TDT4506 - Datateknikk, fordypningsemne

Fall 2012

Database operations on multi-core processors

Author:
Stian Liknes

Supervisor:
Kjetil Nørvåg

Abstract

This paper give a basic overview of current research for database operations in the multi-core, and goes on to describe a super-awesome multi-core algorithm for top-k-join or something. The algorithm achieves X speedup compared to traditional methods.

Contents

Chapter 1

Introduction

Algorithms used in databases have traditionally focused on external factors like I/O-operations. Disk access is an order of magnitude slower than memory access and minimizing I/O-access is a necessity working on data sets exceeding main memory capacity. However, a rapid increase in main memory capacity are making memory based algorithms more relevant.

There are two main areas for memory based algorithms in database systems: main memory database management systems (MMDB) and disk resident database management systems (DRDB) with very large caches. MMDBs keep all data in memory and thereby completely avoids the I/O bottleneck, therefore traditional algorithms can not be regarded as optimal without close examination. Main memory bandwidth and compute capacity are becoming dominant factors in algorithm performance. In addition, DRDBs with large caches can complete many operations without intermediate I/O access. This leads to a requirement for algorithms that use memory bandwidth efficiently and are capable of optimal cache patterns without making compromises to minimize I/O access.

For a long time, processor manufacturers increased compute power by increasing the operating frequency and placing transistors closer together. Unfortunately, this is no longer an option, increasing operating frequency further shows diminishing gains in performance compared to the power requirements. The so-called power wall has been reached, and processor manufacturers have been forced to find other ways to utilize chip-transistors. However, Moore's law is still valid, number of transistors per chip are continue to increase. To utilize these resources manufacturers have started to increase number of cores per processor, which is an efficient way of increasing compute capacity without exponentially increasing power consumption.

Another trend is that processors are providing greater capacity for data parallelism by in the form of single instruction multiple data (SIMD) processing. SIMD allows one operation to be performed for multiple inputs without additional CPU cycles, i.e. multiply four values at the price of one. For instance both Intel and AMD have are supporting streaming SIMD extensions 4 (SSE4) providing many opportunities for data parallelism. Currently, general purpose processors have limited support, SSE4 supports signed multiplication of no more than four 32-bit integers in one operation. However, a speedup of four in an inner loop are definitely worth pursuing. Combining task parallelism and data parallelism, one can achieve significant performance gains in suited algorithms.

In [19], Stonebraker compares the three primary parallel architectures: shared memory (SM), shared disk (SM), and shared nothing (SN). In SM systems, all processors (or cores) share a common central memory. Each processor in a SM system has private memory, while all processors share a collection of one or more external disks. For SN systems, nothing is shared, this is the case of distributed systems and have been the primary focus in database management systems (DMBS). Stonebraker argued that SM systems did not scale to a large number of processors, hence they were less interesting than the other systems. He also observed that SD systems does not excell in any area compared to the other two, and concluded that SN systems is the primary target for database management systems. Not only did SN show the best characteristics for scaling, it also matched the current marketplace interest, distributed database management systems. At the time (1985), this was the obvious contender for further research. However, with the recent trends regarding

processor- and memory progress, SM systems are becoming more prominent. Multi-core processors are highly-popular SM systems, and developers must resort to SM programming to utilize increased compute capacity.

The increased interest in SM programming has led to an number of frameworks to help programmers gradually parallelize existing algorithms and to develop completely new methods. These frameworks address some of the issues mentioned by Stonebraker in [19], for instance, frameworks provide practical solutions for concurrency control and management of hot spots. OpenMP is a popular choice for incrementally adding parallelism to an algorithm, this framework allows the developer to tag parallel regions as "parallel" using directives. Critical sections can be tagged as "critical" to provide a simple concurrency control. There are high-level and low-level frameworks. High-level frameworks, like OpenMP are great to parallelize loops and typical bottlenecks, but in some cases developers need to resort to lower-level programming to get more fine-grained synchronization and better utilization of the resources available.

Memory based databases are also obtaining increased interest. Commercial products like IBM solidDB and Oracle TimesTen are examples of high-performance relational MMDBs in use today. By managing data in memory, and optimizing data structures and access algorithms accordingly, database operations execute with maximum efficiency, achieving dramatic gains in responsiveness and throughput [14].

Much of the research on multi-core algorithms have focused on the join operation. Hash join for shared memory architectures is evaluated in [4] and sort-merge join is explored in [1]. A common theme is that synchronization costs are a big factor in multi-core performance. Larson et al. explores concurrency control mechanisms for MMDBs in [11], they argue that concurrency control methods used today do not scale to the high transaction rates achievable, and suggests methods based on multiversioning.

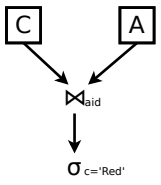
Even though shared-memory systems only recently became popular, parallel database management systems have been around for a long time in the form of shared nothing. Researchers have developed techniques to achieve optimal load balancing, minimize synchronization and to solve other issues related to parallel algorithms. To avoid re-inventing the wheel, traditional techniques should be considered when developing algorithms for new architectures.

This report focus on database operations optimized for multi-core systems. First general database operations are explained using the relational operator terminology. Second traditional, disk-based methods, such as nested-loops, sorting, and partitioning are explored. Third, the multi-core architecture is introduced, including memory and cache systems, multithreading, and common optimization techniques. Fourth, parallel, memory based database algorithms are presented, and a multi-core algorithm (HIPTA) is developed based on the threshold algorithm for top-k search explained in Section 3.6.1, optimization techniques presented in Section 4.6, and traditional partitioning schemes used in parallel database systems. Finally HIPTA is implemented and evaluated in Chapter 6.

Chapter 2

Database operations

Database operations are commonly described using relational algebra, a set of well-defined operations on relations (or tables). The relational model was designed to enhance human productivity working with large amounts of data [6]. Relational algebra makes no assumptions about the underlying algorithms or data structures, but provide clear definitions of the expected behavior. The advantage is that it allows database developers to implement efficient algorithms tailored for different systems and use cases without changing the terminology or user interface. Relational operators can be combined to execute complex queries, for example, one can select all cars that are red and located in Trondheim from two relations using select and join. Database queries can be described using a mathematical representation, graphically (tree of operations), or using structured query language (SQL), see Figure 2.1a - 2.1c.



(a) Tree

$A = address$
$C = car$
$c = color$
$R = \sigma_{c='Red'}(C \bowtie_{aid} A)$

(b) Mathematical

```
SELECT *
FROM car, address
WHERE car.aid = address.aid
AND color = 'Red'
```

(c) SQL

Figure 2.1: Query representation

This chapter describes common database operations, and some more exotic operations that have appeared in recent years.

2.1 Selection and projection

Selection σ is used to select a number of rows from a relation by specifying a condition. E.g. select every row that has an id below 10. The condition is typically a boolean combination of terms that have the form *attribute* op *constant* or *attribute* op *attribute*, where op can be any of the following comparison operators: $<, \leq, =, \neq, or >$. [17]

```
SELECT id,name
FROM car
WHERE id < 10
```

Listing 2.1: Selection and projection example

The projection operator π extract columns from a relation, e.g. select the *id* of each car. To optimize a query it is often reasonable to do projection before selection to minimize data movement, only the relevant attributes should be kept. Projection will also remove all duplicate entries, so that every row is unique after completion.

2.2 Join

The join operation is one of the most useful operations in relational algebra and the most common used way to combine information from two or more relations [17]. Join can be defined as the cross-product of two (or more) relations followed by selection and projection. However, the cross-product is very big compared to the join result, so the operator is rarely (never) implemented this way in practice. Joins have received a lot of attention, and there are several variants of the operation.

Conditional join is the most common. This variant accepts a join condition c and a pair of relation instances as arguments, it returns a relation instance.

```
SELECT *
FROM car,address
WHERE car.aid = address.id
```

Listing 2.2: Join example

Equijoin is a special case of the join operation where the join condition consists of only equalities, connected by $\wedge$. Listing 2.2 is an example of equijoin, here the result of a conditional join would include both car.aid and address.aid, two identical values, equijoin use projection to remove such redundancies.

Natural join is a further special case of the join operation where the equalities are specified on all fields having the same name in joined relations. The result are guaranteed not to have two fields with the same name.

2.3 Top-k

In many domains, end-users are interested in the most important (top-k) query answers, from a potentially huge answer space. Answers can be ranked by multiple criteria, like price, location, and size, each criterion is weighted.

To efficiently produce answers to top-k queries, database management systems (DBMSs) can use specialized top-k operators like top-k select and top-k join. An overview of is found in [8].

2.3.1 Select

Top-k select is a special case of select where the k best matches for a given scoring function is returned. The scoring function is typically defined as a weighted sum over a set of attributes, i.e. $f(a, b, c) = 0.5 * a + 0.4 * b + 0.1 * c$. Top-k select requires that all attributes are contained in one relation.

```
SELECT *
FROM posts
WHERE author='Pete'
ORDER BY 0.5*comments + 0.4*views + 0.1*likes
LIMIT 5
```

Listing 2.3: Top-k select example

The query in listing 2.3 selects the 5 best posts authored by Pete, posts are ranked by comments, views, and likes, respectively.

2.3.2 Join

Top-k join is a generalization of top-k select where attributes can be selected from any number of relations. The answer to a top-k join query is an ordered set of join results according to some provided function that combines the orders on each input [7]. Compared to the normal join operator, top-k can be executed much faster if an appropriate algorithm is used. The algorithm only needs to return k results and do not necessarily have to join every row in the relations (as opposed to traditional join).

```
SELECT * FROM R_1, ..., R_m
WHERE join_condition(R_1, ..., R_m)
ORDER BY f(R_1, ..., R_m)
LIMIT k
```

Listing 2.4: Top-k join example

Chapter 3

Disk-based algorithms

In disk-based databases I/O is the bottleneck, therefore volume of data moved from disk to memory (and back) is a good indication of algorithm efficiency. Disk-based algorithms are designed to work with large data volumes in a "slow" medium (disk) by temporarily storing small parts in a vastly faster medium (main memory) and writing them back when processing is done. In some cases data has to be written back to a temporary file on disk, further increasing I/O-traffic.

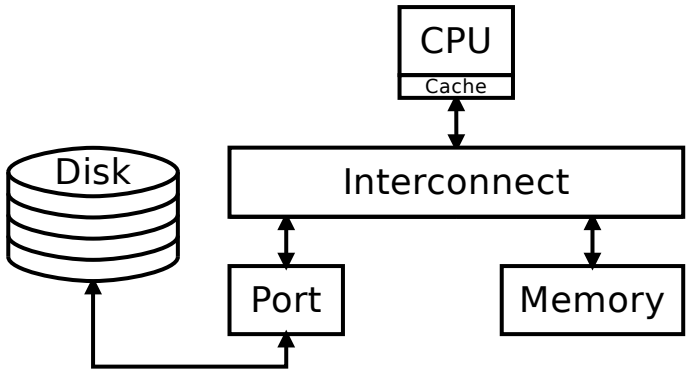


Figure 3.1: Disk-based architecture

Figure 3.1 shows a simplified view of the target architecture. The processor is connected to disk and memory by an interconnect bus. Disk has an extra layer before the interconnect, this can be a PCI or similar, an external bus. Communicating with main memory is a relatively fast operation, data is transferred over the internal bus (interconnect), while communicating with disk needs to go by an external bus, in addition to waiting for a slower medium (typically a magnetic hard disk drive requiring mechanical motion to access data).

There are three main groups of disk-based methods: nested loop, sorting, and partitioning. In addition indexes, filters, histograms and other methods are used to improve performance.

3.1 Nested loop

Nested loop (NL) is the most straight forward method, it is very efficient for small to moderate operand volumes [6]. This method exploit main memory to save CPU and I/O work by temporarily storing records in a workspace (WS) while processing. Using a big workspace will give improved performance. Ideally WS is big enough to hold the entire input relation A. Method is outlined in Algorithm 1, upper case letters are relations while lower case are records. Note that the smallest input relation, A, is used in outer loop to minimize I/O-traffic.

Algorithm 1 Nested loop

Require: A = smallest input relation, B = bigger (or equally sized) input relation, R = output relation
Ensure: R contains result of algebra operation
  while A has unread records **do**
    read records from A and store in WS until WS is full or A is read
    for all $b \in B$ **do**
      if $b \in A$ **then**
        complete algebra operation for record b and save result to R

The workspace structure is a critical part of this method, and it should be optimized for fast access to records when complete key is known. Bratbergsengen use a combination of hashing and binary trees in [6]. Hashing partitions keys into smaller sets and binary trees give a reasonable search length, even if hash partitioning fails. Each node consists of four values: left subtree pointer, right subtree pointer, pointer to record and signature. The signature can be used to rule out records that does not match a search, when a potential match is found record key must be read. Using 4 words (16B) per node provides an additional advantage, cache-friendliness for architectures where cache-lines are multiples of 4 words (i.e. the common architectures). Figure 3.2 illustrates the idea, hash table is depicted at the top, actual tuples at the bottom, and binary trees between.

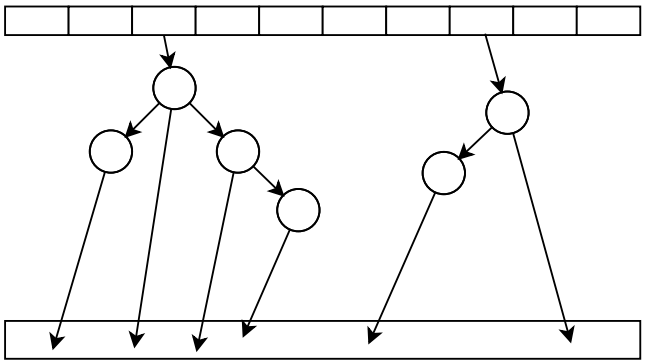


Figure 3.2: Nested loops workspace

3.2 Sorting

Sorting method is used to exploit properties of sorted datasets. In the general case, all input operands are sorted by operation key. If input are sorted already this can give a performance gain, but for single operations where no sorting is required, other methods are likely to be more efficient. For instance, checking for duplicates is a trivial task if sorted input is available. Latest unique record can be used to discard all subsequent duplicates without any additional runs.

3.3 Partitioning

Partitioning is used to prepare input data for nested loop method by splitting smallest operand into workspace-sized partitions. The goal is to avoid repeated operand passes, and thereby reduce I/O. An effective way of partitioning is to use a hash formula that uniformly distributes data among partitions. Partitions are stored on disk before nested loop method is executed.

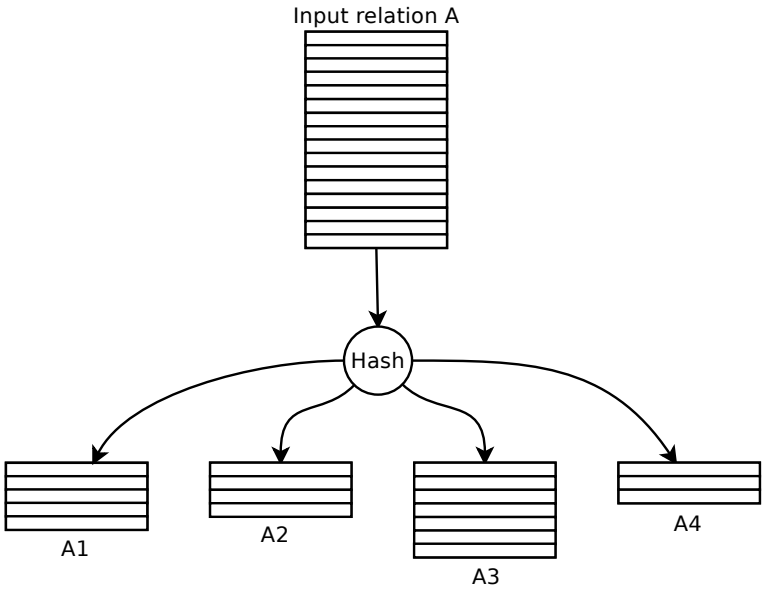


Figure 3.3: Partitioning

Figure 3.3 illustrates the partitioning of an input relation A. It is split into four partitions, A1, . . . , A4 using a hash function, each partition contains a similar amount of records. These partitions are in turn fed to the nested loop method, and results are stored in one relation. The figure also illustrates that partitions may have different sizes, i.e. some skew are observed. This can be caused by a hash function that are unable to create a uniform distribution of values on the current input set. To ensure that most groups fit into WS without a perfectly uniform distribution, nominal 1roup size is normally chosen slightly smaller than WS size (5-10%).

3.4 Selection and projection

Selection can be implemented using nested loop by inspecting all records, and writing matches to the result. If sorted input is available on the relevant key, only a subset of the input relation needs to be processed. Problem is reduced to finding the first match and reading records until there are no more matches. Partitioning can be utilized for parallel execution by distributing input relation to different processes (or nodes).

Projection is more involved due to the requirement of only returning unique records. There are two principal ways to solve this problem using nested loop:

1. Store records in result set on first discovery, and check result for duplicate values every time a new records is processed
2. Fill WS with records from input relation, and subsequently remove duplicates from WS by reading the entire relation before moving all records in WS to result set, see Algorithm 2

The first method will produce results earlier, but will stall when WS is full. Reason being that we need to compare each record to all records in result set, and therefore need to read result set back from disk. It will also run slower as the result set increases due to an increased number of records that have to be compared. Second method has opposite properties, it needs to fill WS and read entire relation before any records are returned. On subsequent runs it reads a smaller amount of input relation and compares less values before record uniqueness is ensured.

If input is sorted, projection can be done very efficiently by looping through the input relation once. Only one value needs to be stored in WS, the previous record key, if current record is equal to previous it is skipped.

Algorithm 2 NL-Projection

Require: A = input relation, P = set of attribute identifiers that should be included in result, R = output relation
Ensure: R contains no duplicate rows, and each row includes only attributes identified by P
  while more records in A **do** {ends after all unique records have been stored in result}
    p = record read from A
    if $p.key \notin WS$ **then**
      store p in WS {values $\notin$ P disregarded}
    if WS is full **then**
      m = position of last record read from A
      while more records in A **do**
        p = record read from A
        if $p.key \in WS$ **then**
          delete record p from WS
    write all records in WS to R
    delete all records in WS
    position = m-1 prepare to reread last read record

To avoid multiple runs, partitioning can be used as explained in 3.3. The relation is split into WS sized chunks and each chunk is processed separately so that the relation only needs to be read once after partitioning (note that partitioning step may require multiple runs, depending on number of output buffers available).

3.5 Join

Join is an essential operator in modern databases, and many algorithms have been devised over the years. Hash based join is a very efficient method, described and evaluated by Bratbergsengen in [5] as early as 1984. He suggests hash based partitioning to utilize WS efficiently and filtering to remove irrelevant records early thus reducing data that requires processing by the more expensive join operation.

The most basic join algorithm is nested-loop, as described in Algorithm 3. This is an efficient algorithm for data sets that fits entirely into WS, but not very good when input size increase.

Algorithm 3 NL-Join

Require: A = smallest input relation, B = second input relation, R = result relation
Ensure: R = A $\bowtie$ B
  while more records left in A **do**
    read record from A
    store record in WS
  reset B
  while more records left in B **do**
    read record from B
    for matching B records in WS **do**
      write constructed record to R
  clear WS

By utilizing the hashing methods described in [5] to improve nested loop join, an efficient join algorithm is achieved. Irrelevant records are filtered out early using a hash filter, and inputs are partitioned such that similar values are close together. Most partitions will fit into WS, allowing best-case runtime for nested loop join.

Algorithm 4 Hash-Join

Require: A = smallest input relation, B = second input relation, R = result relation, f = hash function
Ensure: R = A $\bowtie$ B
  partition A using hash function f
  partition B using hash function f
  for all partitions **do**
    NL-Join(A_i, B_i, R)

There are many variations of hash join, in addition to methods based on sorted data, like merge-sort join. These methods are useful, however, the algorithms described in this section should provide a good foundation for understanding the multi-core algorithms evaluated in this report.

3.6 Top-k

A naive way to answer top-k select queries is to compute the score for each tuple, sort the result, and return the top k tuples. This is not very efficient, the algorithm constructs a complete ordering of the table even if the end-user only needs k rows (k is often very small compared to total number of tuples).

3.6.1 Threshold Algorithm (TA)

Collection L contains one list for each attribute used in the scoring function. Each row contains a tuple $(key, value)$, where *key* is primary key for the relation, and *value* is attribute used in scoring function. The algorithm uses a threshold value (t) to determine when to stop searching. Threshold value is calculated using the scoring function on the last tuple retrieved in sorted order from each sorted list L_i. This means that it is impossible to find any tuples scoring better than t in the remaining rows. When k or more rows with score greater than or equal to t is found, the algorithm terminates. R will never hold any more than k elements, if this limit is exceeded, the lowest scored tuple is removed.

R contains the k highest scoring tuples seen, the *add* method ensures that if $R.length = k$, the lowest scored tuple is removed to make room for a new tuple with higher score. If the new tuple has a lower score, *add* will do nothing.

Algorithm 5 Threshold Algorithm

Require: L $\leftarrow$ collection of sorted lists, f $\leftarrow$ scoring function, k $\leftarrow$ requested number of tuples, R $\leftarrow$ output relation
Ensure: R = sorted set of k tuples with highets score
  S $\leftarrow$ {} {all keys seen so far}
  while relation has more rows **do**
    for all $L_i \in$ L **do**
      if $key \notin$ S **then**
        S $\leftarrow$ S $\cup\{key\}$
        score $\leftarrow f(p_0, \ldots, p_n)$
        R.add(key, score)
    $t \leftarrow f(\overline{p_0}, \ldots, \overline{p_n})$
    if $\left|\{r \in R | r.score \geq t\}\right| \geq k$ **then**
      return R
  return R

3.7 Parallel algorithms

It is not uncommon to have parallel database management systems, where data is distributed among a set of nodes connected by a network. Traditional database management systems use the shared-nothing architecture, i.e. each node contains its own CPU, memory, and storage unit. Communication is done through message passing. In parallel database management systems, disk I/O is not necessarily the bottleneck, sometimes re-allocation of data between nodes will be a dominating, and other communication can be dominating factors. A straight-forward method to perform parallel relational operations is the meeting place algorithm, described in Algorithm 6. Each node run the algorithm independently until all records are processed, nodes communicate by sending records to other other nodes based on record id.

Algorithm 6 Meeting place algorithm

Require: h ← hash formula, I ← input stream from other nodes
Ensure: relational operation performed correctly and distributed to all affected nodes
  while true **do**
    if more records in local storage **then**
      r ← next record read from local storage
      send r to node h(n)
    if more records in I **then**
      r ← node read from I
      do algebra operation on node r

An efficient partitioning strategy is essential to ensure the efficiency of parallel database management systems. There are two main categories of partitioning: horizontal and vertical. Horizontal partitioning distribute data row-wise, thereby keeping each record unchanged. Vertical partitioning distribute data column-wise by splitting each record and scattering their attributes to different nodes. It is also possible to combine these methods, or to use more domain-specific methods in certain areas. For the horizontal partitioning strategy, there are three primary placement strategies:

Round Robin Records are distributed to each node in a circular pattern. An attractive property of this method is that one always achieves good load balancing, because each node will contain the same number of records (plus minus one).

Hashing Records node address is given by a predetermined hash formula used on the primary key. Hash formula should be chosen so that records are uniformly distributed among nodes.

Range Each node covers a value range of primary key values, records are placed accordingly.

For the data distribution to be efficient, records should be placed uniformly among nodes. In addition, the insert operator should be cheap to execute. In [6] it is noted that insertion requires duplicate control, and round robin has no correlation between node keys and placement. As a consequence all nodes must check for duplicates during an insert call if round robin placement is used. For this reason, round robin is normally not a good placement strategy. In contrast, the hashing strategy allows for simple and efficient duplicate checking, and is regarded as a robust placement strategy.

Chapter 4

Multi-core processors

Multi-core processors consists of multiple CPUs or cores placed on a single chip. Typically, each core is equipped with private level one cache, other caches may or may not be shared between the cores [15]. Each core is independent and the architecture is categorized as a multiple instruction, multiple data (MIMD) system. Processors and cores usually communicate implicitly via shared memory.

In a shared-memory system a collection of independent processors are connected to a memory system via an interconnect network. There are two principal types of shared-memory systems: uniform memory access (UMA), and non-uniform memory access (NUMA).

4.1 Uniform memory access

In UMA systems all processors are connected directly to main memory ,see Figure 4.1, each core can access data from all of the other cores directly. The time to access all memory locations are the same for each core. UMA systems main advantage in relation to NUMA systems are their simplicity.

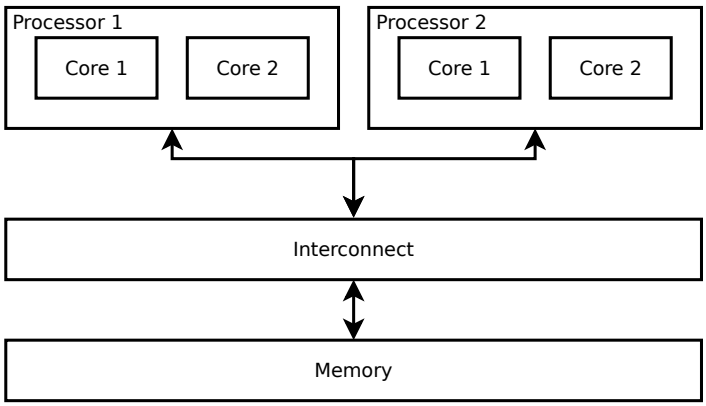


Figure 4.1: UMA architecture

4.2 Non-uniform memory access

In NUMA systems each processor is directly connected to a block of main memory, and the processors can reach each others data through special hardware built into the processors, see Figure 4.2. A memory location that a processor is directly connected to can be accessed more quickly than a memory location that must be accessed through another chip [15]. Advantages of the increased complexity is that the system can address a larger memory space, and directly memory access typically is faster than in UMA systems.

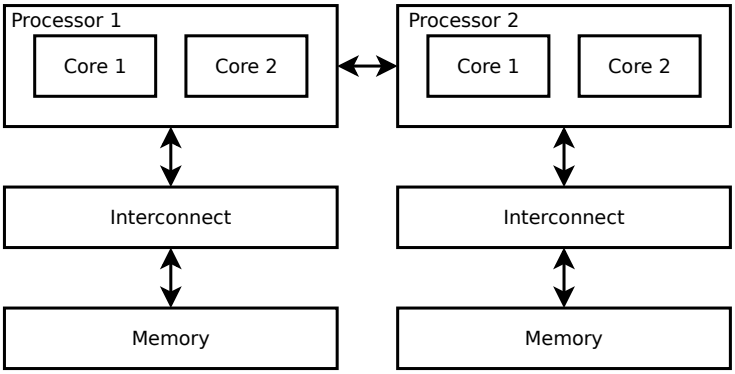


Figure 4.2: NUMA architecture

4.3 Multithreading

Multithreaded processors concurrently executes instructions from different threads of control within a single pipeline. These can be placed in two basic categories: temporal multithreaded processors and simultaneous multithreading (SMT) processors. Temporal multithreaded processors issues instructions from a single thread every clock cycle while SMT processors issue instructions from multiple threads in the same cycle [20].

Hyper-threading, Intels implementation of SMT, maps one physical core into two logical cores by decoding instruction streams in alternate cycles. The operating system sees each logical core as one processor. The main purpose of Hyper-threading is to decrease the number of dependent instructions in the pipeline. Logical threads executed on one core shares most of their cache/execution resources [9].

4.4 Cache hierarchy

Each core in a multi-core processor typically have a private level 1 cache (L1). In the Intel Nehalem family, each core contains an instruction cache, a first level data cache and a second-level unified cache (L2). Furthermore, each physical processor includes a third level cache (L3) shared by all cores in the processor. L1 and L2 are writeback and non-inclusive, L3 is writeback and inclusive so that a cache line that exists in any other level is also included in L3. An advantage of using L3 this way is that snoop traffic between processors can be reduced. [9]. Table 4.1 shows data cache parameters for the Intel Core i7 processor, a member of the Nehalem family.

Level	Capacity	Access latency (clocks)
L1	32 KB	4
L2	256 KB	10
L3	8 MB	35-40

Table 4.1: Cache parameters of Intel Core i7 Processors

4.5 Programming frameworks

This section contains an overview of selected shared-memory programming frameworks and libraries. Advantages and disadvantages are noted for each framework. Data from this section are used to choose a suitable framework for implementation in Chapter 6.

4.5.1 Pthreads

Pthreads is short for POSIX threads and specifies a low-level application programming interface (API) for multithreaded programming in Unix-like operating systems. It is implemented as a C library and is widely used in both real-life implementations and research papers.

Thread distribution Programmers can create an arbitrary number of threads using the function pthread_create. The operating system (OS) is responsible for distribution threads among cores. Pthreads use the fork-join paradigm, where all threads spawn from a master thread and are joined back to one thread before program termination (using pthread_join).

Memory sharing Memory sharing is achieved by giving all threads access to global variables, and keeping local variables private for the thread executing the function [15]. It is possible to pass pointers to pthread_create, typically used to give each thread a unique id and working set.

Synchronization Pthreads include mutexes, read-write locks, conditional variables, and barriers for synchronization. In more involved cases semaphores can be used (not included in the pthread definition, but provided in by semaphores.h)

4.5.2 OpenMP

OpenMP is a mature and widely used API for shared memory parallel programming. Code is parallelized using compiler directives and C code. Programmers define areas that can be executed in parallel and OpenMP handles the threading [15]. This framework is more high-level than Pthreads, it aims make the creation of large-scale high-performance programs easier. OpenMP was explicitly designed to allow programmers to incrementally parallelize existing serial programs. It is best equipped with pragmas for loop-level parallelism, typically found in compute intensive workloads. For more general workloads, the simplicity may be counter-productive.

Thread distribution OpenMP allows programmers to give the compiler hits about code sections that can be run in parallel using pre-compiler directives (pragmas). Number of threads started is system dependent, but most systems will start one thread per core. Programmers can explicitly set number of threads that should be started using an environment variable (OMP_NUM_THREADS), or a function call. However, allowing OpenMP to handle thread count is normally the best solution. A classical example is OpenMPs pragma for loops, this will automatically partition data and execute loop in parallel. OpenMP also have more complex (or specific) directives like simd, that indicates that data parallelism can be utilized.

Memory sharing Each OpenMP thread can use shared, and private memory. By default, shared variables are determined by static scope, as in sequential C code. For more fine-grained control OpenMP allows programmers to define variables that should be regarded as thread private or shared. It is common to define all variables as private by a call to default(none), and specify all shared variables explicitly using the shared directive. If a private variable is declared outside the parallel region, OpenMP generates one copy for each thread.

Synchronization Synchronization is achieved by defining critical sections, each section can be named for fine-grained control. In some cases the atomic directive can be used to increase performance, this directive is designed to exploit special hardware instructions. For reduction operators (sum, product, etc.) OpenMP provides a reduction clause. In addition, simple locks, and barriers are provided. [15]

4.5.3 Cilk++

Cilk++ is a commercial version of the Cilk language developed at the MIT for multithreaded parallel programming. It is an extension to the C++ language adding three basic keywords to process loops in parallel, launch new tasks, and synchronize between tasks [12]. Intel, the owner of Cilk++, claims that the runtime system operates smoothly on systems with hundreds of cores. Cilk++ has features similar to OpenMP.

Thread distribution Cilk++ use an efficient work-stealing scheduler to automatically distribute tasks among available cores. Tasks can either be implemented as separate functions or in iterations of a loop. The keyword _Clik_spawn are used to modify a function call to tell the runtime system that the function may run in parallel. There is also a keyword equivalent to OpenMPs loop pragma that automatically run a for-loop in parallel.

Memory sharing Memory is shared similar to OpenMP, but Cilk++ introduces a new concept, hyperobjects. Hyperobjects enables many parallel threads to coordinate updating of a shared variable in the form of reducers, holders and splitters. Each thread have its own unique view of a reducer.

Synchronization The keyword _Cilk_sync can be used to indicate that the current function can not continue past a certain point in parallel with its spawned children. Additionally, Cilk++ provides reduction operators to allow threads to access nonlocal variables safely. Locks can be implemented in various ways, for instance, pthread mutexes can be used directly.

4.5.4 UPC

Unified Parallel C (UPC) is an extension of the C programming language designed for high performance computing on large-scale parallel machines. The language provides a uniform programming model for both shared and distributed memory hardware [3]. UPC is actively developed at Berkley, latest version released in 2012.

Thread distribution UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program startup time, typically with a single thread of execution per processor. Number of threads can be changed using a variable named THREADS.

Memory sharing The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any thread. Each thread can use a combination of shared and private data. By default, data is considered private, data can be shared using a C type-qualifier.

Synchronization Thread interaction is explicitly managed by the programmer through primitives provided by the language. UPC includes locks, barriers and memory fences.

4.5.5 SWARM

SWARM is short for "SoftWare and Algorithms Running on Multicore". It is a open source programming framework for multi-core hardware. SWARM is built on POSIX threads and allows developers access to low-level functionality as well as high-level constructs for more convenient parallelization [2].

Thread distribution Threads can be distributed by defining functions as tasks that can be run in parallel, or by using loop parallelism similar to OpenMP. Several basic loop parallelization directives are included to implicitly partition loops among cores and so that they can be executed in parallel. Both block and cyclic partitioning interfaces are supported.

Memory sharing SWARM provides directives, to dynamically allocate and release shared data structures. For thread private variables, a replication primitive is provided to create a unique copy for each core.

Synchronization Common synchronization primitives like barrier and reduce are provided. In addition pthread mutexes can be used to implement locks.

4.5.6 Phoenix MapReduce

Phoenix is a shared-memory implementation of Google's MapReduce model for data-intensive processing tasks [18]. Phoenix can be used to program multi-core chips as well as shared-memory multiprocessors. The framework is developed at Standford University, latest update was released in 2011. The current implementation provides an API for C and C++.

Thread distribution The user specifies the algorithm using two functions, map and reduce. The map function processes the input data and generates a set of intermediate key/value-pairs. The reduce function merges the intermediate values which have the same key. Phoenix use threads to spawn parallel map and reduce tasks. Tasks are dynamically scheduled across available processors (or cores) in order to achieve balanced load and to maximize throughput.

Memory sharing Shared-memory buffers are used to facilitate communication without excessive data copying. However, the MapReduce paradigm is best suited for independent tasks that can be merged at the end with minimal data sharing.

Synchronization The MapReduce paradigm does not provide explicit synchronization constructs. Synchronization is implicitly performed in the reduce operation. As a consequence, MapReduce is not a very general method for shared-memory programming.

4.5.7 Intel TBB

Intel Threading Building Blocks (TBB) is a commercial library that aims to help developers leverage multi-core performance without being threading experts [16]. This library allows developers to program multi-core processors in a high-level view using tasks. It can coexist with other multi-core libraries, like pthreads [10]. TBB emphasize scalable, data parallel programming, enabling multiple threads to work on different parts of a collection. According to Intel TBB is a widely used C++ template library for task parallelism.

Thread distribution Developers specify tasks that are automatically mapped to threads in an efficient manner. TBB supports nested parallelism, making it possible to build larger parallel components by combining smaller parallel components. Loops can be parallelized by moving the loop body into a functor (C++ objects that can are treated as a function) and passing the functor in a library call.

Memory sharing TBB provides two memory allocator templates designed to improve scalability and avoid false sharing. Two objects allocated by cache_aligned_allocator are guaranteed to not have false sharing. For memory sharing, TBB provides concurrent containers with fine-grained locking and lock-free techniques.

Synchronization Mutual exclusion is implemented by mutexes and locks. TBB supports a number of synchronization constructs, like spin_mutex, scoped_lock, and recursive and non-recursive mutexes. There is also support for atomic operations using template functions.

4.5.8 Comparison

Table 4.2 give an overview of frameworks previously discussed. Compiler- and language support is based on data from recent articles and information posted on the official web sites for each framework.

Name	Language support	Compiler support [1]	API type	Recommended parallelization	Notable features
OpenMP	C, C++, Fortran	Good	Compiler directives	task or data	
Cilk++ (Intel)	C++	Average	C++ keywords and hyperobjects	task or data	Race detector, Performance analyzer
TBB (Intel)	C++	Good	Special C++ objects (lambda functions available in a few compilers)	task or data	Concurrent container classes (hash maps, vectors and queues)
Pthreads	C	Good	types and procedure calls	task	
SWARM	C	Good	types and procedure calls	task	
UPC	C	Average	keywords, types, procedure calls, and expressions	task or data	
Phoenix++	C++	Good	C++ templates	map-reduce	

Table 4.2: Feature comparison between multi-core frameworks

4.6 Code optimization

In shared-memory programming it is important to consider common bottlenecks to achieve an acceptable performance. If algorithms are parallelized without careful planning, they can easily be slower than sequen-

[1] Good: All major compilers. Average: At least one open source compiler. Poor: Only Proprietary compilers.

tial versions due to less optimal memory access patterns, synchronization and thread creation costs, or false sharing. It is essential to avoid false sharing and minimize time in critical sections.

False sharing is a performance degrading access pattern that can occur in systems with concurrent caches. It is encountered if a thread periodically access data that never will be altered by another party and this data is placed in the same cache block as data that is altered. There are different methods to handle this. The most straight-forward method is to partition data so that such a situation will never occur. If this is not possible, data structures can be padded so that they fit in an entire cache line, making it impossible for data structures used by other threads to be placed in the same block.

Critical sections are sections in the code that require sequential execution, i.e. are unable to exploit parallelism. In some cases critical sections must be used to avoid race conditions. Programmers typically need to place results in a shared data structure. Insertions and reads from this structure in normally have to to be synchronized to ensure data consistency. It is normally not a good idea to lock the entire structure, as this will induce a big sequential section, rather fine-grained locking should be used so that multiple threads can work on the same structure concurrently. This is closely related to Amdahls law: $\frac{1}{(1-P)+\frac{P}{S}}$, where P is parallel part and S is sequential part. Speedup is limited by the sequential fraction of the program. Fine-grained locking can in many cases be used to reduce the sequential fraction can be reduced.

Obviously, shared-memory algorithms will take advantage of traditional optimization techniques as well. Efforts should be made to improve cache locality and sequential memory access. General purpose processors are best on sequential access, and have efficient cache hierarchies that should be exploited to improve algorithm performance. However, the authors of [4] note that modern processors are very effective in hiding cache miss latencies through multi-threading. Therefore computation and synchronization costs can be just as important when shared-memory algorithms.

Techniques for Non-uniform memory access (NUMA) are evaluated in [1]. This article states that sequential scans of non-local memory heavily profit from pre-fetching and cache locality. Based on micro-benchmarks on NUMA-affine versus NUMA-agnostic data processing, authors define three basic rules for NUMA-affine scalable multi-core parallelization:

C1 Avoid random writes to non-local memory locations

C2 If remote reads are necessary, use a sequential access pattern

C3 Avoid waiting for other nodes

In conclusion, algorithms should be cache-concious, use sequential memory access, avoid false sharing and spend minimal time in critical sections. Data structures can have a major impact on performance, both for sequential and parallel algorithms. Concurrent data structures will in many cases take advantage of fine-grained locking mechanisms to allow multiple threads to access different regions at the same time.

Chapter 5

Memory-based algorithms

In this chapter algorithms optimized for memory-based architectures is considered. Efficient memory access, use of cache, and synchronization are dominating factors of performance in memory-based systems. For memory-intensive algorithms like sorting, bus bandwidth is also an important consideration.

5.1 Join

5.1.1 Hash join

Efficient hash join algorithms for multi-core processor were evaluated by Blanas et al. in [4]. Authors discovered that a simple hash join algorithm is very competitive to the other more complex methods evaluated. Algorithms explored in [4] are very similar to the traditional traditional hash join algorithms introduced in Chapter 3. They have three steps:

1. Partition (optional if smallest relation fits in main memory)
2. Build
3. Probe

Partitioning is used to avoid cache-misses such, relations are split so that that each partition fits in the CPU cache, equivalent to partitioning used in traditional parallel databases. Build phase writes the smallest relation into a hash table. Probe phase use same hash function on the larger relation to limit the search, and check the keys directly to verify results. Algorithm 7 shows a possible implementation of simple hashjoin. Loops commented with "in parallel" indicates that the data is evenly distributed among available threads and run in parallel.

Authors found that modern processors are very effective in hiding cache latencies through multi-threading. They concluded that the costs of partitioning can be higher than the benefit of less cache-misses.

Algorithm 7 HashJoin

Require: A ← left input relation, B ← right input relation, R ← result
Ensure: R ← A ⋈ B
T ← empty hashtable
for all $a \in A$ **do** {in parallel}
$k \leftarrow hash(a)$
lock bucket k
T.put(k,a)
unlock bucket k
sync
for all $b \in B$ **do** {in parallel}
$k \leftarrow hash(b)$
if $k \in T.keys()$ **then**
for all $a \in T.get(k)$ **do**
if b.key = a.key **then**
lock R
R.add(merge(a,b));
unlock R

5.1.2 Sort-merge join

In [1], Albutiu et al. develop a sort-based parallel join method that scales (almost) linearly with the number of cores. They focus on the NUMA architecture, basing their design on the NUMA-rules presented in Section 4.6. Input data is chunked into equally sized chunks among the workers, each worker get one chunk from relation R and one from S. After the data is distributed, the basic algorithm is presented in Algorithm 8 and consists of three phases:

1. Sort S_i
2. Sort R_i
3. Merge R_i S

The algorithm does not merge chunks to form a globally sorted output, instead the sorted runs are joined in parallel. Authors argue that sorted output is not a requirement for the join operator, and avoiding the global merge reduce synchronization overheads. In phase 3 each worker merge its chunk with the entire S relation (that is, R_i is compared with each S_i, exploiting the local sorting). Phase 3 can begin before phase 2 is completed, allowing the algorithm to use pipelined parallelization. This means that the algorithm does more work than a traditional sort-merge, but improves performance drastically by executing operations in parallel.

Algorithm 8 B-MPSM

Require: A ← left input relation, B ← right input relation, R ← result
Ensure: R ← A ⋈ B
distribute input data into equally sized chunks among workers
BEGIN WORKER {parallel section}
Sort B_i
sync
Sort A_i
for all $B_i \in B$ **do**
$R_i \leftarrow A_i \bowtie B_i$
END WORKER {barrier}
R ← $R_1 \cup \ldots \cup R_n$

5.2 Top-k

5.3 Parallel top-k select

To utilize multiple cores for top-k, it is possible to run parts of TA in parallel. This can achieve a certain degree of parallelism, but TA is a synchronization heavy algorithm. This section will gradually increase the degree of parallelism, while attempting to keep synchronization costs low.

5.3.1 Vertical input partitioning

If the scoring function $f(\ldots)$ is in the form $ax + by + cy + \ldots$, terms can be calculated in parallel. With expensive terms this may have a positive impact on performance. Using as many threads as attributes, costs are reduced by a factor of n, where n is the number of attributes. This can be accomplished using one master thread to execute the algorithm and a pool of worker threads calculating the terms on demand each time $f(\ldots)$ is called.

A shortcoming of this approach is that the number of threads are limited by number of attributes. Only the vertical potential for parallelism is exploited. In addition, worker threads can end up periodically waiting for the master thread. A better solution is to exploit the data parallelism in such a function and use SIMD operations. A typical processor allows up to four terms to be calculated simultaneously without any extra overhead if SIMD primitives are used. A great advantage of SIMD is that they can be utilized with minimal overhead, there is no need to create threads to exploit data parallelism.

5.3.2 Horizontal input partitioning

Another way parallelize TA is to calculate one row per thread in a round robin fashion. This allows for a greater degree of parallelism, but requires some synchronization, see algorithm 9.

Algorithm description

The function processRow contains lines 3-9 from the sequential algorithm, in addition to some synchronization logic. When values are written to R, a write lock is required. Checking if a key has been processed and adding it to the set of processed keys is an atomic operation. This may result in threads waiting to access collections.

Algorithm 9 HIPTA

Require: L ← collection of sorted lists, f ← scoring function, k ← requested number of tuples
Ensure: R ← sorted set of k tuples with highets score
  S ← {} {all keys seen so far}
  while relation has more rows **do**
    for i ← 0 to threadcount **do**
      spawn processRow(L, f, S, R)
    sync
    t ← $f(\overline{p_0}, \ldots, \overline{p_n})$
    if $\left|\{r \in R | r.score \geq t\}\right| \geq k$ **then**
      return R
  return R

Chapter 6

Implementation and evaluation

This chapter intends to give a detailed description of the implementation and evaluation of HIPTA. First, methods and tools are described, including test environment and parameters. Second, implementation is described in terms of programming language, data structures and shared memory framework. Finally, algorithm is evaluated based on several benchmarks.

6.1 Methods and tools

Tests are executed on a Linux system equipped with two Intel Xeon 5650 processors, described in Table 6.1. Code is compiled using g++ with optimization level -O2 and the C++0x standard.

Processors	2x Intel Xeon 5650 @ 2.67 GHz
Cores per processor	6
Contexts per core	2
Cache size, sharing	12MB L3, shared
Memory	125GB
Operating system	Debian 7.0 (wheezy)
Kernel	3.2.0-29-generic

Table 6.1: Platform characteristics

Each test is executed 10 times and the median is displayed. An important criteria is the effect of different degrees of parallelism on performance, therefore a predetermined set of thread counts are used in all tests, described in Table 6.2. The following parameters are evaluated in Section 6.2:

k Number of records requested (i.e. top-k records)

n Number of attributes considered in scoring function

m Number of records in input relation

d Input distribution

t Number of threads

p_{min} Minimum partition size

p_{max} Maximum partition size

Thread count	Test criteria
2	Sequential performance for TA and HIPTA
2	Performance of HIPTA with minimal amount of threads
12	Performance of HIPTA with one thread per physical core
24	Effect of Hyper-Threading (24 threads, where 12 is physical and 12 logical)
1042	Locking and partitioning overhead vs. memory latency hiding

Table 6.2: Reasoning for thread counts

Three test relations are generated beforehand and stored as plain text files. Relations are used to test uniform, positively correlated, and negatively correlated input. All attributes are represented as 8B integers. Before tests are executed, input relations are read into memory, disk access should not have any effect on testing.

6.2 Implementation

Algorithms are implemented in C++, and Pthreads are used for parallel functionality. Aside from the additional synchronization and partitioning code, sequential and parallel algorithms use the exact same logic and data structures. Consistency and synchronization is implemented using atomic operations supported by GCC, in addition to standard pthread mutexes where necessary. To minimize thread creation/destruction overhead, a thread pool is used. Tasks are pushed to a concurrent queue by the master thread, while idle worker threads repeatedly request more work by polling the queue. Compared to creating new threads for each iteration, this technique proved very efficient.

Input relation is represented as an array where data is stored row-wise. Sorted access for attributes is implemented using one sorted array per attribute, each entry includes a pointer to the input relation. Similarly, results are stored in an array of pointers. However, the result structure includes additional meta data, like size (number of records processed) and capacity (requested number of records).

Each time a new result is processed, the result structure needs to be updated. To avoid race conditions, worker threads lock the entire structure before before updating. The structure is kept sorted at all times using memory move operations. Obviously, this is a critical section and a bottleneck for the parallel algorithm. In order to allow more fine-grained locking, other data structures, like binary trees were considered. However, the array structure is easy to implement, and proved efficient in combination with a small optimization. The lowest scored value is stored in a separate variable and compared to the current record before result is updated. This is done using atomic operations, thereby incurring only a small synchronization overhead. This allows records that are not relevant to be discarded without searching the result structure.

6.3 Experimental evaluation

In this section, HIPTA is evaluated using a number of benchmarks. Each parameter is examined and discussed. If results differ from the expected outcome, possible reasons are explored. Input data is specified in a table before each test and results are presented using a standard set of graphs which display runtime, speedup, work, and variations in test results.

6.3.1 Number of requested records

To evaluate the increasing k has on run time, speedup and work amount, both algorithms are executed with parameters in Table 6.3. It is expected that increasing k will increase the total runtime for both algorithms,

but have a smaller effect on the parallel version. If that is the case, HIPTA shows a positive degree of scale-up. However, a bigger k will also force HIPTA to spend more time synchronizing, therefore that speedup may be limited.

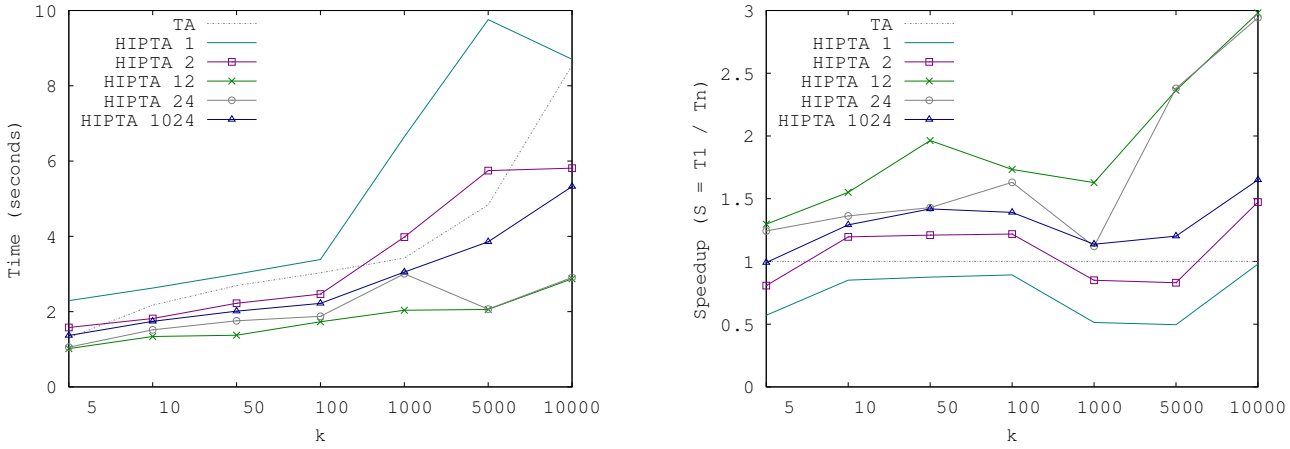
Parameter	Value(s)
k	5, 10, 50, 100, 1000, 5000, 10000
n	4
m	128M
d	Uniform
t	1, 2, 12, 24, 1024
p_{min}	50
p_{max}	500

Table 6.3: Test 1, input parameters

As Figure 6.1 shows, increasing k will increase the amount of work required to produce a result. This is no surprise, requesting a bigger set of results will obviously require processing an increased number of records. Another observation is that the initial assumption was correct, HIPTA does take advantage of an increasing amount of work. However, speedup is far from linear, the best result for two threads shows a speedup of approximately 75% compared to linear growth. For higher thread counts, the relative speedup decreases, twelve threads only obtain a speedup is only 25% compared to linear.

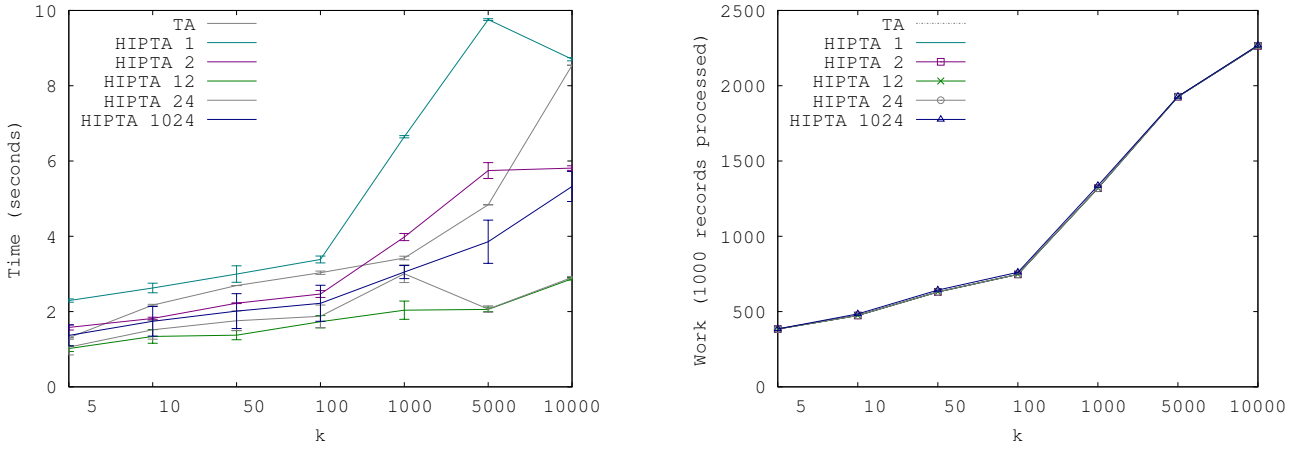
Increasing number of threads to more than processors physical core count is sometimes used to hide memory latencies. When one thread stalls, another can work on other tasks. This in turn, allows the program to utilize all cores for useful work, even if some threads are stalling. In this case, this had no effect. It is likely that memory stalls would be in a critical section for HIPTA, therefore an additional thread will not be able to do any processing before the stall is resolved. This optimization seems to have an adverse effect.

An unexpected observation is that all runs of HIPTA shows a decrease in performance when k reaches 1000, this may be the effect of unsuited p_{min} and p_{max} values for the current input. These parameters are important to achieve an efficient partitioning strategy, they are evaluated in Section 6.3.3.



(a) Execution time

(b) Speedup



(c) Execution time with variation

(d) Work

Figure 6.1: Test 1, increasing k

6.3.2 Input size

The input size for top-k search can variate in two dimensions. Horizontal input size be varied by increasing/decreasing number of attributes used in scoring function, while horizontal input size can be changed by increasing/decreasing number of records in relation.

This test evaluates both horizontal and vertical scaling. First, horizontal scaling is evaluated using the parameters defined in Table 6.8. Second, horizontal scaling is evaluated using parameters defined in Table 6.5. It is expected that increases in both dimensions will show a positive effect on speedup due to an increased amount of work available for each thread.

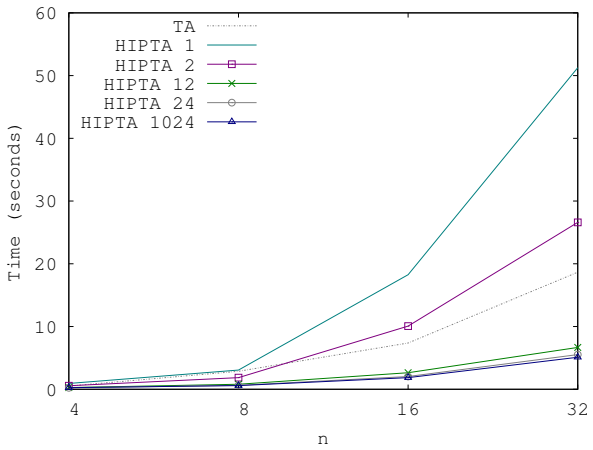
An interesting feature of this test is the effect of synchronization. Horizontally scaling should have minimal effect on synchronization costs as the score calculation is independent. In turn, this will give HIPTA an increased speedup as more work can be done in parallel without the additional synchronization overhead. For vertical partitioning, more synchronization will be needed as number of records increase, therefore it is expected that horizontal scaling will show better results than vertical.

Note that the vertical input size in Table 6.8 is low compared to other tests in this paper. The reason for this is that moving test data into memory is very time-consuming and this test has to read data once for every step in the x-direction. In addition, an increase horizontal input will add more work and reduce the effect of smaller vertical input.

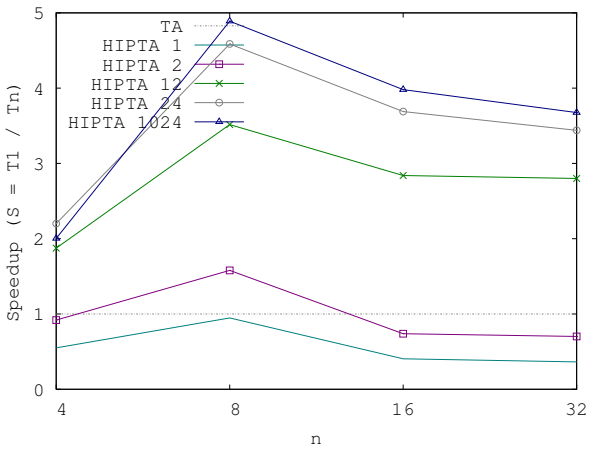
Parameter	Value(s)
k	1000
n	4, 8, 16, 32
m	16M
d	Uniform
t	1, 2, 12, 24, 1024
p_{min}	200
p_{max}	400

Table 6.4: Test 2, input parameters

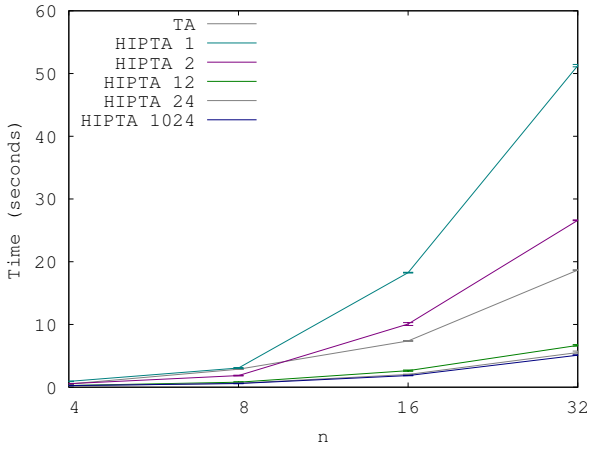
Figure 6.2 shows that horizontal scaling does increase speedup for HIPTA. This effect is limited, however, there is a peak at 8 attributes, after that performance actually start to decrease. Looking closer at the input size reveals that each attribute adds 8 byte to data amount required per score calculation (stored consecutive in memory). Using 8 bytes requires 64 bytes, exactly the line size for L1 and L2 data cache. It is common knowledge, that reading and writing cache line sized chunks is optimal performance wise, therefore decreased performance is not a big surprise. A more interesting observation is that the sequential algorithm shows a smaller reaction that the parallel. This can not be attributed to synchronization costs, but may have something to do with memory bandwidth. Each thread in HIPTA needs to use an increased amount of memory bandwidth, increasing the likelihood of blocking other threads. It is possible that a two-step scoring function, calculating 8 attributes (cache line size) per round will reduce this effect.



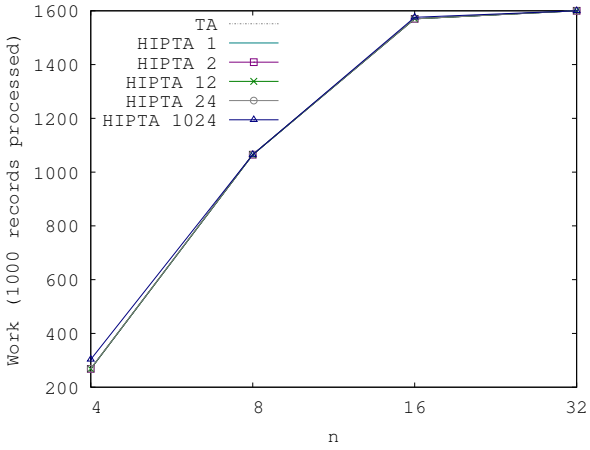
(a) Execution time



(b) Speedup



(c) Execution time with variation



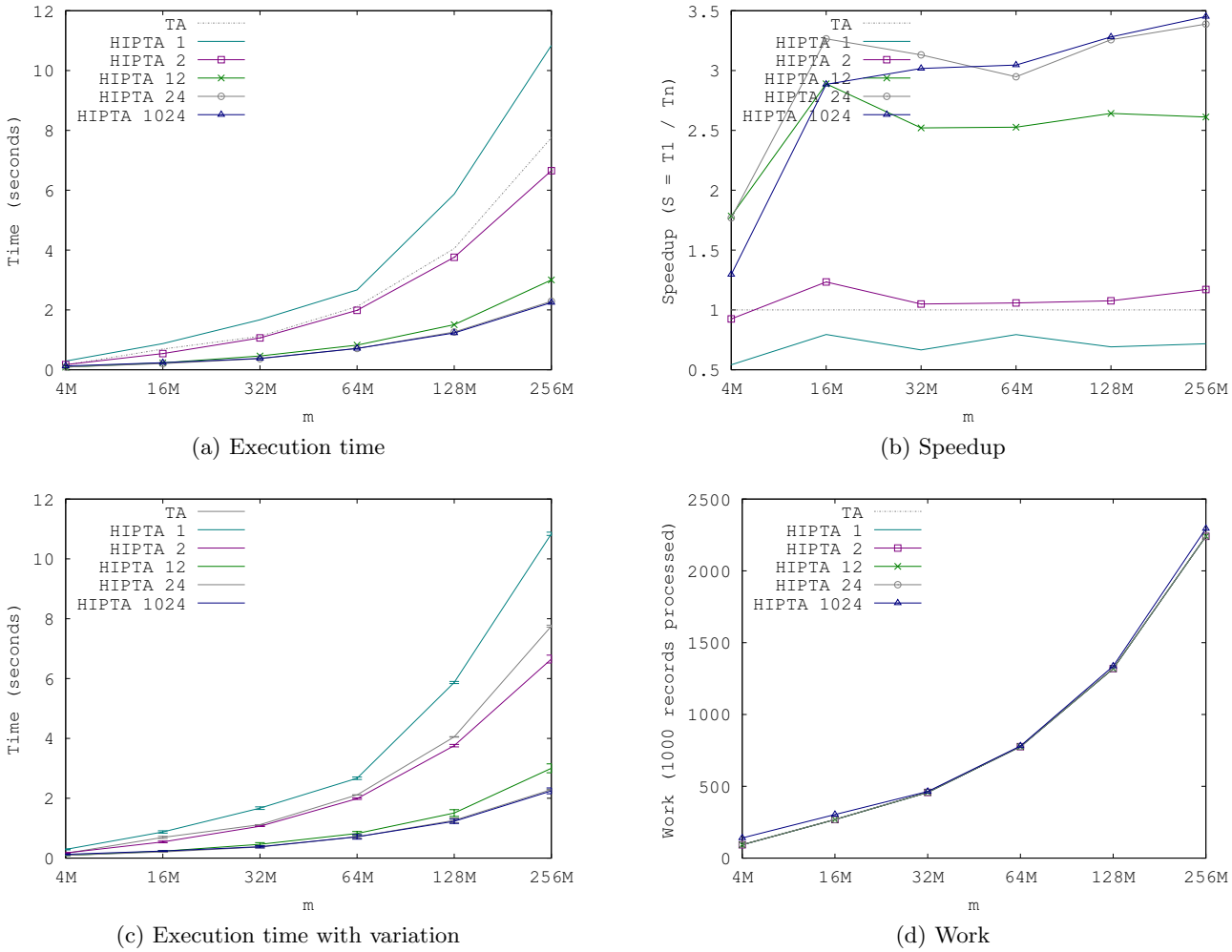
(d) Work

Figure 6.2: Test 2, horizontal scaling

Figure 6.3 shows that in increased input size give increased speedup as expected. However, speedup does not increase much when relation size is increased to more than 16M records. Comparing results for different thread counts, one can see that low thread count shows only a slight increase when vertical input size increase. Runs with higher thread counts, however, give a significant improvement from 4M to 16M. It is assumed that 16M is a sufficient input size to fully utilize all cores in the test environment. For an increased number of physical cores, parallel speedup is expected to increase with input sizes greater than 16B.

Parameter	Value(s)
k	1000
n	4
m	4M, 16M, 32M, 64M, 128M, 256M
d	Uniform
t	1, 2, 12, 24, 1024
p_{min}	200
p_{max}	400

Table 6.5: Test 3, Input parameters



(a) Execution time

(b) Speedup

(c) Execution time with variation

(d) Work

Figure 6.3: Test 3, vertical scaling

6.3.3 Partition size

The partition size should be adjusted to minimize synchronization, and at the same time allow the algorithm to terminate as soon as possible. To achieve optimal performance, a trade-off has to be made. The algorithm can only terminate in synchronization points, this is where the threshold value is evaluated, therefore partition size must me sufficiently small to avoid too many unnecessary calculations. At the same time it need to be sufficiently big, so that synchronization costs are kept low. To achieve this, partition size is determined by the following heuristic:

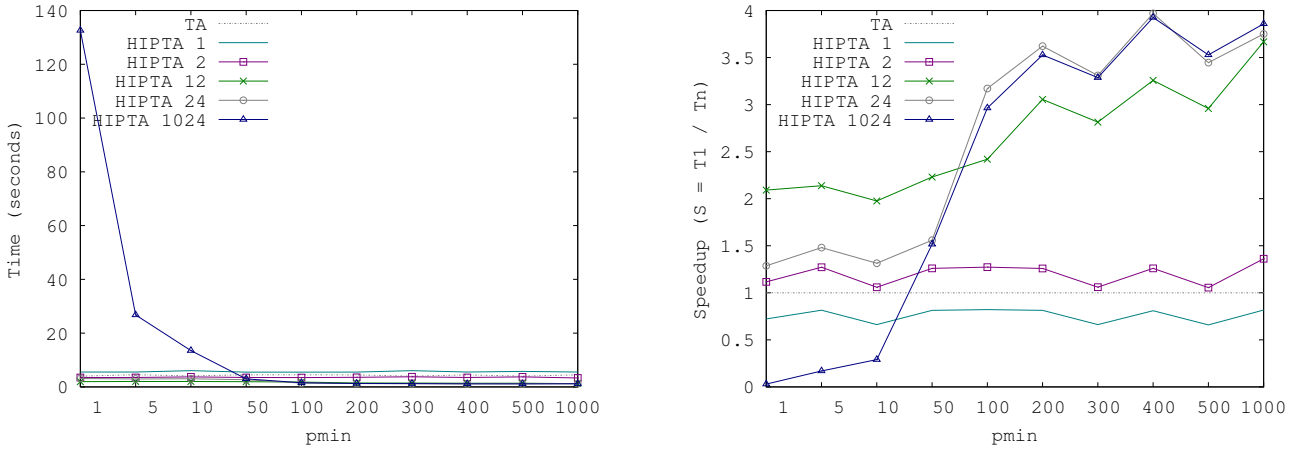
$p = k/t, p_{min} \leq p \leq p_{max}$

In this test, p_{min} and p_{max} is looked at separately. First, p_min is tested by starting at size of 1 and gradually increasing up to 1000 (see Table 6.6). Second, p_max is tested in the same manner, detailed parameter information is placed in Table 6.7. Finally results are discussed in relation to the heuristic.

Parameter	Value(s)
k	1000
n	4
m	128M
d	Uniform
t	1, 2, 12, 24, 1024
p_{min}	1, 5, 10, 50, 100, 200, 300, 400, 500, 1000
p_{max}	1000

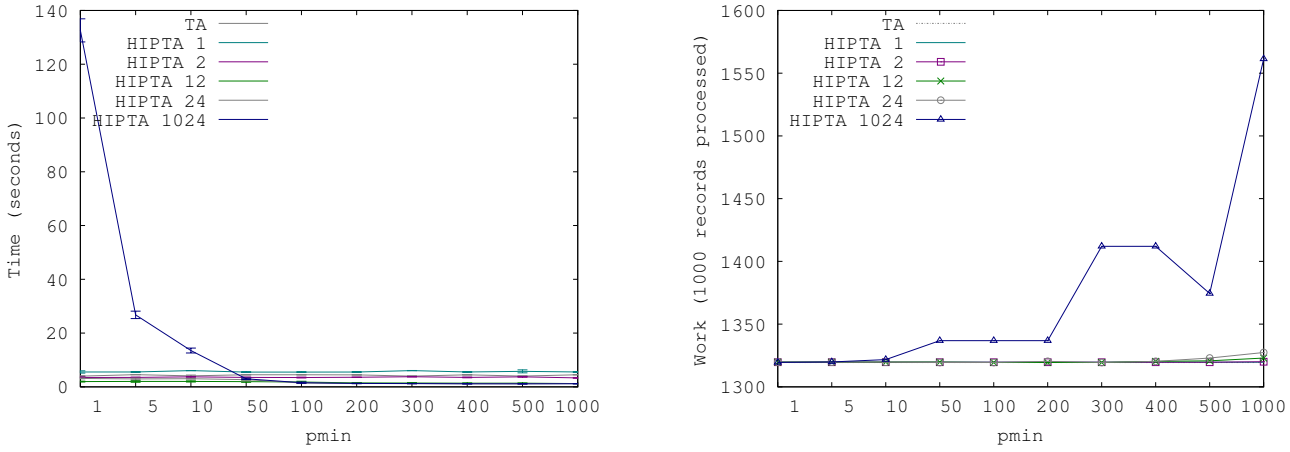
Table 6.6: Test 4, input parameters

Figure 6.4 shows that a small p_min value impairs the performance of runs with a high number of threads, but have no apparent effect on runs with low thread count. The reason for this is the heuristic used, when t is small, k/t will be larger than p_min, and the parameter will have no effect. When t is large enough that k/t ¡ p_min, p_min is used to increase partition size to reduce synchronization costs. Obviously p_min of 1 does not give any gain. However, when p_min is raised to 400, there is a peak in performance. These results confirms assumptions earlier in this section. As partition size in increased, both run-time and records processed before producing a result is increased. When partition size reaches 100, speedup increase at a significantly lower rate, it is expected that speedup will decrease when a sufficiently high partition size is reached.



(a) Execution time

(b) Speedup



(c) Execution time with variation

(d) Work

Figure 6.4: Test 4, minimum partition size

Parameter	Value(s)
k	1000
n	4
m	128M
d	Uniform
t	1, 2, 12, 24, 1024
p_{min}	1
p_{max}	1, 5, 10, 50, 100, 200, 300, 400, 500, 1000

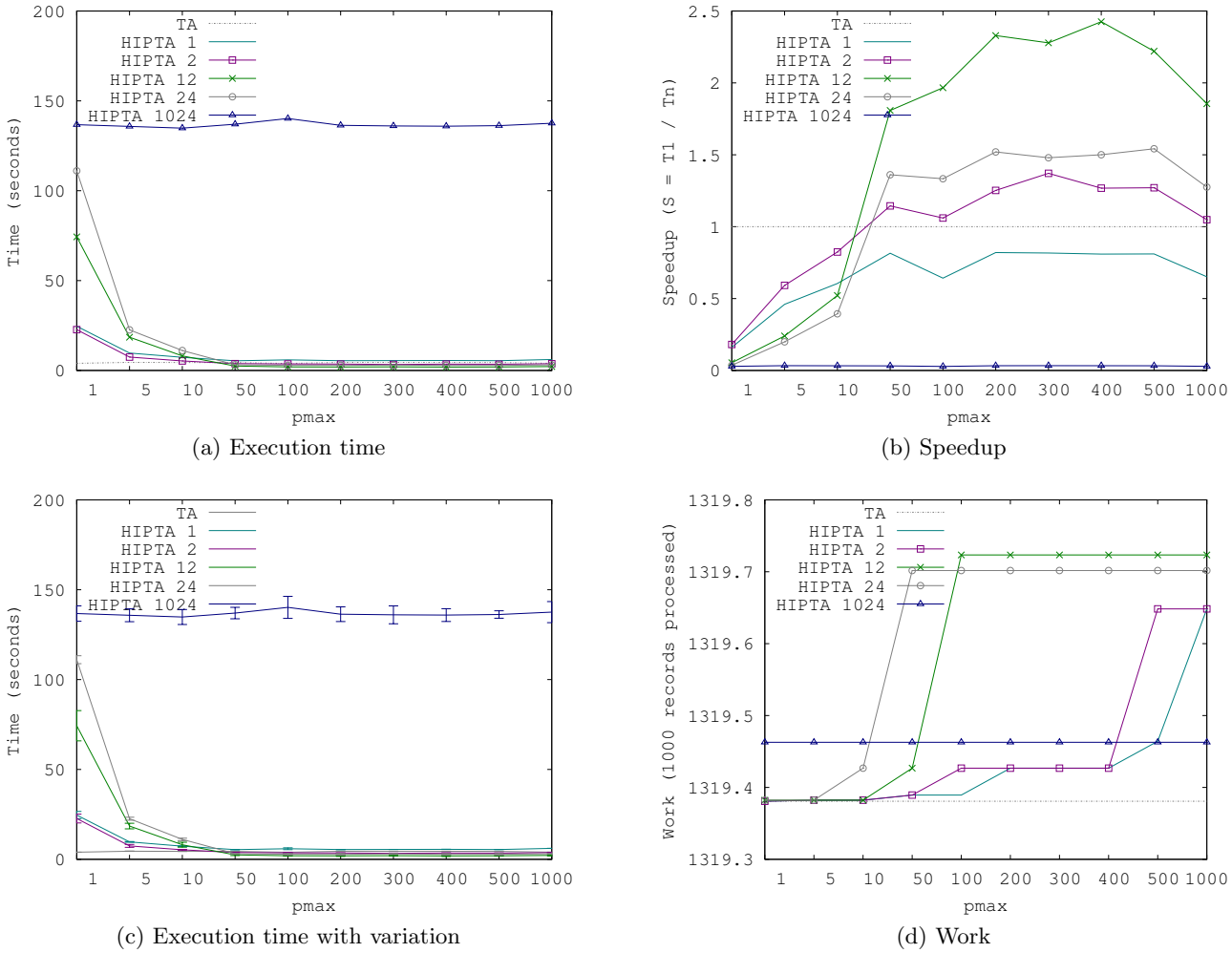
Table 6.7: Test 4, input parameters

In Figure 6.5 the results of increasing p_{max} is displayed. An interesting observation is that the run with 1024 threads is very slow compared to runs with lower number of threads in all cases. This is due to the low p_{min} value and should be disregarded, p_{max} has no effect when the number threads are to high.

When p_{max} reaches 400, there is a peak in speedup, similar to the results for p_{min}. This is attributed the input data, a partition size of 400 seems to be the ideal for the input used. This is also an indication of a poor heuristic function, only k is taken into account, when the input distribution and size is a significant factor in work required. Instead of simply setting $p_{min} = p_{max}$, a better heuristic is needed to achieve optimal performance for different input data.

Looking at the work graph in 6.5, one can see a similar pattern for all of the HIPTA runs, a flat area where work is similar to the sequential algorithm, following a steep raise, ending in a flat area at peak value. This

may seem somewhat peculiar at first, but it has a simple explanation, it can be explained by the heuristic function. For two threads, p = 1000/2 = 500, this is exactly where the peak value is found. Before p_max = 500 is reached, partition size will be limited by p_max, and work will be relatively low. In the transition from p_max = 400 to 500, the heuristic function will increase partition size to its peak value of 500 and stagnate. Same reasoning can be used for other thread counts.



(a) Execution time

(b) Speedup

(c) Execution time with variation

(d) Work

Figure 6.5: Test 5, maximum partition size

6.3.4 Thread count and input distribution

Based on earlier testing, parameters well suited for the parallel algorithm are chosen. This test is two-fold, first, it should provide a view over algorithm behavior as number of threads are increased. Second, it should examine the effect of positive- and negative correlation between attributes. It is expected that run-time will be reduced with positive correlation as less records need to be processed to reach the threshold. For negative correlation, the opposite effect is expected. Reducing number of records processed will also reduce the effect of parallelization, as synchronization and thread allocation overhead per record will be increased.

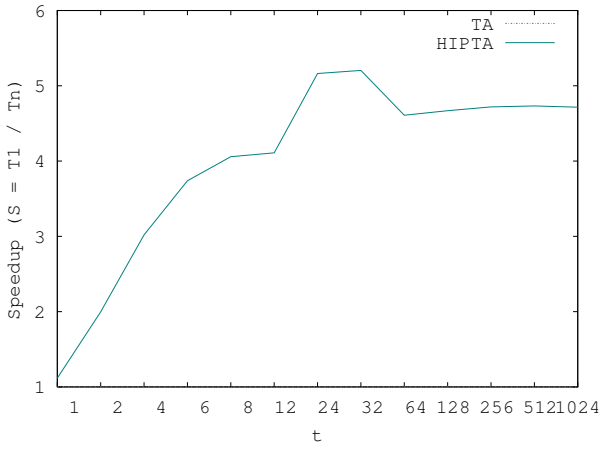
Parameter	Value(s)
k	1000
n	8
m	256M
d	Uniform, positive correlation
t	1, 2, 4, 6, 8, 12, 16, 24, 32, 64, 128, 256, 512, 1024
p_{min}	200
p_{max}	400

Table 6.8: Test 6, input parameters

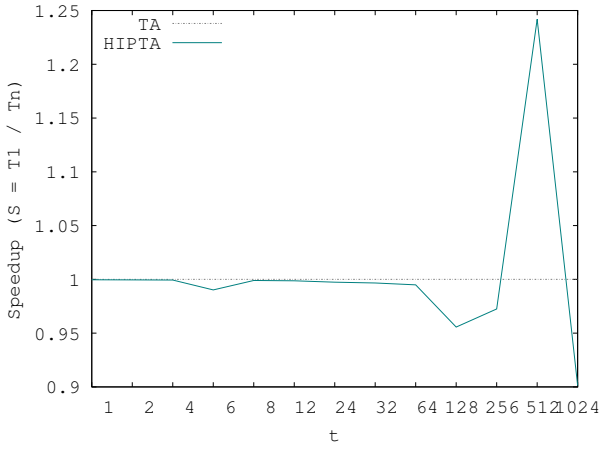
In this test, only speedup are included for positive and negative correlation. Work are in both cases very similar to work amount required to process uniform input. Positive correlation generally requires significantly less work, while negative correlation increase required work amount. However, form of the work graph is the same in all cases. Results are reported in Figure 6.6.

Starting with uniform distribution, one can see that increasing thread count from one to two give near perfect speedup of 2, further increase in thread count give improved speedup, but at a much slower rate. Four threads perform at about 75% of perfect speedup, twelve threads perform at 33%, and twenty four threads perform at 20%. Further increase in thread count gives no performance gain. This decrease in relative performance gain is attributed to the structure for storing results. Each time a record is processed it is inserted in a shared data structure, this limits the algorithms capability of utilizing an increased amount of parallel units. And, obviously, due to the limited amount of cores in the test environment (12 physical, 24 logical). Speedup is limited by Amdahls law.

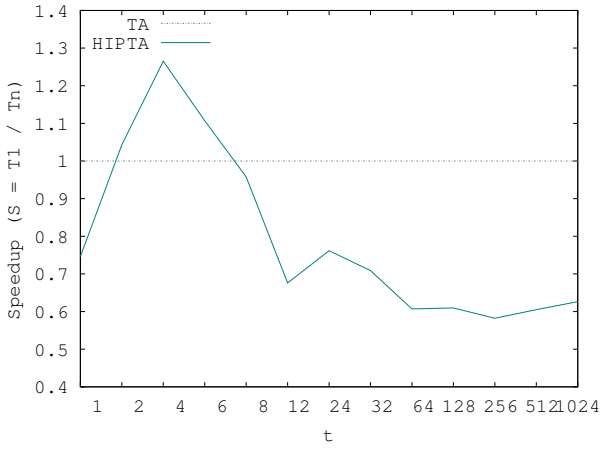
For the positively correlated input distribution, the graph is completely different. There are no speedup until 512 threads are used, where it jumps up to about 1.25, only 0.0025% of linear speedup. It is no reason for the algorithm to perform any better for 512 threads than 24, this increase is assumed to be due to a random event. Positive correlation lead to termination in less than one second, and results showed a relatively big variation. For such a quick termination the sequential algorithm seems to perform better than HIPTA. For cases like this it would be helpful to have some heuristic that can be used to decide if TA or HIPTA should be used to answer the query.



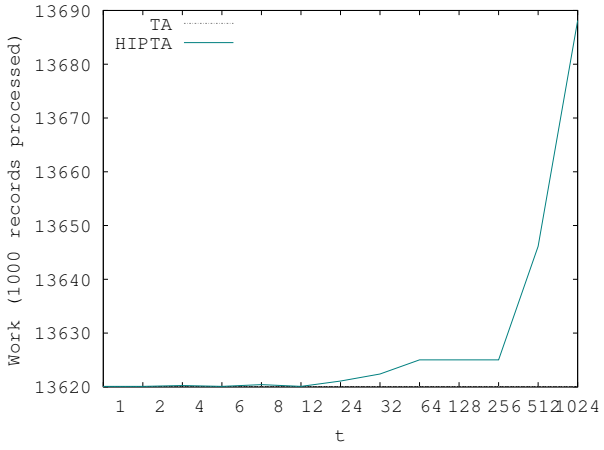
(a) Speedup, uniform



(b) Speedup, positive correlation



(c) Speedup, negative correlation



(d) Work, uniform

Figure 6.6: Test 6, thread count and input distribution

Negative correlation was expected to increase work amount required, and thereby increasing the potential for parallel speedup. However, the test results differ. A potential reason for this is the fact that n was set to 2, lower than for the other tests to simplify the process of generating a negatively correlated data set. Comparing this result to findings in Section 6.3.1, one can see that limiting n has a severe effect on HIPTA. Another factor is the result data structure, it is possible that the optimization described Section 6.1 is ineffective for this data set. If that is the case, the algorithm will have a bigger sequential part, and parallel speedup will be limited. There is a slight speedup for four threads, but most of the graph shows negative speedup, HIPTA seem to be less efficient than TA for this distribution. The assumption of increased work required to produce results were correct, about twice as many records need to be processed before the algorithm can terminate.

6.3.5 Implications

These results imply that it is possible to improve performance with minimal changes, even for synchronization-heavy algorithms like top-k select. However, algorithms most be carefully examined to be able to exploit shared-memory systems in an efficient manner. Speedup achieved in this report was not optimal, even with a low number of threads, and the algorithm was sensitive to skewed data sets. By examining the algorithm for different input sets, one may be able to construct useful heuristics. These heuristics could be used by the query optimizer to decide whether the parallel algorithm is likely to be more efficient than the sequential. For instance, the sequential algorithm is superior when processing small data sets and data where attributes shows a high degree of positive correlation. In contrast, for a big data set with uniform distribution, HIPTA

achieves linear speedup using two threads and a speedup of 3 using four threads.

Another implication is that methods from traditional parallel database systems could prove useful, also for shared-memory programming. HIPTA was able to implicitly use horizontal partitioning in a round robin fashion. Although round robin placement is considered a bad partitioning technique in parallel databases, because of the insertion operator, it can be efficient in shared-memory algorithms. In HIPTA, data is not explicitly partitioned, however, tasks are distributed among threads to achieve the same result. An advantage of this method is that data does not have to be moved or copied, each thread are working on a different part of a big array. If this array can be kept unchanged, there will be no synchronization costs while fetching rows, and good performance can be achieved.

The data structure for storing results was identified as a bottleneck. This points to the importance of efficient concurrent data structures. In many cases, a hash map would be considered as a good alternative. However, for top-k select sorted output is required, therefore a concurrent binary tree, or a skip list may be more efficient

Chapter 7

Conclusions and future work

Recently, multi-core processors have become commonplace. Even laptops are equipped with processors including four cores or more. This has led to an increasing interest in shared memory programming. There is a need for efficient algorithms and data structures for shared memory systems with multi-core processors.

In this report, data operations in general were described, including terminology, common methods, and algorithms. Additionally, the multi-core architecture was introduced and a set of frameworks for shared memory programming were described. Algorithms for shared memory systems were explored, including hash join and sort-merge join, and a new algorithm for top-k select were developed by parallelizing the threshold algorithm, first introduced by Nepal et al. in [13].

HIPTA, the parallel threshold algorithm was implemented and evaluated by running several benchmarks. Testing showed that the algorithm was able to utilize the additional parallel compute power to a certain degree using a simple horizontal partitioning technique. The algorithm achieved near to linear speedup for a big uniform data set, with two threads, further increase in thread count did also result in speedup, but to a lesser degree. This was attributed additional synchronization costs, and the fact that a large part of the algorithm were inherently sequential due to a shared data structure for processed rows, requiring one lock for the entire structure. It was suggested to improve the algorithm by replacing this data structure by a data structure that could handle more fine-grained locking so that a greater part of the algorithm could take use of parallel processing. It was also suggested that database management systems should utilize heuristics to determine if parallel, shared memory algorithms were likely to outperform the well established sequential versions, and thereby utilizing multi-core computing where appropriate.

It would be interesting to test different data structures for the HIPTA algorithm to check if the assumptions made were correct. There are already implementations of concurrent binary trees, skip lists, and similar that should be considered in future versions. It is possible that more fine-grained locking will improve performance, this could also improve the poor performance seen for skewed data sets. Another technique that could be used is to allow each thread to process its partition independently, and only synchronize at the point where a new threshold is calculated. This would allow for bigger parts of the algorithm to be executed in parallel, but would incur an additional merge overhead.

Reported results suggests that multi-core processors can be utilized to improve performance for database operations, even for operations that requires a big amount of synchronization. However, for certain algorithms it may be possible to develop better algorithms using more creative techniques. This points to points to a need for further research on database algorithms and structures for multi-core processors. It is expected that core count will continue to increase, and database management systems must be able to efficiently utilize the additional compute power.

Bibliography

[1] M.-C. Albutiu, A. Kemper, and N. Thomas. Massively parallel sort-merge joins in main memory multi-core database systems. In *Proceedings of VLDB'11*, 2012.

[2] D. Bader, A, V. Kanade, and K. Madduri. SWARM: A parallel programming framework for multicore processors. In *Proceedings of IDPS'07*, 2007.

[3] Berkley unified parallel C (UPC) project. `http://upc.lbl.gov`, 2012.

[4] S. Blanas, Y. Li, and J. M. Patel. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *Proceedings of SIGMOD'11*, 2011.

[5] K. Bratbergsengen. Hashing methods and relational algebra operations. In *Proceedings of VLDB'84*, 1984.

[6] K. Bratbergsengen. *TDT4225 Storing and Management of large Data Volumes.* Tapir akademisk forlag (kompendieforlaget), 2011.

[7] I. Ilyas, F, W. Aref, G, and A. Elmagarmid, K. Supporting top-k join queries in relational databases. In *Proceedings of VLDB'03*, 2003.

[8] I. Ilyas, F, G. Beskales, and M. Soliman, A. A survey of top-k query processing techniques in relational database systems. *ACM Computing Surveys, Vol. 40, No. 4*, 2008.

[9] Intel. *Intel® 64 and IA-32 Architectures Optimization Reference Manual.*

[10] Intel. *Intel® Threading Building Blocks (revision 1.6)*, 2007.

[11] P.-Å. Larson, S. Blanas, C. Diaconu, C. Freedman, J. Patel, M, and M. Zwilling. High-performance concurrency control mechanisms or main-memory databases. In *Proceedings of VLDB'11*, 2011.

[12] R. Membarth, F. Hanning, J. Teich, M. Korner, and E. Wieland. Comparison of parallelization frameworks for shared memory multi-core architectures. Technical report, University of Erlangen-Nuremberg and Siemens Healthcare Sector, 2010.

[13] S. Nepal and M. Ramakrishna. Query processing issues in image (multimedia) databases. In *Proceedings of ICDE'99*, 1999.

[14] *Oracle TimesTen In-Memory Database Overview*, 2012.

[15] P. Pacheco. *An Introduction to Parallel Programming.* Morgan Kaufmann, 2011.

[16] N. Popovici and T. Willhalm. Putting Intel® threading building blocks to work. In *Proceedings of IWMSE'08*, 2011.

[17] R. Ramakrishnan and J. Gehrke. *Database Management Systems.* Mc Graw Hill, 3 edition, 2003.

[18] C. Ranger, R. Raghuraman, A. Penmetsa, G. Bradski, and C. Kozyrakis. Evaluating mapreduce for multi-core and multiprocessor systems. In *Proceedings of HPCA'07*, 2007.

[19] M. Stonebraker. The case for shared nothing. In *Proceedings of HPTS'85*, 1985.

[20] A. Tumeo, S. Secchi, and O. Villa. Designing next-generation massively multithreaded architectures for irregular applications. *Computer*, pages 53 –61, 2012.