)
    { init_queues(); }

    // copy constructor (a policy hint may only be used for one root-job)
    policy_hint(policy_hint const &o)
        : _cpus(o._cpus), _strict(o._strict)
    { init_queues(); }
  ...
```

Listing 3.9: Constructors of `static_policy` and its associated `policy_hint`.

A basic scheduling policy that can be used in NBB is the `static_policy` class. It allows to define exactly which CPUs can be used to work on a job. While the policy constructor does

not take any arguments, a vector of CPUs (identified by their ID number) must be handed over to the associated `policy_hint` constructor. Upon construction of a `policy_hint` instance, one work-queue per node is created. When the policy is asked to schedule a child job that has affinity for a specific node, it is put into the according work-queue. Others will be distributed across nodes using round-robin. One lock per node and root-job is used to protect the queues from concurrent access.

When a worker thread calls the `get_job(worker_t *w)` method (through the scheduler) it retrieves a job from the appropriate queue. If there is no work available on that node, it tries to get a job object from another node's queue. This behaviour can be switched off by the `NBB_DONT_STEAL_FROM_FOREIGN_NODE` preprocessor macro. It can be set globally in the SCons build scripts if jobs should never be scheduled on foreign nodes.

3.3.5. Worker Threads

Upon initialization of the scheduler one *worker thread* is spawned for each available CPU. The `sched_setaffinity` Linux system call is used to fix each thread to one specific CPU. Furthermore, the `numa_set_preferred` function is used to ensure executed jobs allocate new memory on their executing node.

As NBB was implemented as a user-land library, the NBB scheduler cannot influence thread scheduling of the operating system in a direct manner. Therefore, NBB can neither inhibit nor easily detect preemption of worker threads. Note, that this can lead to starvation issues when other processes run on the same system.

After a worker thread was started it repeatedly calls the `get_job` method of the scheduler, which forwards this call to the corresponding policy. If a job object `j` is returned, `j.execute()` is called. Afterwards, the method `finish(j)` of the scheduler is invoked to indicate that the job object has been processed. When there is no job available for the requesting worker thread, the method call might return `NULL`. In that case, the `wait_for_work` method is called, which may block the worker thread until there is a new job object available.

3.3.6. Parallel Iteration

```
template<class Range, class Partitioner, class Body, class
    Scheduler>
void nbb::parallel_for(
  Range const &r,
  Partitioner const &p,
  Body const &b,
  Scheduler &s,
  typename Scheduler::policy_hint_t &h )
```

Listing 3.10: Function signature: `parallel_for`.

Parameters of `parallel_for`.

`r` Range for parallel iteration, usually something like `my_array.range()`.
`p` Partitioner object to determine node affinity of sub-ranges. Usually the one used to partition the range `r`, like `my_array.partitioner()`.
`b` Instance of functor implementing **`operator()(Range const &r)const`** method.
`s` Scheduler object to use.
`h` Hint for used scheduling policy.

```
struct my_for_body {
  void operator()(Range const &r) const; // parallel iteration
};
```

Listing 3.11: Methods of functor class required for `parallel_for`.

Interface required for `parallel_for` functors.

`operator()(Range const &r) const` Operator for parallel iteration. This method must allow multiple concurrent calls.

The `parallel_for` loop can be used to iterate over a range of values in parallel. When calling the `parallel_for` template function, an instance of `for_job` which is a subclass of `ranged_job` will be scheduled. Range-splitting and assignment of node-affinity according to the given partitioner is done in the `ranged_job` class as described in section 3.3.2 on page 21. The `execute()` method is overridden to call `b.`**`operator()`** with the range to be processed. As the `execute` method is declared **const** in `Body`, all worker threads can share the same instance of `Body`. No guarantees are made regarding order of execution.

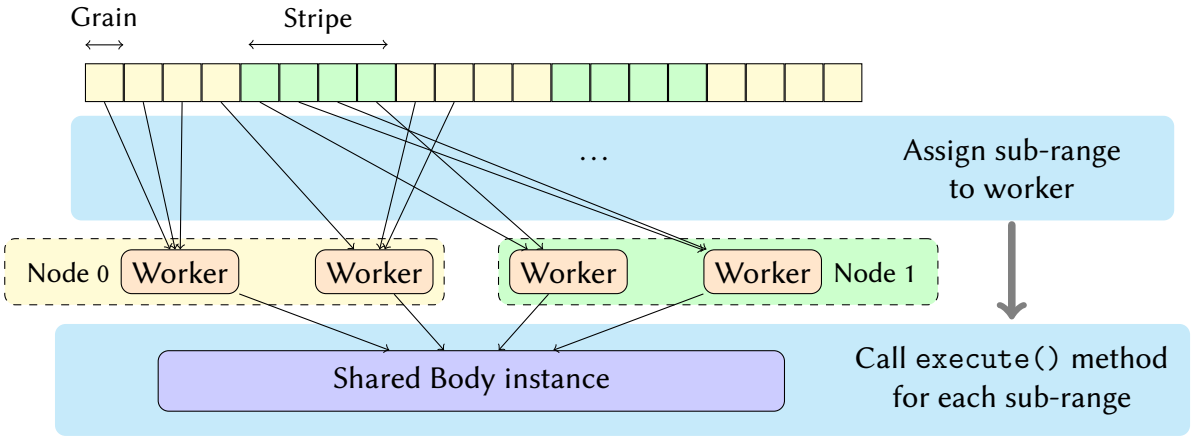


Figure 3.6.: Iterating over a striped memory region.

3.3.7. Parallel Reduction

```
template<class Range, class Partitioner, class Body, class
    Scheduler>
void nbb::parallel_reduce(
  Range const &r,
  Partitioner const &p,
  Body &b,
  Scheduler &s,
  typename Scheduler::policy_hint_t &h )
```

Listing 3.12: Function signature: `parallel_reduce`.

Parameters of `parallel_reduce`.

- **`r`** Range for parallel reduction, usually something like `my_array.range()`.
- **`p`** Partitioner object to determine node affinity of sub-ranges. Usually the one used to partition the reduction range `r`, like `my_array.partitioner()`.
- **`b`** Instance of functor to use. The required interface is shown in listing 3.13 and described below. This instance will also hold the result of the reduction. In the current implementation **`operator()`** is required to be associative and commutative.
- **`s`** Scheduler object to use.
- **`h`** Hint for used scheduling policy.

```
struct my_reduce_body {
  my_body(my_body const &other, nbb::split); // splitting
      constructor
  void operator()(Range const &r); // parallel iteration
  void join(my_body const &lhs); // join operation
};
```

Listing 3.13: Methods of functor class required for `parallel_reduce`.

Interface required for `parallel_reduce` functors.

`Body(Body const &other, nbb::split)` Splitting constructor. Each worker thread will create one instance of the functor object using this constructor. Newly created instances usually has to be the neutral element to the `join` operation.

`operator()(Range const &r)` Operator for parallel iteration. In contrast to the functor usage of `parallel_for` (compare section 3.3.6 on page 27), in parallel reduction no concurrent calls to this method will be made. Instead, multiple functor instances will be created using the splitting constructor. No guarantees about processing order of `r` are made.

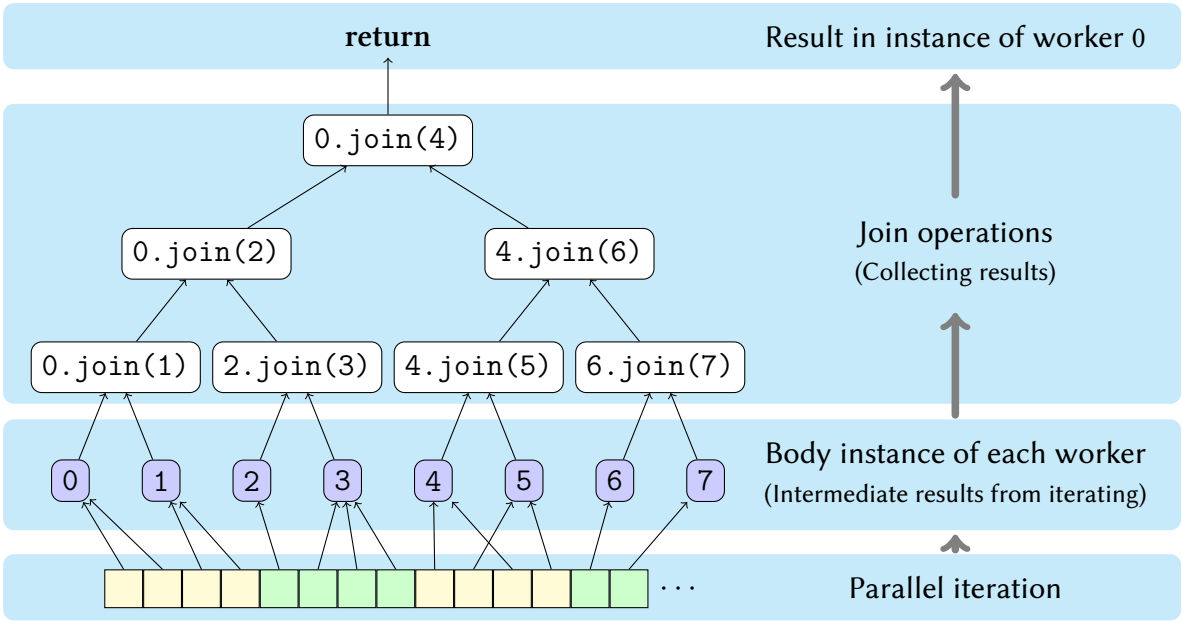


Figure 3.7.: Reduction of `parallel_reduce` body instances.

`join(Body const &rhs)` Join method. After each sub-range of `r` has been processed using **`operator()`**, the instances of `Body` will be joined as depicted in figure 3.7.

Parallel reduction over a given range can be performed using the `parallel_reduce` function. The first step in a reduction is the same as for `parallel_for`: a range of items is processed in parallel using a C++ functor object of type `Body`. But it differs in the fact that multiple instances of the functor may be created using a splitting constructor. Concurrent calls to `Body.operator()` will always be made using different instances of the `Body` functor. This ensures mutual exclusion when accessing data members of `Body` from **`operator()`**. It is achieved by creating a new instance of `Body` for every worker thread involved in the reduction.

Thus, concurrent calls to the splitting constructor may occur, even while a worker is executing **`operator()`**. However this is usually not a problem, as the splitting constructor initializes the new instance as neutral element to the `join` operation.

After all elements have been processed, intermediate results have been accumulated in each created instance of `Body`. Now, `join` operations are executed as depicted in figure 3.7. Assume the amount of work for one join operation of two body instances depends only on the number of join operations that have been executed with those instances so far. Then the amount of work to perform in one call to the `join` method is the same for all join-nodes in one level of the tree. This way work is balanced perfectly among the join operations.

A call to `lhs.join(rhs)` should update the left-hand-side instance `lhs` to include the results stored in `rhs`. The final result will be stored in the instance of `Body` that was passed to the `parallel_reduce` function call. While an instance of `Body` is reserved for a specific worker during iteration to ensure exclusive use, the `join` operation can be executed by any available worker. However, it is guaranteed that only one concurrent call to the `join` method will be made per instance.

4. TPC-H Database Benchmark

The TPC-H benchmark [17] is a well-known benchmark for database systems, focusing on queries as they appear in business scenarios. It describes eight database tables, which are sized according to a *scaling factor* s. The total size of all data is about s GiB. Furthermore, 22 database queries are listed as SQL statements. These queries are then used to define two benchmark tests: *Power Test* and *Throughput Test*.

In this chapter, we will concentrate on the *Power Test*, where only one query is executed at a time. Its focus is to minimize overall execution time, i.e. *makespan*. The *Throughput Test* defines how multiple query streams shall be executed concurrently to measure database *throughput*. We will point out some thoughts on this test in Section 5.4 on page 46 and in the future work section in chapter 6.

Jonathan Dees wrote an in-memory implementation of those benchmark queries. As mentioned before, Intel's TBB library was used for intra-query parallelism. In this implementation all data is kept in memory, thus utilizing memory bandwidth as efficiently as possible can be expected to be an important factor for run-time performance of memory intensive queries.

Generating the Database. The TPC ships a tool called `dbgen` to generate the tables required for the TPC-H benchmark. Before building, one needs to edit the shipped `makefile.suite` to specify some system-dependent settings. After generating the plain-text files with dbgen, the data needs to be converted in order to be used with Jonathan Dees' TPC-H implementation. Use the `queries` binary to achieve this.

```
$ make -f makefile.suite    # build dbgen
$ dbgen -s 1 -v             # generate database for scale factor 1
$ queries --build           # build and transform tables in three steps
$ for i in $(seq 1 3); do queries --transform $i; done
```

The TPC-H benchmark specifies a *scale factor* that roughly scales the database size in gigabytes. Expected query results for all queries are provided for scale factor 1, which mainly serves for validation purposes. To allow for better comparability of results, only the scale factors 1, 10, 30, 100, …, 30,000 and 100,000 are specified as being valid for testing.

All benchmark results discussed in the remainder of this thesis were obtained from running the queries for the scale factor 100 database. This was the maximum size that would fit into main memory and thus best allows us to observe effects of memory bandwidth.

After generating the required database tables, we can now discuss the results from running the benchmark queries.

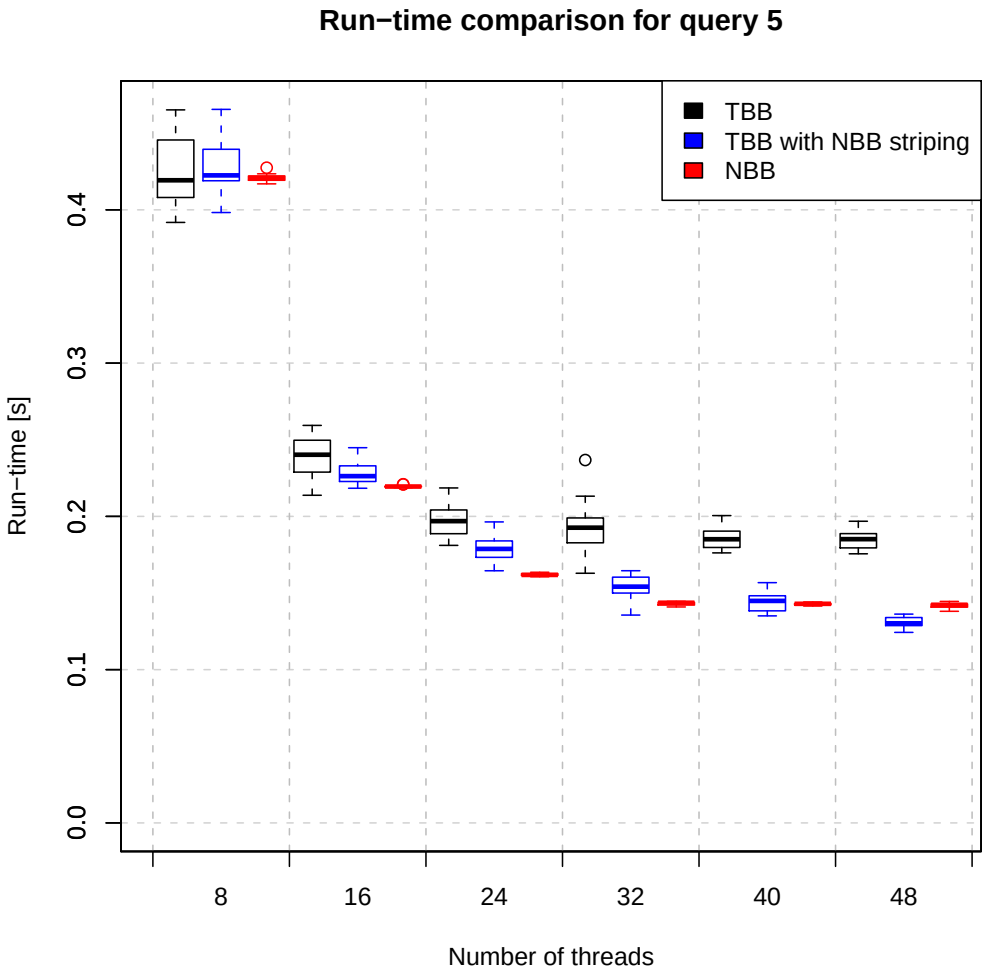


Figure 4.1.: Run-time comparison of query 5 on our Opteron system with TBB and NBB.

4.1. TPC-H Power Test

The power test is part of the TPC-H benchmark specification. The goal of the power test is to execute each TPC-H query in order and measure the time needed to complete all queries. Consequently, the power test result can be optimized by minimizing each query execution time. Therefore we will now consider each query separately.

Using the implementation based on TBB as baseline, we find two main results: First, even when using the TBB scheduler with memory striped by the NBB framework, execution times of the benchmark queries decreased in some cases. One example for this is query 5.

Figure 4.1 shows run-time results for execution of that query with TBB, TBB with memory striping and the NBB scheduler on our Opteron system. With 24 threads, i.e. 3 threads per NUMA-node, the striping provided by the NBB library boosts TBB's average performance by about 10%, while NBB gains 18% performance. Using all six threads per node, TBB with memory striping runs 30% faster than TBB without striping, while performance gain of NBB over TBB is slightly lower at 23%.

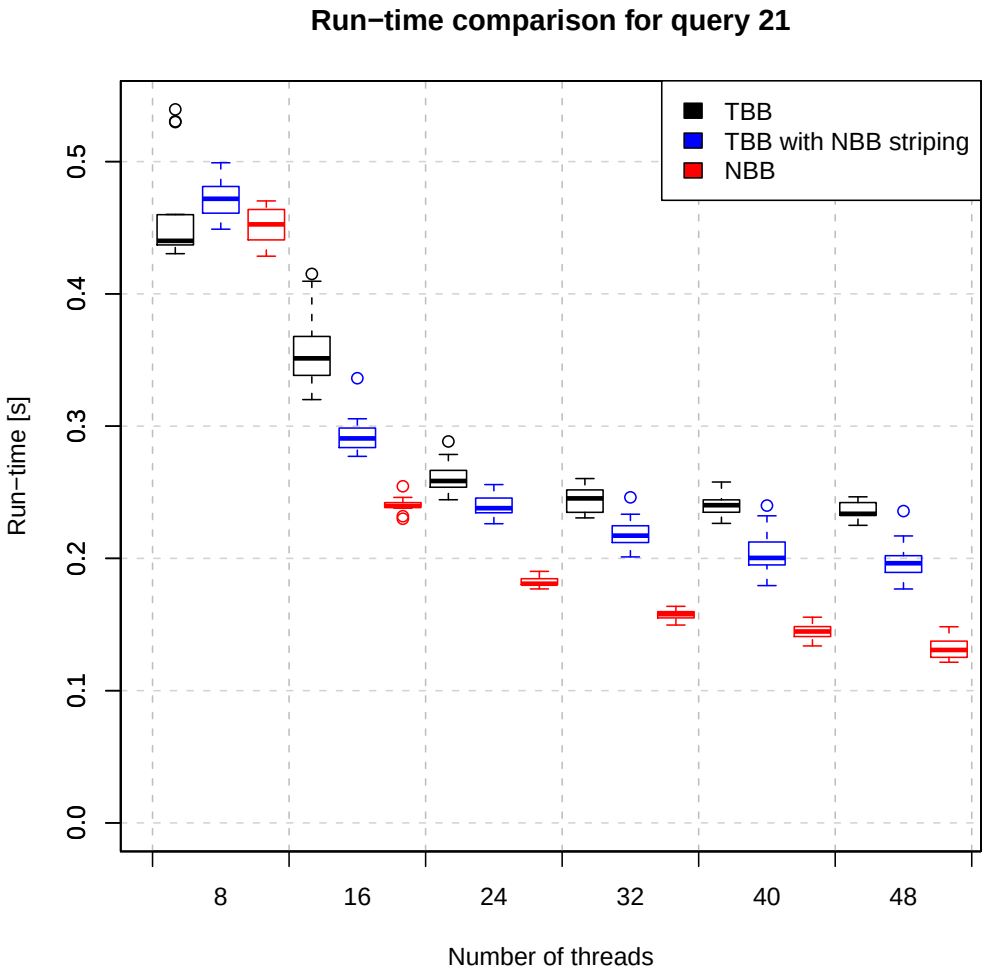


Figure 4.2.: Run-time comparison of query 21 on our Opteron system with TBB and NBB.

The reason why even TBB can benefit from uniform memory striping at all is that the TBB scheduler tries not to switch threads when processing adjacent data. Thus, a thread has a higher chance to process data residing on the same node repeatedly, and might even be migrated to the correct node by the operating system scheduler.

The previously described drop in performance gain for NBB could indicate non-local memory access during the execution of query 5. We back this claim by analyzing the run-time behaviour of query 21, which is shown in figure 4.2. Its run-time is roughly the same as for query 5, thus we can rule out effects resulting from scheduling overhead. Nevertheless, we observe performance gains of NBB over TBB ranging from 2% on one processor per node to 44% for all six processors per node. This was also the maximal performance gain observed on the Opteron system. Meanwhile, performance gained by TBB due to memory striping is only between -2 and 16%. It is therefore important how the accessed data structures are laid out in memory, which may not always be a simple task.

Our second observation is a decrease in run-time deviation. Compared to NBB the run-time

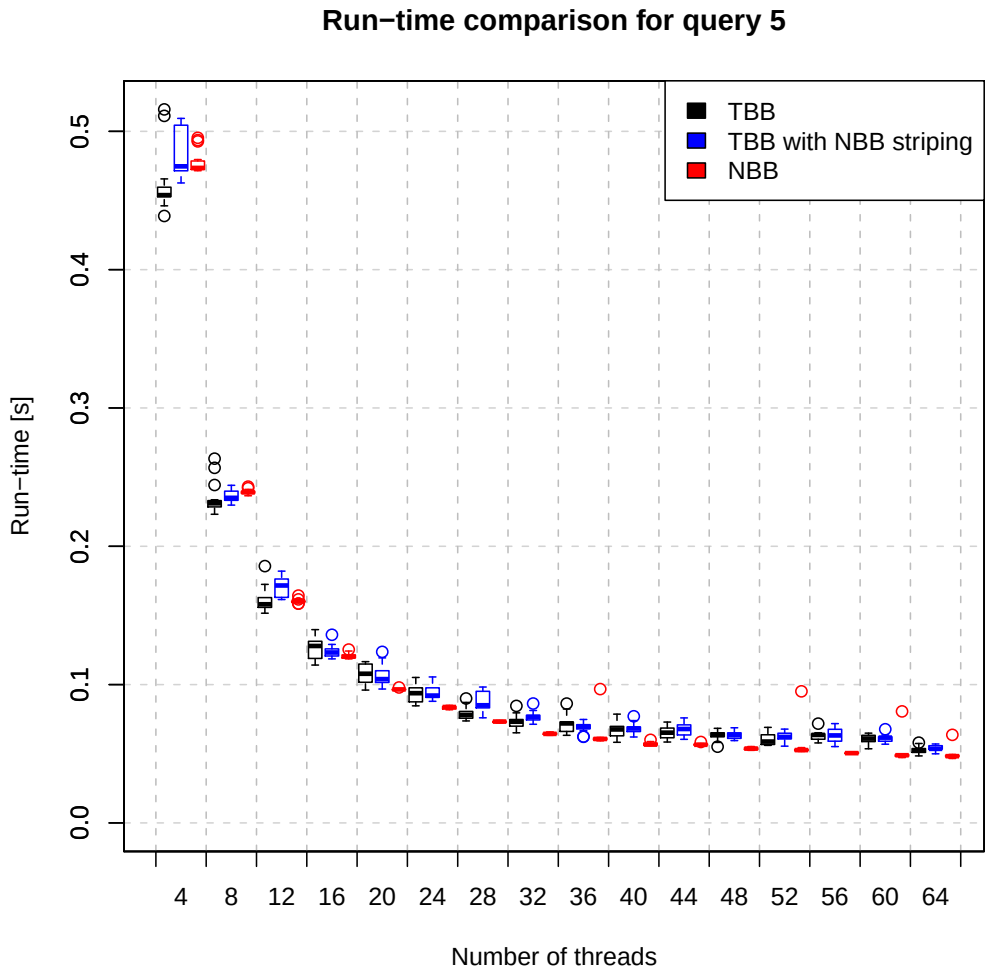


Figure 4.3.: Run-time comparison of query 5 on our Xeon system with TBB and NBB.

of a single query when executed with TBB can deviate in a relatively large range, even when memory is striped using NBB. This is caused by the aforementioned fact that TBB may schedule job execution on the most advantageous NUMA nodes by chance.

Both effects can be attributed to the knowledge about memory layout, as the TBB scheduler is quite sophisticated and has been thoroughly optimized. Scheduling jobs which are not memory bound or run only for a short amount of time therefore performs better with TBB than with NBB (also see the following section 4.2).

Another thing to note is that the described effects are hardware-dependent. Figure 4.3 shows the run-times of query 5 as observed on the Xeon system. Remember the Xeon system consists of four nodes, and each pair of nodes is interconnected. Thus, effects of foreign memory access are not as significant as they are on the Opteron system, and NBB outperforms TBB with memory striping when more than 3 CPUs per node are in use. The mean performance gained for NBB over TBB range from -4% to 19% when executing query 5.

The maximum performance gain was observed when executing query 13 on the Xeon system with one CPU per node. Its run-time is shown in figure 4.4. While memory striping shows

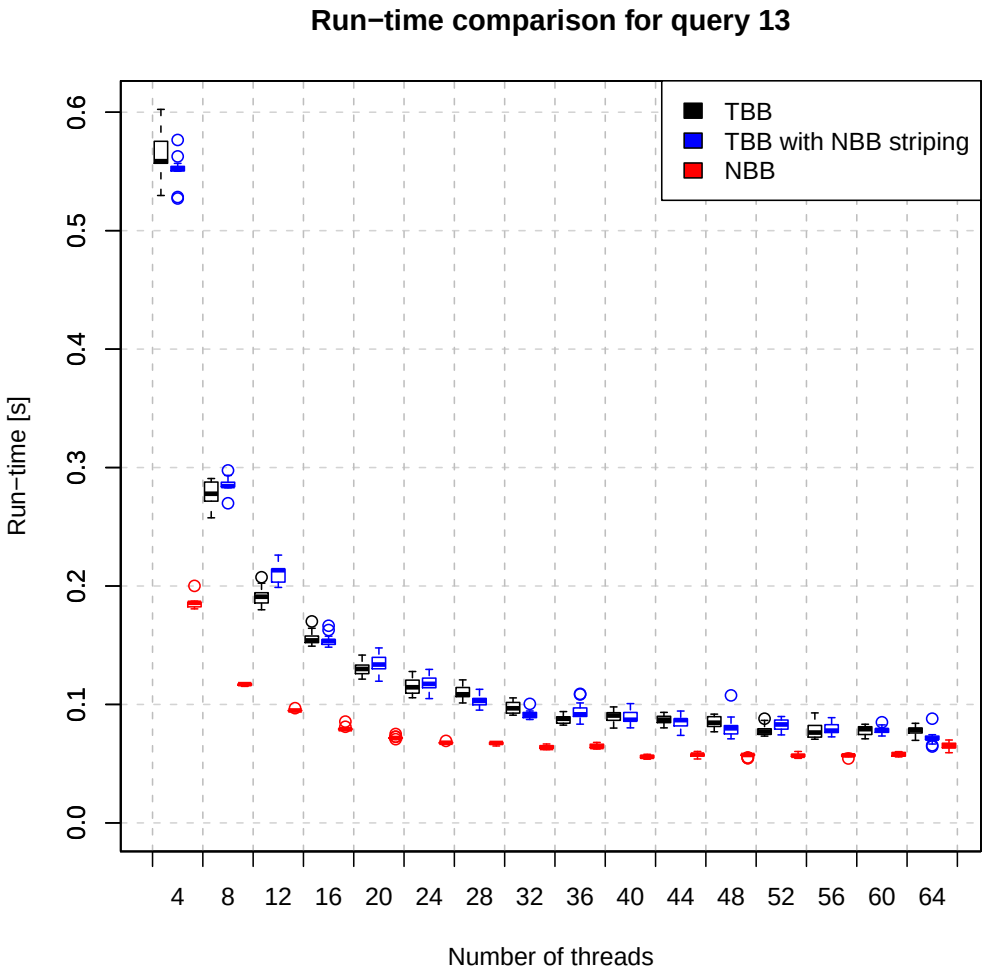


Figure 4.4.: Run-time comparison of query 13 on our Xeon system with TBB and NBB.

only a 2% performance gain for TBB with memory striping, the NBB scheduler manages to gain 67% in performance. The gap decreases as more processors are alotted, as this helps TBB utilize more memory bandwidth. Still, with 16 CPUs per node (i.e. 64 CPUs in total) NBB is 16% faster than TBB, while TBB with memory striping is only 8% faster. We also observe that NBB run-time does not decrease significantly after using 5 CPUs per node, indicating memory bandwidth becomes the bottleneck for execution speed at that point.

4.2. Observed Problems

Our experiments revealed some issues with the software we used for evaluation as well as with NBB itself. Query 3 displayed very odd changes in run-time when increasing the number of allotted processors, as can be seen in figure 4.5 on the next page. Given the more predictable behaviour of other queries this can be attributed to the implementation of query 3. We thus discard this query from our further studies.

Another limitation is that our NBB scheduling framework is not as highly tuned as TBB.

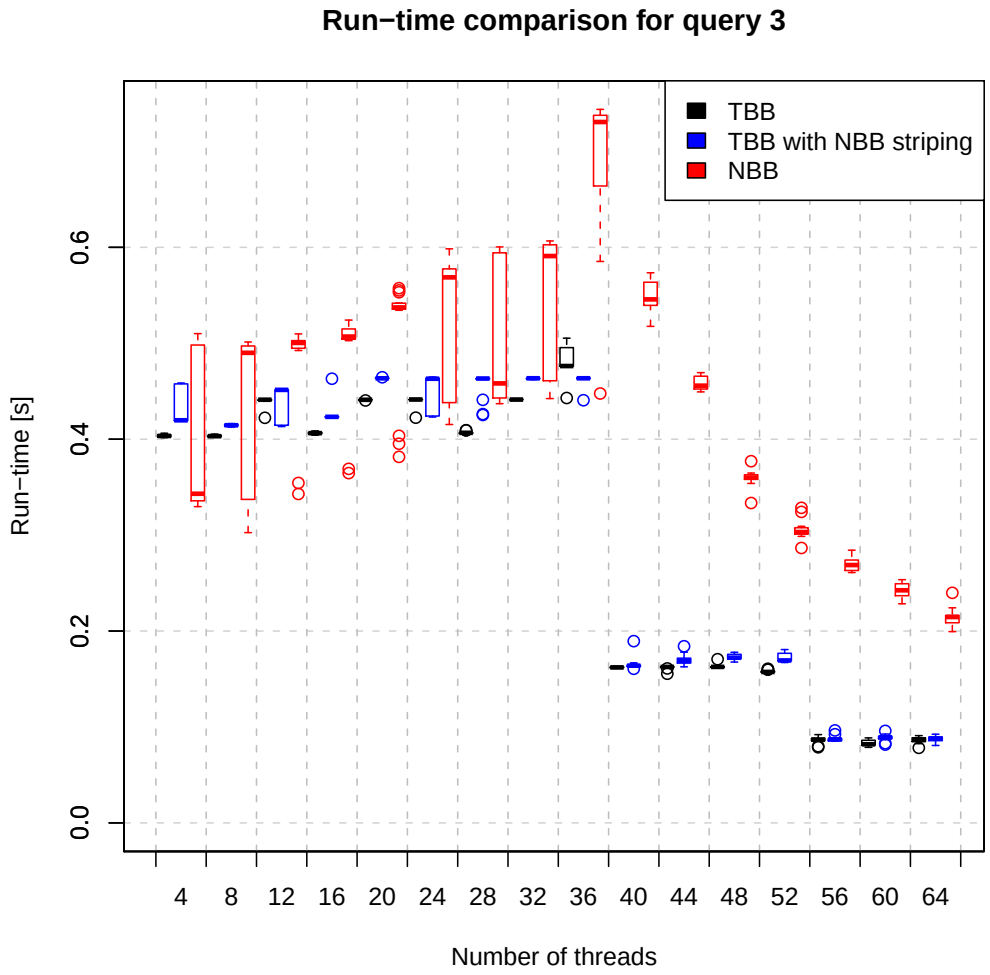


Figure 4.5.: Run-time comparison of Query 3 on our Xeon system.

Accomplishing that lies beyond the scope of this thesis. It is not only a problem of the scheduler itself, but also that in the current implementation of NBB the *grain-size* is set as a constant value. Although it can be chosen by the application programmer before scheduling of a job starts, poor choices for that parameter can lead to high scheduling overhead. While Jonathan Dees introduced heuristics in the TPC-H query implementation to choose an appropriate value, we expect dynamically choosing the job size as TBB does to yield a performance boost.

These limitations of NBB appear only for relatively short-running jobs. Run-times of queries 2, 4, 11, 17, 19 and 20 are in the range of 1 to 10 milliseconds and display effects of overhead in NBB. This is depicted using query 17 as example in figure 4.6 on the opposing page.

4.3. Summary of the Benchmark Results

Summing up, we conclude NUMA-aware scheduling can yield high performance gains for memory-bound jobs. Maximum gain can only be achieved by carefully arranging the memory layout. This is not always easy to accomplish, even with the utilities provided by the NBB

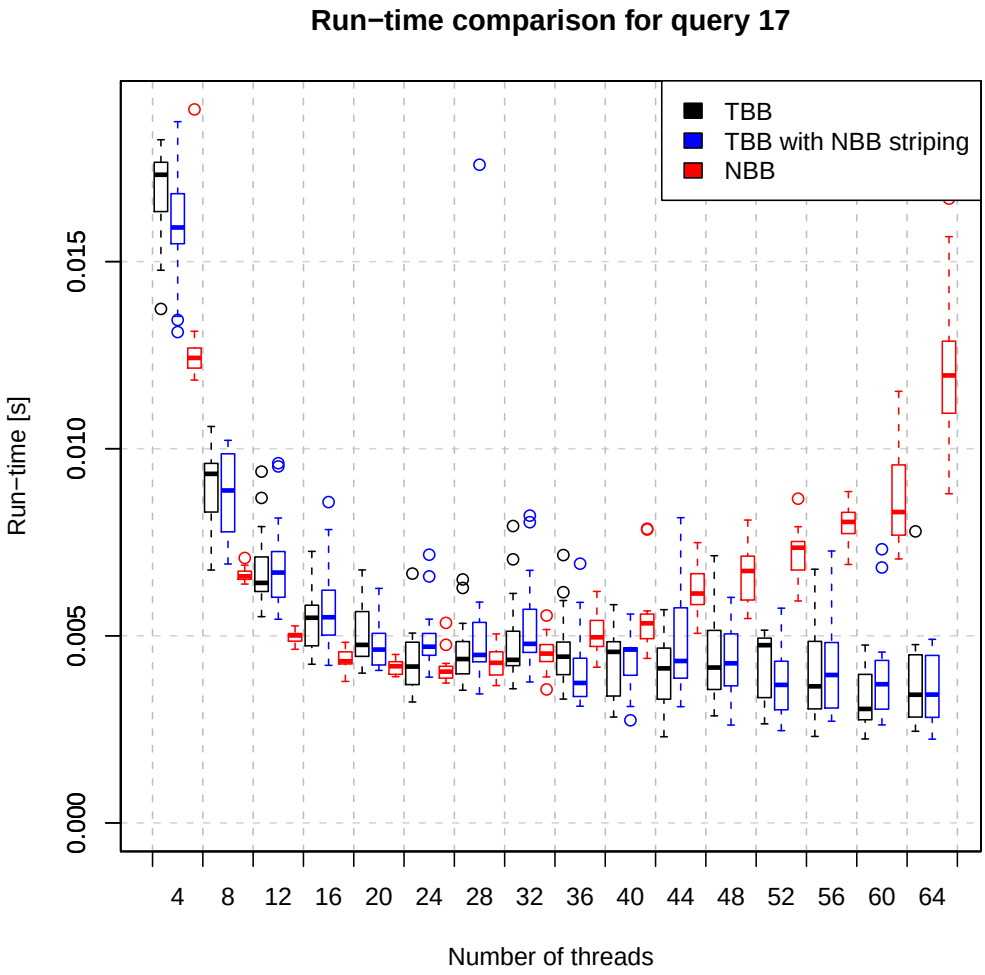


Figure 4.6.: Run-time comparison of short-running Query 17 on our Xeon system.

framework.

Recapitulating the benchmark results, the question arises whether we can find a simple model to predict the run-time behaviour and effects of interactions between bandwidth-dependent jobs. This will be addressed in the following chapter, where we will devise a model for NUMA-aware scheduling and try to predict interaction of concurrently running jobs.

5. A NUMA-Aware Processing Model

As could be seen in section 2.1 on page 6, only few threads on a single node can already utilize all available local memory bandwidth of this node. To see how queries respond to bandwith sharing, we execute them in parallel with a bandwith consuming job. The results of this experiment lead us to the question of how one could represent the interaction between demand of processor time and memory bandwidth with a simple realistic model. It should take moldability and malleability of jobs and corresponding changes in memory bandwidth into account, as well as the effects of memory bandwidth limitation.

After validating the model with single queries, we use it to argue why parallel execution of certain jobs can improve efficiency and try to prove this claim by experiments.

5.1. A New Processing Model

Figure 5.1 on the next page shows execution time of a single query job. The horizontal axis corresponds to an increasing number of jobs competing for memory bandwidth. Different colors indicate different numbers of CPU cores assigned to the running query job. Query run-time is plotted on the vertical axis and decreases when more cores are assigned, but increases when competing jobs consume memory bandwidth. Thus we will split job run-time into two fractions q and $(1 - q)$, both $\in (0, 1)$, representing time spent waiting for memory requests and computation time respectively.

First we will now discuss how to model bandwidth, and then describe how to take this into account when modeling job run-time.

Memory Bandwidth. Figure 5.2 on page 41 displays how much bandwidth is available when accessing local memory on a single node of the Xeon system. The bandwidth is scaled so that the first thread consumes one bandwidth-unit. As discussed in section 2.1 on page 6, only a few threads are needed to utilize a large portion of available bandwidth. Furthermore, bandwidth growth decreases as the number of competing cores increases. Thus, the maximum bandwidth b, which is less than the number of available CPUs, is not achieved when the number of running threads reaches b.

Since we will model bandwidth demand with fractions, we will use a continuous function to model available bandwidth. In the course of this thesis we will denominate bandwidth functions by $h(x)$, where x is the total non-negative bandwidth demand. We assume bandwidth functions to be non-negative, concave and bounded by some constant b for the domain $[0, \infty)$. Additionally we demand the derivative h' to be bounded by 1, because bandwidth is measured on a per-thread basis.

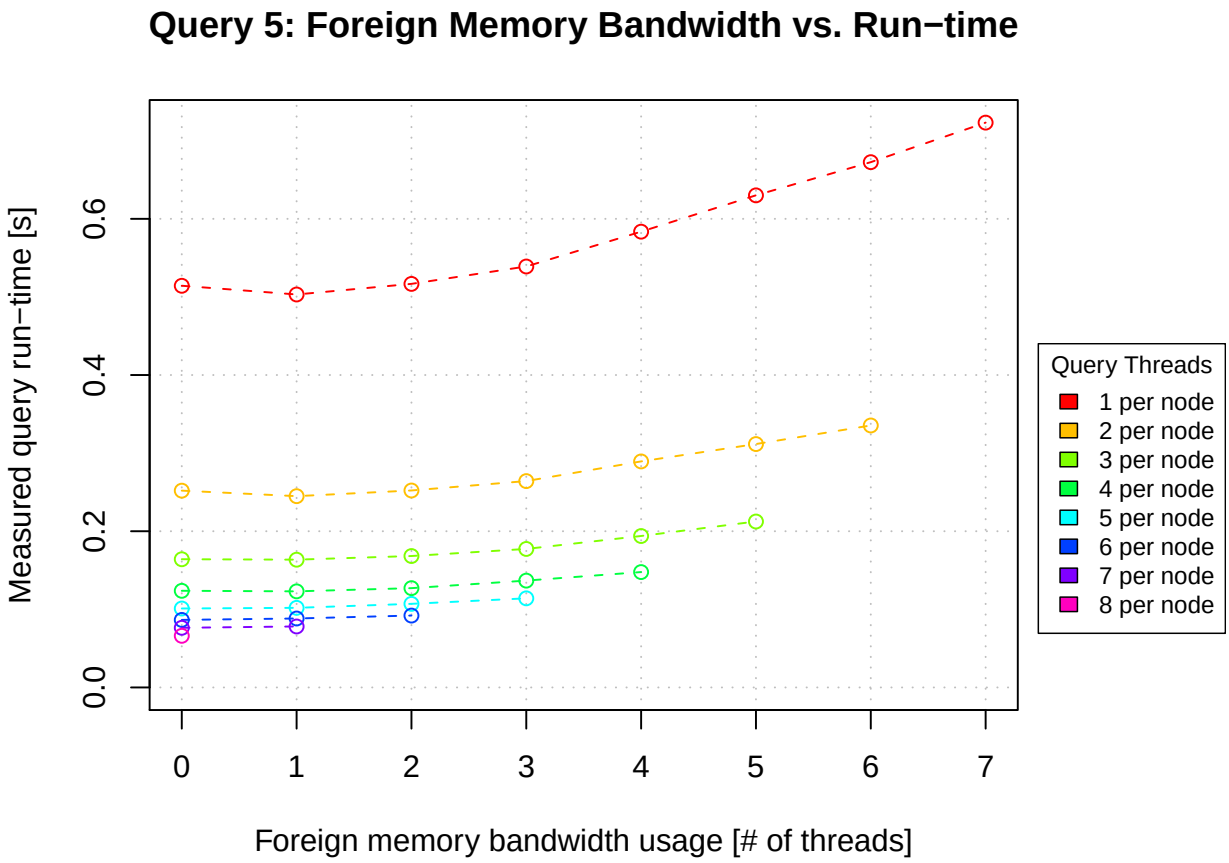


Figure 5.1.: Run-time of Query 5 on the Xeon system when increasing memory load or processor count.

A simple solution is to use a spline interpolation[1] $h_{\text{spline}}(x)$ of the bandwidth observed in section 2.1 on page 6. Another possibility would be to connect the measured values by a piecewise linear curve $h_{\text{poly}}(x)$. In our experiments, which are discussed in detail in section 5.3 on page 43, we found that the spline interpolation yielded good results. In figure 5.2 on the facing page it is plotted in blue.

When bandwidth demand exceeds the available bandwidth, time for memory access increases. This is modeled by introducing a slowdown-factor $f(x)$:

$$f(x) := \frac{h(x)}{x}$$

Because $0 \leq h(x) \leq x$ we have $f(x) \in [0, 1]$. We will use this factor to adjust the otherwise linear speedup of the memory-bound fraction of the program. Figuratively speaking, the available bandwidth is divided into equal fractions for each processor.

Job Execution Time. As said before, job execution is modeled as consisting of q fractions of memory access and $(1 - q)$ computation fractions. Processors are not shared while a job is

[1] Note that the spline curve violates the requirement $h'(x) < 1$ for some $x \in (0, 1)$.

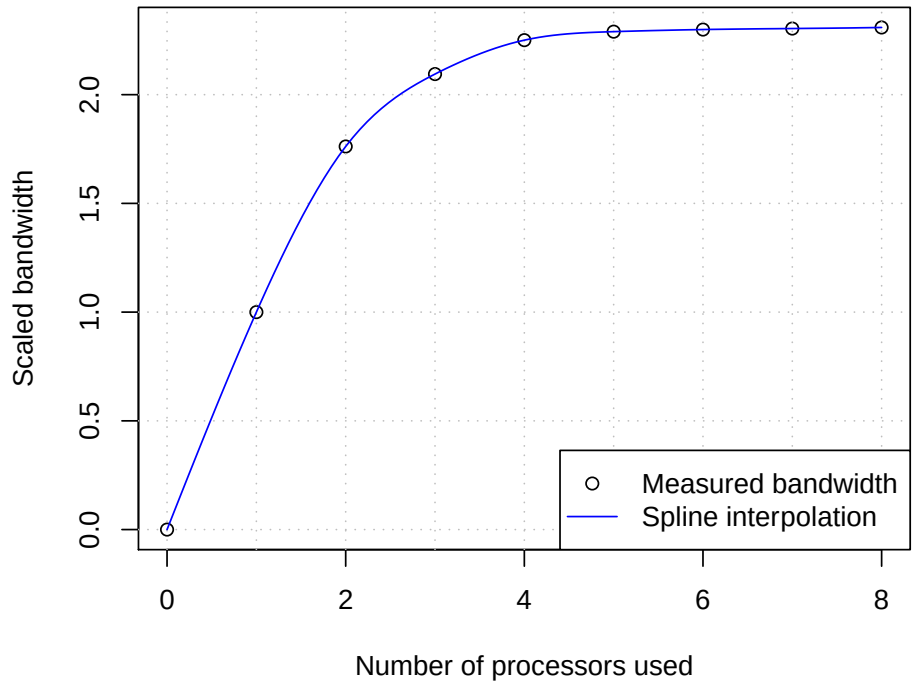


Figure 5.2.: Different models for memory bandwidth on the Xeon system.

running. Therefore, we assume computations to scale linearly with the number of processors used. Time for memory access on the other hand depends not only on available processors but also on bandwidth available.

Denoting the number of available processors by p, the memory bandwidth requested by other jobs as k and the running time on one processor as t, we can write

$$\text{time}(p, q, k, t) := \frac{t}{p} \cdot \left(\frac{q}{f(p \cdot q + k)} + (1 - q) \right)$$

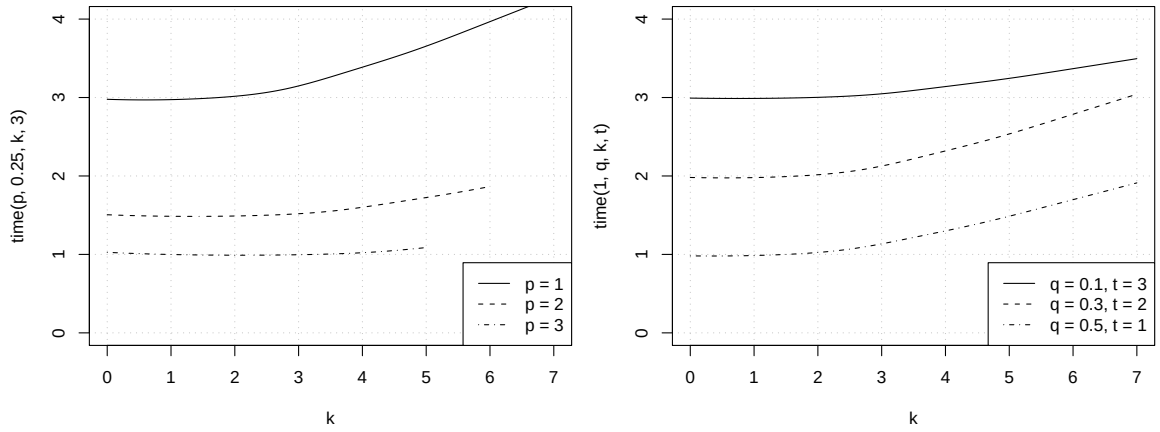
for the execution time of a job. Note that k as well as p in the malleable case might not be constant for the whole job duration.

Assume k changes to k' after t' units of time, but the job is not finished, i.e. $t' < \text{time}(p, q, k, t)$. In that case, only a ratio $r := \frac{t'}{\text{time}(p,q,k,t)}$ of the job has completed when k changes. The remaining time needed to complete the job is then $\text{time}(p, q, k', (1-r) \cdot t)$, resulting in a total execution time of $\text{time}(p, q, k, r \cdot t) + \text{time}(p, q, k', (1-r) \cdot t)$. Further changes of k' (or p in the malleable case) can be dealt with in a similar manner.

Examples of run-times modeled by this function are given in figure 5.3 on the next page.

5.2. Limitations of Our Model

Note that this model disregards many aspects which can influence job run-time. First of all, we assume the job to scale linearly in the number of processors—neither are there sequential program sections nor is there overhead for parallelization. The latter limitation can be removed



(a) Varying k for different numbers of processors p

(b) Varying k for different values of q and t

Figure 5.3.: time(p, q, k, t) for different parameters and bandwidth models.

by introducing an overhead function $o(p)$ for each job, depending on the number of processors used as model parameters. It is unclear, though, how sequential portions of the program should be modeled. Adding additional terms q_{seq} and $(1 - q_{\text{seq}})$ for sequential memory-dependent and CPU-dependent fractions would be possible. However, sequential and parallel sections of a program are normally not mixed arbitrarily. Thus, extending the model by *precedence constraints* and splitting jobs into sequential and parallel jobs would be a better approach.

Furthermore, we assume all memory access to be local, because in our case the NBB scheduler has exact knowledge of the data layout. In our experiments in section 2.1 we found the foreign memory bandwidth to be only about half as large as local memory access. Additionally, in some cases we observed *decreasing* foreign memory if bandwidth demand was increased. When taking foreign memory access into account, this loss of performance needs to be incorporated. Additionally, it has to be determined how local memory access on one node A interferes with foreign memory access from another node B to node A. Another issue with foreign memory access occurs when the communication graph is not complete. Not only will higher penalties occur for access to more distant nodes, but routing memory access from node A to B through a node C might also affect other foreign memory accesses routed through node C.

Another problem is hidden bandwidth usage. This can occur when the processor can prefetch required data from memory while performing independent calculations on already available data. The effect would be a small value of q for the given job, because its computation does not stall from memory access. Bandwidth usage of this job nevertheless does have an effect on other jobs running on the same node. To address this problem, the fraction q for memory access could be split up in two parts $q = o + q'$, o being the fraction that overlaps with computations, while the fraction q' does not overlap. The resulting new run-time function is $\text{time}(p, q', o, k, t) := \frac{t}{p} \cdot \left(\frac{q'}{f(p \cdot (q' + o) + k)} + (1 - q') \right)$. The use of system status counters that meter memory transfers in conjunction with the method currently used to determine q (which really determines q') could help to find good values for o.

One more unmodeled aspect which is local to a given node are cache effects. Many ques-

tions arise when trying to model caches: What access patterns does the job generate? How large is the cache footprint in relation to the number of processors used? When and how does data get displaced by another thread? The latter point can be overcome by techniques like page colouring [12]. Access patterns and to some extent displacement have been modeled in Manegold's thesis covering MonetDB [14]. But moldability and malleability have not been taken into account in these studies, and we are not aware of any that have.

Finally, SMT has not been taken into account in our model, as the Opteron processor we used does not have this feature. Consequently, this feature was disabled for the following experiments on the Xeon system as well.

The question remains whether these limitations have severe impact on the model quality. We will discuss this in the following section.

5.3. Model Validation

To validate our model, we try to predict how two queries running concurrently affect each other's run-time. Because each query will be executed on all available nodes, we slightly alter the semantics of the parameters used in the model from what we described before: The parameter p now stands for the number of cores used *per node*, and t is the time needed to execute the query when running on one *core per node* with full memory bandwidth. These alterations are valid, because due to the use of NBB memory access is local on each node, and we assume work to be evenly distributed across all nodes.

In order to predict execution times of queries running in parallel, we need to know the two parameters q and t for each query. We used the measurements displayed in figure 5.1 on page 40 to find these. To that end we use the nonlinear least squares method of GNU R[2] to fit the time function on the measured values by varying q and t for each query. We assumed the memory bandwidth consuming threads to have $q = 1$ in our model, so the parameter k of the time function used to find values for q and t is equal to the number of those threads running.

The memory bandwidth consuming thread uses the `movntq` Streaming SIMD Extensions (SSE) instruction to write data directly to memory, bypassing the caches to achieve maximum throughput and not pollute the caches.

Table 5.1 on the next page lists values for q and t as determined by minimizing the sum of squared errors. As stated in section 4.2 on page 35, queries 2, 3, 4, 11, 17, 19 and 20 were not considered. For queries 1 and 18 minimizing the squared error lead to *negative* values for q on both systems. We assume this is an effect of parallelization overhead and hidden bandwidth usage as described in the previous section 5.2. The same effect can be observed for queries 10, 13, 16 and 22 on the Opteron, but not on the Xeon system. Furthermore, the residual error of query 9 is large on both systems. This can be addressed to bad scalability of this query. Not modeling these effects is apparently a shortcoming of the devised model which needs to be taken care of. This is, however, beyond the scope of this thesis.

For the remaining queries the squared residual error is relatively small. Figure 5.4 on page 45 shows an example of how the fitted curves compare to the values measured in our experiments.

[2] http://www.r-project.org/

	Xeon			Opteron		
Query	q (%)	t	RS^2	q (%)	t	RS^2
1	-0.15	1.68	0.0026	-24.22	11.37	248.8587
5	20.24	0.50	0.0015	6.83	0.46	0.0043
6	0.86	0.34	0.0005	1.66	0.39	0.0017
7	22.66	0.87	0.0097	8.60	1.11	0.0034
8	18.64	0.19	0.0003	7.42	0.33	0.0027
9	20.38	2.52	1.2253	1.58	1.32	0.7577
10	16.09	0.37	0.0224	-0.08	0.42	0.0168
12	6.29	0.70	0.0471	5.64	0.68	0.0930
13	3.75	0.22	0.0173	-11.30	0.26	0.0754
14	23.71	0.51	0.0060	5.61	1.21	0.0206
15	39.37	0.93	0.0434	—	—	—
16	12.99	0.41	0.0164	-2.57	0.31	0.0373
18	-2.05	0.83	0.0018	-0.05	0.40	0.0011
21	16.55	0.66	0.0055	0.88	0.46	0.0134
22	13.39	0.11	0.0007	-8.12	0.08	0.0039

q and t are the job parameters determined by minimizing the sum of squared error. RS^2 is the residual sum of squared differences between model function and observed values. Query 15 was not executed on our Opteron system.

Table 5.1.: Job characteristics q and t of TPC-H queries as observed for scale factor 100.

After having determined values of q and t for each query we can now predict execution time of two concurrently running queries. Those experiments were only conducted on our Opteron machine.[3]

It is not surprising that run-time prediction for pairs that include queries 1 or 9 is erroneous. Looking at table 5.1, this could be expected as the residual errors for those queries are larger than for other queries. Run-time prediction for other pairs of queries is better, e.g. for queries 6 and 7 which are shown in figure 5.5.

In the following section we use the model to predict effects of scheduling jobs in a way that augments average bandwidth usage.

[3]When running preliminary tests on a Xeon machine similar to the one we described in this thesis, the results were similar though.

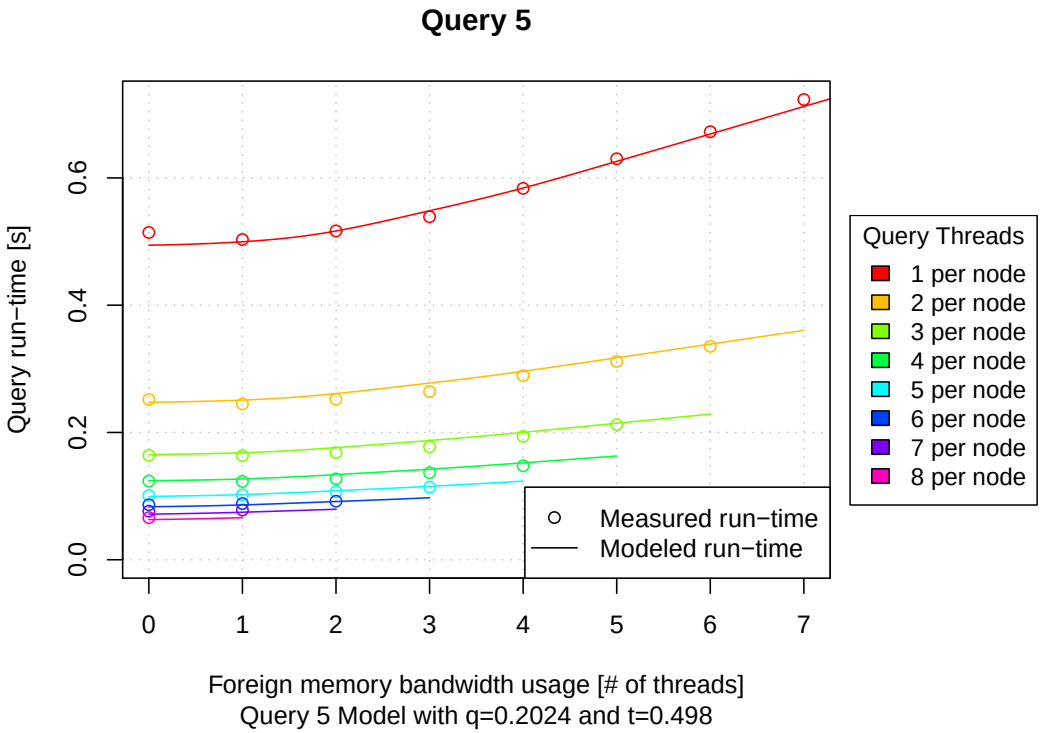
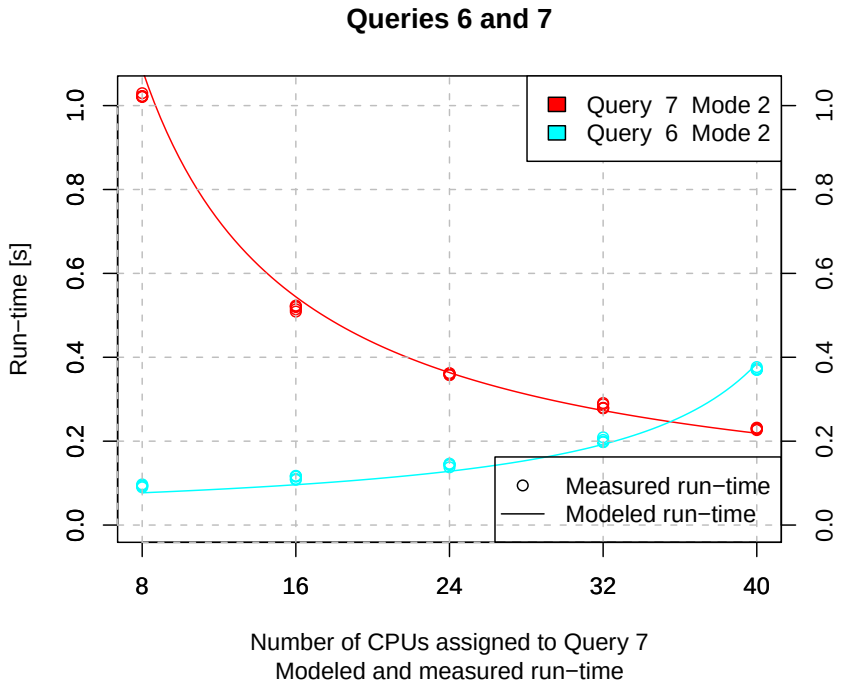


Figure 5.4.: Measured and modeled run-time of query 5 on the Xeon system.



CPUs not assigned to query 7 were used to execute query 6.

Figure 5.5.: Comparison of measured and modeled run-time of queries 6 and 7 running concurrently on our Opteron system.

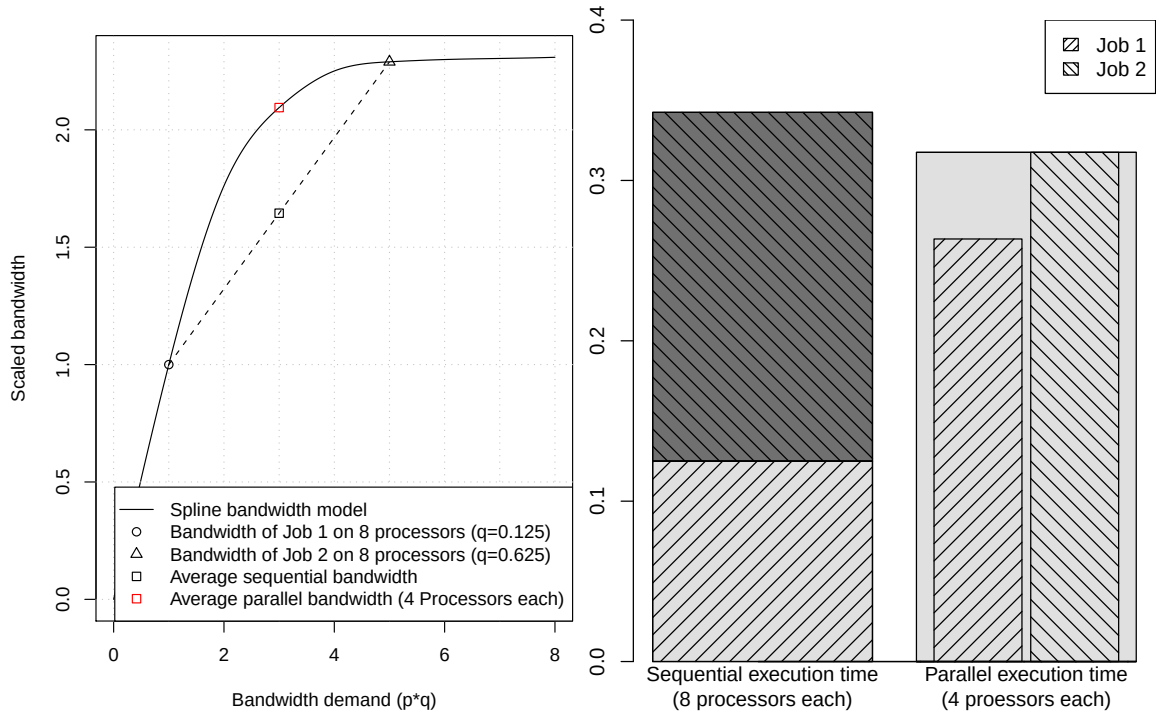
5.4. Gaining Efficiency by Parallel Execution

Our model suggests that sharing memory bandwidth and processors during parallel execution of two programs can be more efficient than executing the programs sequentially utilizing all processors. The reason is that one job alone may not be able to utilize the full memory bandwidth, while the other one does not gain more memory bandwidth than it would already have with just some of the processors it uses. These effects are shown in figure 5.6: (a) shows how average bandwidth usage can be increased by executing the jobs in parallel, and in (b) the resulting execution time is shown. For both jobs in the figure, t was set to 1. The effect is larger if the values for q lie further apart.

Figure 5.6 also shows why the bandwidth model predicts only little gain in efficiency if each jobs uses only little bandwidth when running alone (or both use all available bandwidth), i.e. the difference of q for both queries is small. This is because average bandwidth for sequential execution is then nearly the same as when running in parallel.

In subfigure 5.6b the problem already described previously occurs: After the shorter job has finished, available bandwidth for the longer running job changes. While it would in principle be possible to take these effects into account as described in section 5.1, we chose another approach.

Instead of running each job only once, we choose a minimal time $t_{\min}$ to run both jobs in parallel—job 1 on p_1 processors and job 2 on p_2 processors per node. Both jobs are then re-executed until the experiment has run for time $t_{\min}$.



(a) Increase in average bandwidth (b) Comparison of parallel and sequential execution time

Figure 5.6.: Example for efficiency gains due to parallel execution.

During the experiment we count how many times each job has been executed. The point in time at which the first of both jobs finishes after t_{min} units of time have passed is called t_{par}, depicted by the bold line in figure 5.7. This will be the actual duration used in our calculations. Let job 1 be the first job to finish execution after t_{min} units of time have passed. We denote the number of times job 1 was executed by k_1. The finishing-time of job 1's last execution is thus equal to t_{par}.

When job 1 has finished its last counted execution, job 2 may still be running. Let job 2 be in its k_2'-th execution phase at time t_{par}. However, we do not wish to count the whole runtime of this last execution of job 2, but only the fraction that was finished at time t_{par}.

To determine this fraction, we keep track of the k_2'-th starting- and finishing-time of job 2. They are denoted by t^2_{start} and t^2_{stop} respectively. We then can determine the fraction r of job 2 that was executed after t_{par} by

$$r := \frac{t^2_{\text{stop}} - t_{\text{par}}}{t^2_{\text{stop}} - t^2_{\text{start}}} \in [0, 1).$$

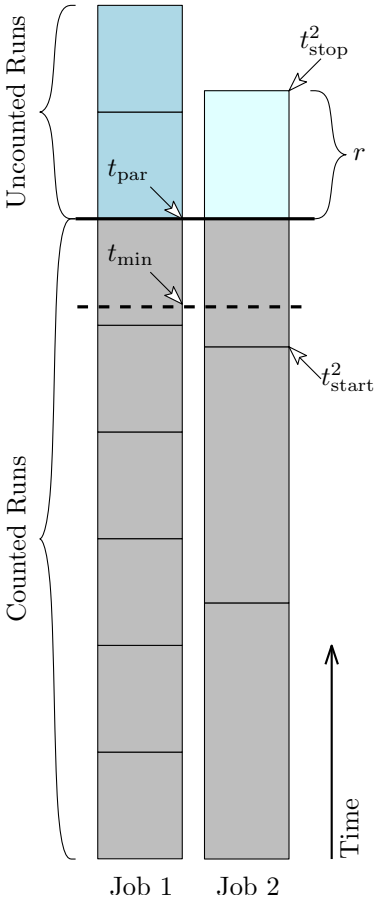


Figure 5.7.: Method and nomenclature for measuring efficiency.

To determine efficiency we use $k_2 := k_2' - r$ runs of job 2 instead of k_2' runs. In summary, job 2 has completed k_2 (fractional) runs in the time job 1 needed to complete k_1 (integer) runs.

Note that we have to ensure foreign memory bandwidth usage for job 2 does not change during the course of its last execution. To that end we continue to repeatedly execute job 1 as long as job 2 has not finished, without counting those runs.

We compare the time needed to run both jobs in parallel to the time needed for sequential execution of both jobs on all processors. Therefore, both jobs are executed k_1 and k_2' times respectively on all $p_1 + p_2$ processors. Total time taken to execute the jobs is measured as t_1 and t_2' accordingly. Now, the execution time t_2' of job 2 needs to be reduced with respect to k_2, i.e. $t_2 := t_2' \cdot (k_2'/k_2)$. The total time t_{seq} taken to execute both jobs on all available processors sequentially is then the sum of t_1 and t_2.

To perform the same amount of work, namely executing job i k_i times, in the sequential case all processors have been in use for t_{seq} units of time. In the parallel case the same amount of work was performed in t_{par} units of time utilizing the same number of processors. The resources (processor-time) required were $(p_1 + p_2) \cdot t_{\text{seq}}$ and $(p_1 + p_2) \cdot t_{\text{par}}$ respectively. Thus we can measure the efficiency gain ρ by $(1 - \rho) \cdot (p_1 + p_2) \cdot t_{\text{par}} = (p_1 + p_2) \cdot t_{\text{seq}}$.

Similarly, we can use the time function to predict the performance gain. We will denote the number of times each job is executed in the model by m_1 and m_2 respectively. With $p := p_1 + p_2$

and constants q_1, t_1 and q_2, t_2 that describe the characteristics of each job, we have

$$\begin{aligned} & (1-\rho_{\text{model}}) \cdot \big(p \cdot m_1 \cdot \text{time}(p, q_1, 0, t_1) + p \cdot m_2 \cdot \text{time}(p, q_2, 0, t_2)\big) \\ = & \underbrace{p_1 \cdot m_1 \cdot \text{time}(p_1, q_1, p_2 \cdot q_2, t_1)}_{L} + \underbrace{p_2 \cdot m_2 \cdot \text{time}(p_2, q_2, p_1 \cdot q_1, t_2)}_{R} \end{aligned}$$

Because we assume $L = R$ due to our experiment setup, we can write m_2 in terms of m_1, p_1, p_2, t_1, t_2, q_1 and q_2. By substituting m_2 and the time(p, q, k, t) function, we see that t_2 can be dropped. Furthermore, we can factor out $m_1 \cdot t_1$ on both sides of the equation. We can choose m_1 arbitrarily, and set $m_1 := \frac{1}{t_1}$. As expected ρ_{model} only depends on p_1, p_2, q_1 and q_2, i.e. the partitioning of processors used and the memory bandwidth demand of each job.

With

$$\pi_i := \frac{q_i}{f(p_1 \cdot q_1 + p_2 \cdot q_2)} + 1 - q_i \text{ and } \sigma_i := \frac{q_i}{f((p_1 + p_2) \cdot q_i)} + 1 - q_i$$

for parallel and sequential execution respectively, we can write the ratio as

$$\rho_{\text{model}} = 1 - \frac{2 \cdot \pi_1 \cdot \pi_2}{\sigma_1 \cdot \pi_2 + \pi_1 \cdot \sigma_2}$$

Now we can compare the performance gain predicted by our model to what was determined experimentally.

5.4.1. Experimental Results

The following table lists ρ_{model} and actually observed values for ρ on both systems. Not all pairs of queries could be tested due to issues with our test systems. In this experiment both queries were allotted half the number of available CPUs for parallel execution on each system.

Table 5.2.: Predicted and observed efficiency gain ρ_{model} and ρ.

		Xeon			Opteron		
Query 1	Query 2	ρ_{model} (%)	ρ (%)	$\rho - \rho_{\text{model}}$	ρ_{model} (%)	ρ (%)	$\rho - \rho_{\text{model}}$
1	1	0.00	2.11	2.11	0.00	49.44	49.44
5	1	0.80	3.23	2.44	2.42	37.19	34.77
5	5	0.00	9.98	9.98	0.00	0.14	0.14
6	1	0.00	3.70	3.70	1.43	35.97	34.54
6	5	0.76	5.30	4.54	0.13	14.16	14.03
6	6	0.00	4.20	4.20	0.00	4.67	4.67
7	1	1.17	-0.96	-2.13	2.85	34.42	31.58
7	5	0.04	0.36	0.32	0.02	-0.94	-0.96
7	6	1.13	0.30	-0.83	0.24	6.49	6.24
7	7	0.00	3.92	3.92	0.00	-2.63	-2.63

Table 5.2.: Predicted and observed efficiency gain ρ_{model} and ρ (continued).

		Xeon			Opteron		
Query 1	Query 2	ρ_{model} (%)	ρ (%)	$\rho - \rho_{\text{model}}$	ρ_{model} (%)	ρ (%)	$\rho - \rho_{\text{model}}$
8	1	0.60	2.97	2.37	2.55	39.69	37.13
8	5	0.01	3.06	3.04	0.00	-0.38	-0.38
8	6	0.57	4.01	3.44	0.16	19.77	19.61
8	7	0.09	-4.55	-4.65	0.01	-2.44	-2.45
8	8	0.00	10.06	10.06	0.00	7.15	7.15
9	1	0.81	39.29	38.48	1.42	50.19	48.77
9	5	0.00	40.23	40.23	0.13	34.36	34.23
9	6	0.77	39.66	38.89	0.00	35.10	35.10
9	7	0.03	37.80	37.77	0.25	30.35	30.10
9	8	0.02	40.76	40.74	0.17	39.20	39.04
9	9	0.00	56.95	56.95	0.00	43.23	43.23
10	1	0.38	11.54	11.17	1.18	40.54	39.36
10	5	0.08	6.24	6.16	0.22	21.05	20.83
10	6	0.35	11.88	11.53	0.01	16.69	16.68
10	7	0.22	4.98	4.76	0.36	17.24	16.88
10	8	0.03	10.32	10.29	0.26	23.77	23.51
10	9	0.08	37.72	37.63	0.01	37.37	37.36
10	10	—	—	—	0.00	12.23	12.23
12	1	0.03	-3.06	-3.09	2.16	34.72	32.56
12	5	0.51	-1.66	-2.17	0.01	11.44	11.43
12	6	0.02	-3.07	-3.09	0.07	6.25	6.18
12	7	0.83	-6.58	-7.40	0.05	1.87	1.82
12	8	0.36	-1.82	-2.18	0.02	13.48	13.47
12	9	0.53	36.98	36.45	0.08	33.91	33.83
12	10	—	—	—	0.15	14.20	14.06
12	12	—	—	—	0.00	-0.13	-0.13
13	1	0.01	14.95	14.94	0.23	49.97	49.74
13	5	0.63	10.72	10.08	1.14	35.07	33.93
13	6	0.00	19.51	19.51	0.51	30.52	30.02
13	7	0.97	14.12	13.15	1.45	18.99	17.54
13	8	0.46	14.34	13.88	1.24	30.14	28.91
13	9	0.65	44.72	44.07	0.50	47.00	46.50
13	10	—	—	—	0.36	27.44	27.08
14	1	1.38	-2.04	-3.42	2.15	34.00	31.85
14	5	0.08	-0.70	-0.78	0.01	17.66	17.65
14	6	1.33	-1.81	-3.14	0.07	10.60	10.52

Table 5.2.: Predicted and observed efficiency gain ρ_{model} and ρ (continued).

		Xeon			Opteron		
Query 1	Query 2	ρ_{model} (%)	ρ (%)	$\rho - \rho_{\text{model}}$	ρ_{model} (%)	ρ (%)	$\rho - \rho_{\text{model}}$
14	7	0.01	-6.31	-6.32	0.05	11.33	11.28
14	8	0.15	-4.74	-4.89	0.02	21.05	21.03
14	9	0.07	36.91	36.84	0.08	34.93	34.86
14	10	—	—	—	0.14	21.33	21.18
14	12	—	—	—	0.00	10.04	10.04
14	14	—	—	—	0.00	1.75	1.75
15	1	7.54	15.10	7.56	—	—	—
15	5	2.74	3.29	0.56	—	—	—
15	6	7.39	6.81	-0.57	—	—	—
15	7	2.07	-0.36	-2.43	—	—	—
15	8	3.20	-0.06	-3.26	—	—	—
16	1	0.20	16.27	16.07	0.88	42.66	41.79
16	5	0.20	16.24	16.04	0.38	14.97	14.59
16	6	0.18	15.37	15.19	0.07	22.98	22.91
16	7	0.40	13.72	13.32	0.56	13.93	13.37
16	8	0.11	15.86	15.75	0.44	24.31	23.87
16	9	—	—	—	0.07	38.59	38.52
16	10	—	—	—	0.02	22.23	22.20
16	12	—	—	—	0.28	20.04	19.76
16	14	—	—	—	0.28	20.67	20.39
16	16	—	—	—	0.00	16.12	16.12
18	1	0.00	2.50	2.50	1.18	36.68	35.49
18	5	0.86	5.20	4.34	0.22	13.37	13.16
18	6	0.00	4.15	4.15	0.01	11.61	11.60
18	7	1.26	1.84	0.59	0.36	8.00	7.64
18	8	0.66	5.75	5.10	0.26	19.86	19.60
18	9	—	—	—	0.01	36.41	36.40
18	10	—	—	—	0.00	11.31	11.31
18	12	—	—	—	0.15	10.96	10.81
18	14	—	—	—	0.14	21.03	20.88
18	16	—	—	—	0.02	23.26	23.24
18	18	—	—	—	0.00	11.96	11.96
21	1	0.41	9.42	9.01	—	—	—
21	5	0.06	10.40	10.33	—	—	—
21	6	0.39	10.35	9.96	—	—	—
21	7	0.19	5.72	5.53	—	—	—
21	8	0.02	8.22	8.20	—	—	—

Table 5.2.: Predicted and observed efficiency gain ρ_{model} and ρ (continued).

		Xeon			Opteron		
Query 1	Query 2	ρ_{model} (%)	ρ (%)	$\rho - \rho_{\text{model}}$	ρ_{model} (%)	ρ (%)	$\rho - \rho_{\text{model}}$
22	1	0.22	20.92	20.70	—	—	—
22	5	0.18	23.37	23.18	—	—	—
22	6	0.20	19.47	19.27	—	—	—
22	7	0.38	7.50	7.13	—	—	—
22	8	0.10	22.18	22.09	—	—	—

We can observe the predicted efficiency gains are often far below the observed values. This is especially the case when a query is executed concurrently with itself. In such cases, the model will always predict that concurrent execution is not more efficient than sequential execution, because average memory bandwidth is the same in both cases. Thus the gain in efficiency for parallel execution of two instances of the same query can be attributed to parallelization overhead of that query.

In conformance with the statement in the previous section 5.3, query 9 benefits most from this effect: it gains 57% efficiency when running in parallel.

As this effect is huge for other queries as well, the relatively small predicted gain due to increased average memory bandwith becomes negligible. We thus recommend using a job-specific speedup function to better represent overhead effects.

Effects on optimization goals. The model predicts, that parallel scheduling of jobs does never have negative effects on efficiency, because $\rho_{\text{model}} \geq 0$ holds. As described above, the effect is even greater due to decreasing parallelization overhead when executing two jobs in parallel. In other words, the amount of work performed per unit of processor-time does not decrease if jobs are scheduled in parallel rather than sequentially. This implies schedules for malleable jobs are not optimal if they do not utilize all processors at all times. Thus, the *makespan* for a given set of malleable jobs never increases when scheduled in parallel. Similarly, *throughput* cannot get worse by parallel scheduling in our model.

In contrast to that, *average response time* for a given set of malleable jobs can increase by parallel execution. An example is the set of jobs given in figure 5.6 on page 46. Job 1 finishes later when executed in parallel with job 2, which increases average response time for parallel execution. While job 2 finishes earlier when executed in parallel with job 1, this decrease does not compensate for the run-time increase of job 1. This is an effect of the limited memory bandwidth, which denies optimal resource (processor-time) utilization to job 2. The average response time of those two jobs is thus larger in the parallel case.

We conclude the thesis by recapitulating our results and pointing out possible future work on the topic.

6. Results

The results of this thesis are threefold. First, a NUMA-aware user-land scheduling library was developed. Based on memory benchmarks we found striving for node-local memory accesses would be beneficial for jobs with high bandwidth utilization. The operating system schedulers already offer means to enforce certain memory allocation and thread scheduling policies. Our library provides an easy-to-use abstraction to make use of these concepts. Aside from that it provides a framework to implement more sophisticated NUMA-aware scheduling algorithms for experimental evaluation.

Second, we evaluated the effects of our new approach to scheduling using a widely accepted database benchmark. The observed performance improvement over NUMA-agnostic scheduling was up to 67%. Additionally, we noticed a decrease in the meanderings of job run-times that resulted from non-local memory access due to suboptimal scheduling decisions.

The third and final result of our work is a new model for job scheduling. We characterize jobs by two key figures: overall run-time and memory access ratio. The model was shown to reflect the effects of scheduling bandwidth-demanding job in parallel. Furthermore, we demonstrated that the model allows us to augment efficiency by scheduling certain jobs to run in parallel rather than sequentially. This in turn leads to higher query throughput and/or smaller makespan when scheduling certain jobs in parallel. However, this effect is dominated by effects of parallelization overhead in practice.

We come to the conclusion that NUMA-aware scheduling can be beneficial for memory bandwidth demanding workloads as they appear in main-memory based database systems.

6.1. Future Work

This thesis opens many possibilities for further work. While the developed NBB library showed to provide good performance, there is still potential for optimization. This includes replacing the used queues by lock-free variants and dynamic adaption of the grain-size used for parallel iteration and reduction. Also, though it was not required for the algorithms used in our experiments, a parallel join operation supporting *non-commutative* joins could be implemented. The parallel execution of TPC-H Queries as described in the TPC-H Stream Test could be improved by implementing a custom scheduling policy.

Regarding the processing model it is of interest how the model can be refined to better predict job run-time and effects of concurrent job execution as indicated in section 5.4.1. Namely, we expect modeling hidden bandwidth usage and job-specific speedup functions to improve quality of run-time prediction. It then would be necessary to back those findings by more types of parallel programs.

So far, the framework was only used to schedule moldable jobs. A custom scheduler implemented e.g. for the TPC-H Stream Test could be used to determine, whether the model yields good predictions for malleable scheduling as well. Furthermore, optimization goals different from makespan could be targeted.

Another aspect is job scheduling in our new model. Calculating optimal schedules or solving the online scheduling problem is an open problem. When solving the online problem the additional problem of finding good parameter estimates for the jobs to be scheduled comes up.

Appendix A.

Building NBB

A.1. Prerequisites

Most prerequisites for building NBB can be obtained by installing the appropriate programs or development packages of your favourite Linux distribution. For Ubuntu, the following command should provide all packages needed in addition to the default setup:

```
$ sudo apt-get install numactl libnuma1 libnuma-dev scons doxygen
    git-core
```

The SCons version used was v1.2.0. Intel's TBB[1] library is used for comparison in some of the test scripts, but is not a dependency of the library itself. The "tbb30 20100406oss" release is known to work.

To be able to compile all template code, we used the GNU C++ compiler from version 4.5.1 of the GNU Compiler Collection[2]. It was not available as package for our systems at the time of this writing. Please refer to the according documentation on how to build a custom compiler, if this is still the case.

Currently, the latest release of boost is version 1.46.1, which suffers from a bug[3] that can cause deadlocks when NBB is shutting down during program termination. If this causes problems, please patch boost as described in the referenced boost bug tracker entry before compiling boost, or use a to-be-released version of boost including the fix. Other third-party libraries like libNUMA can be found in this thesis' software repository (see **THIRD_PARTY_PATH** described below).

A.2. Directory Structure

The implementation of NBB is spread across several directories. Due to the SCons build process, the script to build the library is contained in the `lib` directory. The `test` directory contains some unit- and functionality tests written using the boost test framework.

All C++ header-library files are located beneath the `include/nbb` directory. Each implemented component is stored in a separate directory, split up into declaration and implementation header files where necessary. Similar to the boost library design, convenience header files

[1] http://threadingbuildingblocks.org/
[2] http://gcc.gnu.org/
[3] https://svn.boost.org/trac/boost/ticket/4978

```
nbb/
├── SConstruct ..... Main build script
├── lib/ ..... Build script for the library
├── src/ ..... Library initialization code
├── test/ ..... Unit- and Functionality tests
├── include/ ..... Base for C++ header-library files
│   └── nbb/
│       ├── range/ ..... Range class implementation
│       ├── mm/ ..... Internal memory management for job objects
│       ├── partitioner/ ..... Partitioner class implementation
│       ├── parallel_for/ ..... Implementation of parallel iteration
│       ├── parallel_reduce/ ..... Implementation of parallel reduction
│       ├── array/ ..... Basic array container class
│       ├── sched/ ..... Scheduer and policy classes
│       └── alloc/ ..... NUMA-Aware memory allocation
```

NBB Directory Structure

which include both declaration and implementation are provided. They reside in `include/nbb` to hide these details from the programmer.

A.3. Build Process

As described before, SCons is used to build NBB. When SCons is invoked it looks for a build script named `SConstruct` written in Python. In the course of writing this thesis several other build scripts were used for side projects. All share a common `local_build_config.py` where several system-specific variables can be set:

BOOST_PATH Path to boost installation, e.g. `/usr/local`.

TBB_PATH Base path to TBB, e.g. `/opt/intel/tbb/tbb30_20100406oss/` if one uses the Open-Source edition (OSS) of TBB.

TBB_RELEASE, TBB_DEBUG When building TBB OSS edition, the libraries will be put in separate directories. The directory name is system-dependent and usually looks like "`linux_intel64_gcc_cc4.5.1_libc2.11.1_kernel2.6.32_release`".

NBB_PATH Path to NBB base directory, e.g. `~/Diplomarbeit/nbb`. This is not required for the NBB build process itself, but to build the benchmark programs that use the NBB library.

NUMA_TOPOLOGY_PATH Provides information about hardware topology. Provided in `Diplomarbeit`

THIRD_PARTY_PATH Path to third party libraries collected in `Diplomarbeit/external`. These include libnuma and libNUMA for information about the system topology and Ro-

man Dementiev's Intel Performance Counter monitoring library[4] to collect information from hardware status registers of Intel processors.

DEES_PATH Path to Jonathan Dees' TPC-H Query implementation, e.g. `~/repositories/dees`. Used when compiling the parallel query benchmark evaluation program.

CXX Set this variable to use a C++ compiler different from the system's default.

opt Set compiler optimization level. Defaults to 3.

debug Controls mode of compilation. If set to zero (0), SCons will build NBB with optimizations turned on (by default, `-O3` is passed to the compiler). Otherwise, NBB will be compiled without optimizations and enabled debugging symbols, i.e. `-O0 -g` will be passed to the compiler.
This variable can also contain a (Python) list of module names to be debugged. In this case additionally various trace messages will be printed from the debugged module. Tracing can be enabled for the following modules: alloc, array, jobs, parallel_for, parallel_reduce, policy, scheduler, range and worker. When tracing is enabled for the job module, parent jobs will also maintain pointers to all spawned child jobs which may help debugging job dependencies.

assertions Boolean to control setting of the `NDEBUG` preprocessor macro. When true, assertions will be enabled (regardless of the **debug** variable described above). Assertions should be switched off for performance evaluation.

The file `local_build_config.py.template` shows some examples of how these variables can be set to control the build. All parameters can also be set by passing the variable definition as a SCons command line argument. Command line variables will override settings from `local_build_config.py`. For example, to build an optimized version with optimization level 2 and without assertions using a custom C++-Compiler, use the following command:

```
$ scons opt=2 CXX=/opt/some-cc/bin/c++ debug=0 assertions=false
```

It is common to split SConstruct files into several SConscript-files for separate parts of the project and load them in the main SConstruct file. For NBB, two SConscript files reside in `lib/` and `test/`. The former is responsible for building the library, the latter for building the test binaries.

Calling SCons without any target name will build the library and test binaries. The following targets are available:

nbb_shared Build shared library `libnnb.so`.

tests Build test suite for NBB.

run_tests Run all tests from suite.

[4] http://software.intel.com/en-us/articles/intel-performance-counter-monitor/

doc Build HTML documentation using Doxygen. Doxygen configuration suitable for NBB is provided in the `Doxyfile` file.

To run all tests and build the documentation at the same time using 42 parallel SCons jobs, run:

```
$ scons -j42 doc run_tests
```

Documentation will end up in the `doc/` directory. Everything else can be found in `build/optimize` or `build/debug`, depending on the value of the **debug** build variable.

Bibliography

[1] N. S. Arora, R. D. Blumofe, and C. G. Plaxton. Thread Scheduling for Multiprogrammed Multiprocessors. *Theory of Computing Systems*, 144(June 1998):115–144, 2001.

[2] J. Blazewicz. SCHEDULING SUBJECT TO RESOURCE CONSTRAINTS: CLASSIFICATION AND COMPLEXITY. *Complexity*, 5, 1983.

[3] T. Brecht. An experimental evaluation of processor pool-based scheduling for shared-memory NUMA multiprocessors. *Job Scheduling Strategies for Parallel Processing*, 1291:139–165, 1997.

[4] W. Cirne and F. Berman. Using Moldability to Improve the Performance of Supercomputer Jobs. *Journal of Parallel and Distributed Computing*, 62(10):1571–1601, Oct. 2002.

[5] M. Drozdowski. Scheduling multiprocessor tasks — An overview. *European Journal of Operational Research*, 94(2):215–230, Oct. 1996.

[6] K. Fatahalian, T. Knight, M. Houston, M. Erez, D. Horn, L. Leem, J. Park, M. Ren, A. Aiken, W. Dally, and P. Hanrahan. Sequoia: Programming the Memory Hierarchy. *ACM/IEEE SC 2006 Conference (SC'06)*, (November):4–4, Nov. 2006.

[7] D. Feitelson, L. Rudolph, U. Schwiegelshohn, K. Sevcik, and P. Wong. Theory and practice in parallel job scheduling. In *Job Scheduling Strategies for Parallel Processing*, pages 1–34. Springer, 1997.

[8] E. Frachtenberg and D. Feitelson. Pitfalls in parallel job scheduling evaluation. In *Job Scheduling Strategies for Parallel Processing*, pages 257–282. Springer, 2005.

[9] M. N. Garofalakis. Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources. *Processing*, 1997.

[10] ISO. ISO/IEC 14882: Programming languages - C++, 1998.

[11] T. Koita, T. Katayama, K. Saisho, and A. Fukuda. Memory Conscious Scheduling for Cluster-based NUMA Multiprocessors. *The Journal of Supercomputing*, 16(3):217–235, 2000.

[12] R. Lee, X. Ding, F. Chen, Q. Lu, and X. Zhang. MCC-DB: minimizing cache conflicts in multi-core processors for databases. *Proceedings of the VLDB Endowment*, 2(1):373–384, 2009.

[13] J. Leung, L. Kelly, and J. H. Anderson. Handbook of Scheduling: Algorithms, Models, and Performance Analysis. Apr. 2004.

[14] S. Manegold. *Understanding, modeling, and improving main-memory database performance*. PhD thesis, Universiteit van Amsterdam (UvA), 2002.

[15] J. Meng, J. W. Sheaffer, and K. Skadron. Exploiting inter-thread temporal locality for chip multithreading. *2010 IEEE International Symposium on Parallel & Distributed Processing (IPDPS)*, (c):1–12, Apr. 2010.

[16] J. Philbin, J. Edler, O. J. Anshus, C. C. Douglas, and K. Li. Thread scheduling for cache locality. *ACM SIGOPS Operating Systems Review*, 30(5):60–71, Dec. 1996.

[17] (Transaction Processing Performance Council). TPC Benchmark H (Decision Support). *Standard Specification, Revision 2.11.0*, pages 1–138, 2010.

[18] J. I. Turek, L. C. U. Fleischer, U. U. O. D. Schwiegelshohn, J. L. I. Wolf, W. U. O. W.-M. Ludwig, P. I. Tiware, and J. I. I. O. T. Glasgow. Scheduling parallel tasks to minimize average response time. In *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 112–121. Society for Industrial and Applied Mathematics, 1994.

List of Tables

List of Figures

List of Listings

Glossary

NBB NUMA Building Blocks, the NUMA-aware parallelization library described in this thesis. 12, 14, 27, 32–36, 42, 43, 53, 55–58

NUMA Non-Uniform Memory Access. 1, 3, 7, 11, 12

PCMCIA People Cannot Memorize Computer Industry Acronyms.

SMP Simultaneous Multiprocessing. 1, 4–6

SMT Simultaneous Multithreading, called "Hyperthreading" by Intel. 1, 4, 5, 43

SQL a standardized language for relational database systems, usually referred to as "Structured Query Language". 3, 31

SSE Streaming SIMD Extensions, a **S**ingle **I**nstruction **M**ultiple **D**ata extension to the IA-32 instruction set. 43

TBB Threading Building Blocks, a parallelization library by Intel. 11, 12, 15, 31–36, 55, 56

TPC Transaction Processing Council. 3, 31

TPC-H TPC Benchmark Specification H. 3, 12, 31, 32, 36, 53, 57