Distributed Searching in Biological Databases

by

Dominic Battré, B.Comp.Sc.

Thesis

Presented to the Faculty of the
School of Computer Science,
Telecommunications and Information Systems,
DePaul University
in Partial Fulfillment
of the Requirements
for the Degree of

Master of Science

DePaul University
October 2005

Distributed Searching in Biological Databases

APPROVED BY
SUPERVISING COMMITTEE:

David Angulo

Massimo DiPierro

Ljubomir Perkovic

Acknowledgement

First, I want to thank my supervisor, David Angulo of DePaul University, for providing me with the support and the space to pursue my ideas and interests throughout my whole course of studies.

Furthermore, I want to express my gratitude to the Paderborn Center for Parallel Computing, esp. Prof. Dr. Odej Kao and Axel Keller, for providing me access to a 96 node cluster and thereby facilitating this research. The help Dr. Rajeev Thakur of the Argonne National Lab has provided on the new multi–threading support of MPICH 2 was very valuable. I am thankful for ideas and comments of the researchers of the Illinois Biogrid, Dr. Gregor von Laszewski, and Dr. Nicholas Karonis of the Argonne National Lab.

I want to thank the Fulbright Commission and the German National Academic Foundation, which supported and facilitated my studies at DePaul University with scholarships.

Finally, and foremost, I want to thank my family for their love and constant support.

Chicago, IL, USA, October 2005 Dominic Battré

Distributed Searching in Biological Databases

by

Dominic Battré, M.S.
DePaul University
SUPERVISOR: David Angulo

This thesis addresses the problem of searching huge biological databases on the scale of several gigabytes by utilizing parallel processing. Biological databases storing DNA sequences, protein sequences, or mass spectra are growing exponentially. Searches through these databases consume exponentially growing computational resources as well. The thesis demonstrates and analyzes a general use, MPI based, C++ framework for generically splitting databases amongst several computational nodes. The combined RAM of the nodes working in tandem is often sufficient to keep the entire database in memory, and therefore to search it efficiently without paging to disk. The framework runs as a persistent service, processing all submitted queries. This allows for query reordering and better utilization of the memory. Thereby, we achieve superlinear speedups compared to single processor implementations. We demonstrate the utility and speedup of the framework using a real biological database and an actual searching algorithm for Mass Spectrometry.

Contents

List of Figures

List of Tables

1 Problem description

This thesis addresses the problem of parallelizing sequence database searches by proposing and analyzing a framework that supports developers with this task. The framework provides a generic library that can be extended by application domain specific database plugins.

This chapter describes the problem of sequence database searches and explains the master/worker paradigm as one possible approach to tackle the problem. Throughout the chapter, the problem should get more and more concrete, such that we can conclude with the requirements for the framework to be developed in the end.

1.1 Applications

Sequence database searches cover a wide area of applications biologists use for their daily work where large databases need to be searched for sequences they have gained from experiments. To illustrate the process, we can consider the sequences for now as DNA sequences, i.e. strings over the alphabet $\Sigma_{\text{DNA}} = \{A, C, G, T\}$.

Finding similar sequences helps biologists in many ways. First, it helps to determine, whether the sequences they are investigating have been identified and analyzed before. But even if a gene has not been identified before, finding similar genes helps to determine its function in the organism, because often similar genes code for proteins with similar functions. If we find a sequence in a new species that is similar to the sequence of hemoglobin in the human body for example, it is likely that this new sequence codes for proteins that are responsible for oxygen–transport in the red cells of the blood as well.

Algorithm 1.1.1 gives pseudocode for a trivial sequence database search algorithm as it is implemented in many single processor implementations of BLAST [5, 6] or FASTA [42, 54] today. As it is not easy to calculate the similarity between two sequences, one cannot use hashing mechanisms as they are employed in text indexing search engines

ALGORITHM 1.1.1. Trivial Sequence Database Search.

Input: Database $\mathbf{D}$, search sequence s
Output: Most similar sequence $d_s \in \mathbf{D}$ to s.

1. $d_s \leftarrow$ dummy sequence
2. For all $d \in \mathbf{D}$ do 3–5
3. If d is more similar to s than d_s then
4. $d_s \leftarrow d$
5. return d_s

like ht://Dig [32]. Instead of that, one usually has to iterate over all records of the database, compare each with the query sequence, and memorize the best match.

The type of queries we are interested in explains the structure of most biological databases. Different from relational databases, biological databases are usually stored as flat text–based files. Figure 1.1.1 illustrates how sequence data can look like in the FASTA format [25]. Each entry starts with a description line, consisting of the > symbol, an entry ID, and a description of the entry. This line is followed by the sequence data. This example describes a sequence of amino acids. Database formats are usually devised, so that it is easy to parse the files, but toolkits like NCBI Blast [49] often create a preformatted binary copy of the database to allow for faster reading.

The databases we are interested in have in common that they are stored in flat files and consist of an immense number of records. The following three sections illustrate the contents of DNA, protein, and mass spectral databases, and the measures that can be used to determine the similarity of database entries. This illustration attempts by

```
>Example1 envelope protein
ELRLRYCAPAGFALLKCNDADYDGFKTNCSNVSVVHCTNLMNTTVTTGLLLNGSYSENRT
QIWQKHRTSNDSALILLNKHYNLTVTCKRPGNKTVLPVTIMAGLVFHSQKYNLRLRQAWC
HFPSNWKGAWKEVKEEIVNLPKERYRGTNDPKRIFFQRQWGDPETANLWFNCHGEFFYCK
MDWFLNYLNNLTVDADHNECKNTSGTKSGNKRAPGPCVQRTYVACHIRSVIIWLETISKK
TYAPPREGHLECTSTVTGMTVELNYIPKNRTNVTLSPQIESIWAAELDRYKLVEITPIGF
APTEVRRYTGGHERQKRVPFVXXXXXXXXXXXXXXXXXXXXXXVQSQHLLAGILQQQKNL
LAAVEAQQQMLKLTIWGVK
>Example2 synthetic peptide
HITREPLKHIPKERYRGTNDTLSPQIESIWAAELDRYKLVKTNCSNVS
```

Figure 1.1.1: FASTA Format [25]

no means to give a comprehensive description of the biological background. Instead it shall serve as an overview to help understanding the motivation of the thesis.

1.1.1 DNA databases

As mentioned before, DNA databases store genes, which are encoded as sequences of the four nucleotides adenine (`A`), cytosine (`C`), guanine (`G`), thymine (`T`). Each record of the database consists of the nucleotide sequence itself plus meta data like primary IDs and annotations. The genes of the human genome have an average length of 10–15 kb (kilo base pairs), but their length varies enormously, ranging from less than 1 kb up to almost 2500 kb [66].

In order to determine the similarity of two sequences, we need to align them to each other. Two important algorithms for determining optimal alignments are Needleman–Wunsch [50] and Smith–Waterman [63]. They are basically slight modifications of the well known algorithm for calculating the minimum edit distance of strings. The smaller the edit distance is, the more similar are the sequences aligned.

This is an example of an alignment for the sequences $S_1 =$ `ACCTATAGATGGAATA` and $S_2 =$ `ACTACAGATGCACCCACGAACCA`:

```
ACCTATAGATG-------GAA-TA
||  ||  |||||         ||  |
AC-TACAGATGCACCCACGAACCA
```

We see that the alignment consists of matching nucleotides, mismatches and gaps. The before mentioned algorithms calculate optimal alignments with regard to given penalties for mismatches and gaps in runtime $O(|S_1| \cdot |S_2|)$. BLAST [5, 6] is a heuristic that attempts to improve the runtime at the cost of not delivering optimal alignments. As current databases are huge, BLAST is often preferred to the optimal alignment algorithms.

The probably most famous databases of non–redundant nucleotides can be found on the NCBI [48] and EMBL–EBI [23] servers. As of writing this thesis, the EMBL database has a collection of roughly 51,000,000 DNA sequences containing in total roughly 88,000,000,000 nucleotides and experiences an exponential growth [24]. Queries usually search only subsets of the whole database, but considering the growth of data,

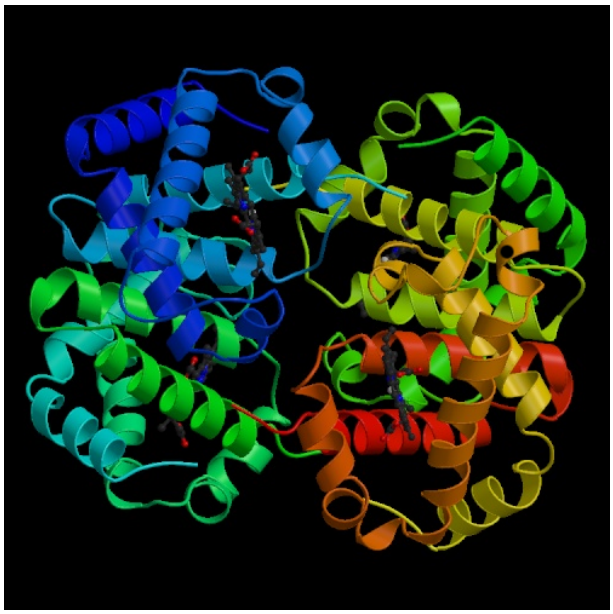


Figure 1.1.2: Tertiary structure of a protein

it is crucial to perform queries in a parallel manner in the future. Cameron, et al. found that the GenBank database growth exceeded the performance gains of new hardware during the years 1999–2003, such that an exhaustive search using BLAST became approximately 64% slower each year [16].

1.1.2 Protein databases

While DNA databases store only linear sequences of nucleotides, protein databases store often many different aspects of proteins. These include the protein's primary, secondary and tertiary structure.

A protein is a chain (polypeptide) of many amino acids. This chain can be modeled as a string over the alphabet AA, where AA represents the set of all 20 amino acids. The sequence of amino acids, i.e. the string mentioned above, is called its primary structure. As sequence strings of the amino acids and sequence strings of nucleotides are very similar apart from the different alphabet, one can use similar algorithms to the ones described above for calculating the similarity of amino acid sequences.

Different from DNA, amino acid chains do not bind to an inverted chain in order to build a helix, but fold up to complex structures. Structures often encountered are e.g.

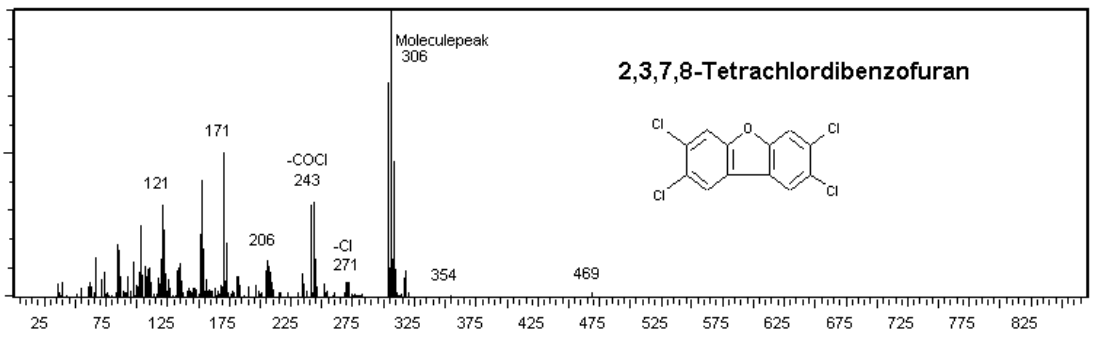


Figure 1.1.3: A mass spectrum [44]

α–helices and β–sheets. These structures describe the secondary structure of proteins. The tertiary structure maps each amino acid of the protein to coordinates in space and thereby describes the three–dimensional structure. Figure 1.1.2 contains an image of a protein's tertiary structure.

As mentioned before, comparing the primary structures of proteins is very similar to comparing genes. Comparing three–dimensional structures, however, is very different. Algorithms like the Root Mean Square Distance (RMSD) [59] and the Unit–vector RMS (URMS) Distance [18] solve this task by mapping the similarity to a real number. These algorithms have in common with sequence database searches that they need to compare each pair of a database sequence and the query sequence in order to find the best matches. Therefore, the problem structure remains the same as in Algorithm 1.1.1.

The RCSB PDB (Protein Data Bank) [60] is an example for a worldwide repository for 3–D biological macromolecular structure data.

Owing to post–translational modifications, the proteome of a species, i.e. the set of proteins expressed in its cells, is several orders of magnitude bigger than the genome of a species. This makes database searches on proteomic data even more expensive than on genomic data in the future.

1.1.3 Mass spectrometry databases

In order to determine the amino acid sequences of proteins, these are often broken into smaller parts (peptides), whose weight can be measured by mass spectrometers. The output of a mass spectrometer is a mass spectrum as depicted in Figure 1.1.3. After breaking the protein (2,3,7,8–Tetrachloridbenzofuran) into smaller molecules, these

molecules were charged and weighted. The horizontal axis describes a peak's ratio of molecule mass–to–charge. The height of a peak describes the relative abundance of molecules of this particular mass–to–charge ratio.

To compare mass spectra, one can group peaks into bins of fixed width by adding the peak values that fall into a bin. A bin i could span the range $[i \cdot 5; (i+1) \cdot 5)$. After calculating the bins for two mass spectra, we can consider the sequence of bin values as a multidimensional vector and calculate the cosine between the vectors with the cross product according to the formula

$$\cos\theta = \frac{\sum_i V_i S_i}{|V| \cdot |S|} \tag{1.1.2}$$

where V and S are the vectors of the bin values. Similar mass spectra will have similar vectors of bins and therefore a $\cos\theta$ value which is close to 1.

Again, we have an algorithm to map the similarity between two sequences to a real number, but need to compare all database sequence to the query sequence one by one in order to find the most similar sequences in the database.

The PeptideAtlas [55] is a representative of this category of databases, containing more than 350,000 MS/MS spectra comprising more than 32 GB of compressed data. The IBG Mass Spectrometry Database [33] is going to belong to this group of databases as well.

1.1.4 Conclusion

We have shown three different database domains which all require a comparison of a query sequence against each sequence in the databases in order to find the best matches. Each database on its own is such big that it is necessary to distribute searches to several nodes in order to get results in a timely manner.

This thesis analyzes many aspects of parallelizing database sequence searches and describes a framework, which can help for many different instances of this problem, including the ones explained above.

The upcoming chapter describes the master/worker paradigm, which we utilize to parallelize database sequence searches.

ALGORITHM 1.2.1. Master/Worker scheme for the server.

Input: Problem Q
Output: Solution S

1. break problem Q into m pieces $Q_1, \ldots, Q_n$
2. For $i = 1 \ldots n$ do 3–4
3. wait for idle worker and assign subproblem i to it
4. collect results of workers, if available
5. wait for and collect remaining results from workers
6. combine results to solution S for the whole problem Q

ALGORITHM 1.2.2. Master/Worker schema for the client.

1. While true do 2–4
2. wait for subproblem Q_i
3. solve subproblem Q_i
4. send results to master

1.2 The Master/Worker Paradigm

The main idea of the master/worker paradigm is to break a problem into many independent and conceptually identical subproblems, which can be solved on many computers in parallel. After that, the partial results can be combined into a final result for the original problem.

Algorithm 1.2.1 and 1.2.2 give pseudo codes for this approach. We partition the set of processing nodes into one master M and many workers $W_1, \ldots, W_p$. In order to solve a problem Q, the master splits this problem into many subproblems $Q_1, \ldots, Q_n$ (often $n \gg p$), which it keeps in a work pool. Whenever a worker has no remaining jobs, the master node assigns it one of the subproblems from the work pool. The worker processes this and sends the result back to the master node. Once all subproblems are processed, the master computes the result of the whole problem Q.

1.2.1 Analysis of speedups

Suppose all tasks have the same complexity and each one can be solved in time t. The total time to process all n subproblems on a single processor is

$$T_1 = nt. \tag{1.2.3}$$

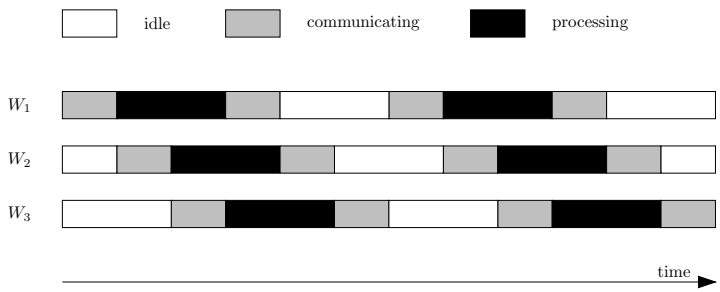


Figure 1.2.1: Idling periods due to long send operations

If one distributes the work between p workers, each worker has to process $\frac{n}{p}$ subproblems of size t. Furthermore, one needs to send the subproblems to the workers and receive their results. Let the communication time be Δ per task. If $\Delta \ll t$, and in particular $p\Delta < t$, we can assume that most communication happens while all other nodes are busy, and approximate the time to process all n subproblems on p workers with

$$T_p = \frac{n}{p} \cdot (t + \Delta). \tag{1.2.4}$$

The speedup, which is defined as $S = \frac{T_1}{T_p}$, is in this case

$$S = p \cdot \frac{t}{t + \Delta}. \tag{1.2.5}$$

The master node can keep only $\frac{t}{\Delta}$ workers busy: If $p > \frac{t}{\Delta}$, we see that $t < p\Delta$. This means that a worker processes tasks faster than it receives them (see Figure 1.2.1). Therefore, it will spend a lot of time idling. For the remainder of this chapter, we assume that this inequation holds, i.e. the master is not the bottle neck.

We can improve our estimation of T_p by incorporating the fact that not all workers can start immediately but shifted (see Figure 1.2.2), and that they need to share the network. Assuming this, we need to add up to $(p-1)\Delta$ to T_p which becomes

$$T'_p = \frac{n}{p} \cdot (t + \Delta) + (p-1)\Delta. \tag{1.2.6}$$

However, this is negligible, unless we have few but big tasks and many processors.

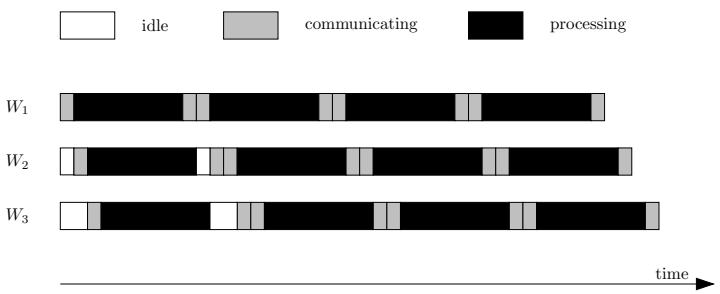


Figure 1.2.2: Idling periods due to short send operations

The efficiency E is defined as the speedup per processor,

$$E = \frac{S_p}{p}. \tag{1.2.7}$$

In case of the simplified speedup formula (1.2.5), we get the following efficiency

$$E = \frac{t}{t + \Delta}, \tag{1.2.8}$$

which holds unless the number of processors is very big or sending a task to a worker takes very long time. Figure 1.2.3 depicts the efficiency for various ratios of Δ to t. As we can see, the efficiency is greater than 90% as long as $\Delta < 0.1t$.

We can decrease the $\frac{\Delta}{t}$ ratio by creating fewer but computationally more expensive tasks (literature speaks of coarse grained instead of fine grained tasks) and assigning several tasks to a worker at once (literature speaks of chunk scheduling). Both methods reduce the ratio of communication to computation, but introduce load imbalances on the other hand. Figure 1.2.4 illustrates this problem in the extreme case, where each processor is assigned only a single job without enforcement that all jobs have equal sizes. Programs like mpiBLAST [46] use this approach because merging results takes a lot of time. The authors of [64] report average runtime reductions of 69% for some cases of heterogeneous hardware by minimizing load imbalances.

As we can see, we need to find a balance between fine and coarse grained tasks, because either extreme position has negative influence on the total runtime.

Other ways to improve the efficiency include the introduction of multi–threading, such that workers send results in the background while processing the next subtask and the idea, that workers can prefetch subtasks in the background, so that they never

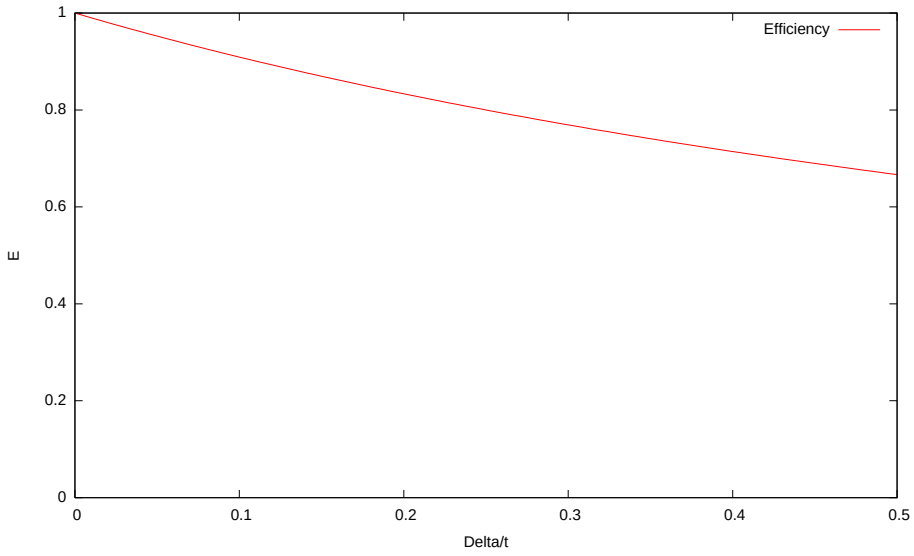


Figure 1.2.3: Efficiency for various ratios $\frac{\Delta}{t}$.

Figure 1.2.4: Long idling periods due to different problem sizes

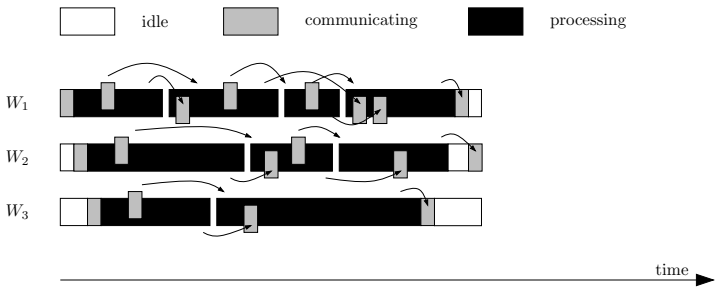


Figure 1.2.5: Network communication in the background

need to wait for the reception of their next work units. Figure 1.2.5 illustrates these strategies.

As indicated before, master/worker algorithms perform pretty well, if the number of tasks is bigger than the number of processors. Even if some tasks are bigger than others or if some nodes are slower than others, the dynamic scheduling distributes tasks in such a way, that all nodes finish their work at approximately the same time. In many cases, the ratio of communication and work is pretty low, and we get a good utilization of resources. Our approach will look slightly different, but will use many ideas of the master/worker pattern as described so far.

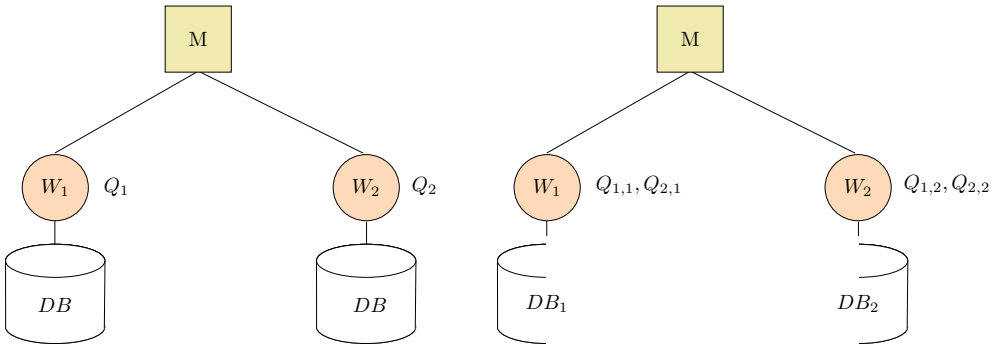
The following chapter illustrates how we can apply the master/worker paradigm to parallel sequence database searches.

1.2.2 Parallelizing sequence database searches

Current biological databases [23, 48, 60] offer web interfaces for scientists to submit queries and to view the results. The servers have to search huge amounts of data, but need to handle many concurrent queries as well.

There are two main approaches to handle the load:

1. *Query dispatching:* A server can simply forward queries it receives to the next idling worker. This worker processes the query and reports the results back to the server, who generates the output for the user. If no workers are idle, the server can queue the query until it can find free resources.
   Figure 1.2.6a illustrates the situation where two workers W_1 and W_2 work on queries Q_1 and Q_2 respectively. Both workers have a copy of the database against which the queries are compared.



(a) Coarse grained tasks (b) Fine grained tasks

Figure 1.2.6: Distributing queries (Q_1, Q_2) from a master M to workers (W_1, W_2)

This approach is used in SOAP–HT–BLAST [70] for example. It follows the master/worker paradigm as it was described above.

2. *Query breaking:* Instead of assigning one query to one worker, a query can be broken into subqueries against different parts of the database, one for each worker. Figure 1.2.6b illustrates this for two workers. The queries are split such that $Q_i = Q_{i,1} \cup Q_{i,2}$ and the parts are distributed between the workers. The advantage of this approach is that each worker needs to know only a part of the whole database. Therefore, the worker saves time reading the database from disk. Often it is possible that each worker can keep its database fragment in memory without paging to hard disk while processing a series of queries. This reduces the run time enormously as we will see in the analysis of the framework's performance. The mpiBLAST [46] and pioBLAST [41] implementations follow this approach of splitting the database to nodes, where each node processes several queries on their database fraction. The motivation for this is the reduction in network communication (each node needs to download only a part of the whole database) and thrashing (the database needs to fit only in the *combined* RAM).

Current parallel BLAST implementations [46, 41, 70] use the master/worker paradigm with great success. However, we believe that we can increase the efficiency even more if the input comes as a stream of queries that arrive at arbitrary times by implementing the BLAST server as a persistent service.

The first factor of improvement is the *application startup time* which can contribute a significant amount of time if the actual processing of a single query takes only fractions of a second. Implementations like SOAP–HT–BLAST [70] start a distributed search

whenever a certain number of queries has arrived or a certain amount of time has passed. This results in many application startups.

Another factor in connection with application startup time is that of *reading the database*. If the application runs persistently, the database needs to be loaded into memory only once and not once for each query. In case of grid systems, this is even more important, as data co–location consumes a lot of time if the databases are big and need to be read or downloaded from a shared disk.

If the database is distributed between many nodes, so that each node is responsible for a fraction of the database, it is often possible to keep this fraction in memory. This eliminates thrashing and allows for super linear speedup with the number of nodes.

If the application does not terminate after each batch of queries, it can *carry over information about the speed of nodes* to the next query. This allows for iterative refinements of the nodes' responsibilities, such that faster nodes are responsible for bigger parts of the database.

If the application keeps running persistently, it is furthermore easier to change the order in which queries are processed. Changing this order can improve the utilization of caches.

And finally, while gathering and formatting results from the workers, all but one processor idle. If the workers run as a service, these processors can continue working on the next queries, while one node formats the results.

1.3 Goals

The goal of this thesis is to implement a *framework* that allows for efficient sequence database searches on many nodes. By implementing a *persistent service* which accepts a *stream of queries*, the framework can *exploit data locality with the augmented combined RAM* of many nodes, which we expect to result in super–linear speedups.

The following paragraphs give more details on this goal:

This *framework* shall provide a simple to use API, which separates the problem independent (common among all types of database searches as seen above) and a problem dependent part. This allows developers to integrate their search methods (problem dependent part) easily into the framework.

The Message Passing Interface [68] shall be used to provide high performance network communication. MPI has been used in several well–known projects [46, 47, 65]

with great success. Existing task farmers use JavaRMI [1], HTTP [15], self–written TCP solutions [15], MPI [47, 65] and other protocols. Because MPI has vendor optimized drivers that circumvent the TCP/IP protocol stack if possible (on multi–processor shared memory machines) and allow for using special low latency network hardware, we believe that MPI is much more efficient for inter processor communication than JavaRMI. Further is hides machine dependent data representations, which makes it easier to use than a self–implemented communication with TCP sockets.

In order to allow for a processing of a *stream of queries* we need an easy means of query submissions. This can be provided by Grid methodologies like SOAP. A SOAP interface allows applications of all kinds (stand–alone desktop applications, web applications, and others) written in almost any programming language to submit queries and request results easily.

The framework shall provide a *persistent service architecture* in order to *exploit data locality*. If workers run permanently, the master can assign them parts of the database statically and ask them to keep these parts in memory. After that, it can assign subqueries to the workers, which have the respective database fractions in memory, i.e. it exploits the location of data. Even if we allow dynamic, changing assignments of responsibilities for database parts, we can come up with schedules, which assign queries to those nodes, which have the necessary data in memory. The persistent service makes it relatively easy to rearrange the order in which queries are executed, in order to speed up processing times. However, it is important to stick relatively close to FIFO order in order to ensure fairness. This is important because the framework shall be employed as a search engine on a public website.

The thesis will contain a sample application that consists of a distributed search backend as well as a web frontend for demonstration and benchmarking purposes.

1.4 Organization of the thesis

After stating the objective of the thesis, chapter 2 continues with an overview of related research. Chapter 3 discusses several alternative approaches and design decisions for the development of the framework. Chapter 4 explains the implementation and gives an example of how to use the framework for searching similar mass spectra. Chapter 5 analyzes of the framework's performance, measured with the application described in chapter 4. Finally, chapter 6 concludes the thesis.

2 Related research

The purpose of this chapter is to illustrate and evaluate the approaches of existing implementations in two areas: We will consider generic frameworks for master/worker applications as well as specialized applications for parallel BLAST searches, as this is a prominent field, where researchers have competed for efficiency for some time now.

2.1 Frameworks

The analysis of frameworks begins with Condor MW [21], as this is one of the simplest master/worker frameworks available, and because it implements relatively closely the concepts discussed in the previous section.

2.1.1 Condor MW

Condor MW [21] is a master/worker framework that builds upon the Condor High Throughput Computing Project [20] (abbreviated Condor) of the University of Wisconsin–Madison. As the Condor environment motivates, facilitates, and restricts, what Condor MW can provide, it is appropriate to start this chapter with a brief introduction to Condor.

The Condor project

The authors of [39] describe Condor as being *"[...] essentially an idle cycle scavenger [...] [which] resides on a set of workstations (also called a Condor pool) observing their characteristics like load average. If it finds machines idle for a considerable period of time it starts jobs submitted to the system on them."* [39]

The focus on idling desktop computers places Condor into a completely different environment than other toolkits which target clusters of computers. For Condor it is important that computers can join and leave the network, and that processes can be stopped and migrated to other computers if a user needs the resources of a computer where a job was started initially.

A comprehensive list of Condor's features can be found in [67]. It provides *distributed job submissions* such that users do not depend on a central submission server which might turn into a bottle–neck in large scale or distributed Condor pools. *Jobs* and *users* can be assigned *priorities*, to allow for jobs which run in the background while other jobs are prioritized. For that *user authentication and authorization* are necessary. The user can set up *dependencies* between the tasks so that complete workflows can be modeled, and jobs can be bound to run on computers which provide the *resources requested* for the jobs. Condor supports both serial jobs and parallel jobs, which run with PVM or MPI. Because Condor targets desktop computers, it needs to support *heterogeneous systems* with *job checkpointing and migration.*

The remainder of this chapter describes the task and workflow model that is used by Condor MW and concludes with some interesting implementation details.

Task and workflow model

Condor MW builds a network of one master and p workers, similar to the network described before. Owing to Condor's background as a desktop grid, the number of workers is not constant. On the one hand, workers can leave the network, and on the other hand, the master node can request more workers from the Condor pool if desired.

Tasks are basically serializable classes which can be passed between the master and workers. Condor MW does not restrict the information a task consists of but assumes that there are no interdependences between tasks. The user can implement sorting criteria for tasks in order to assign priorities. The master node maintains a priority queue of tasks and distributes the tasks to the workers.

The work flow of Condor MW is very simple but at the same time very powerful in terms of modeling many different types of applications. At the very beginning the master reads a problem description and generates subtasks, which are stored in the priority queue already mentioned. From then on, the master node basically reacts on two types of events: If a worker asks for a work unit, the master assigns the first task of the priority queue to this worker. If a worker submits the results of a work unit, the master is basically free to do, whatever the programmer desires. As the results of all work units need to be aggregated eventually, the master can process results as soon as they arrive or store them for later aggregation until all results have arrived. Furthermore, the master can use the information of the work unit result to create new work units. This way, even genetic algorithms or processes of many parallel steps can be modeled. Figure 2.1.1 illustrates the workflow.

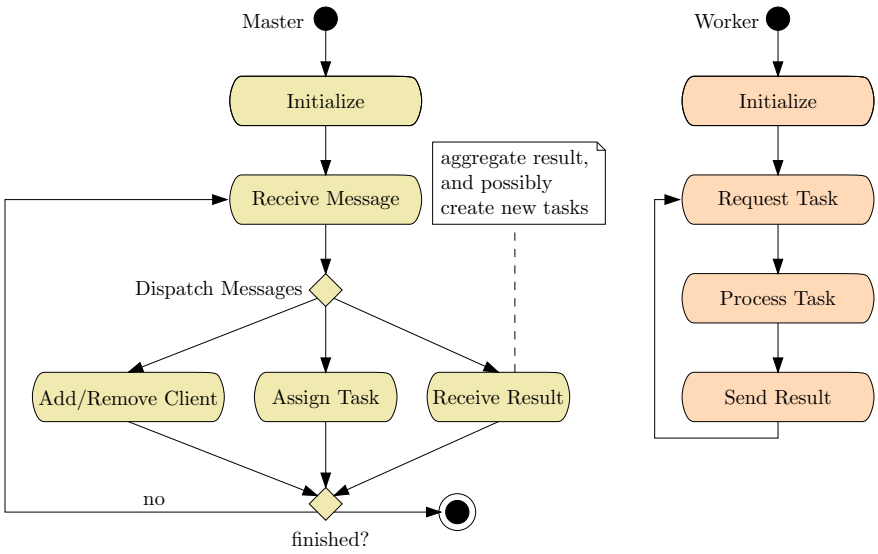


Figure 2.1.1: The Condor MW schema

Condor MW comes with a feature called group management. Workers and tasks can be assigned to certain groups. The scheduler uses this information to distribute tasks only to those workers, who share at least one group membership with the tasks. Memberships can be based on CPU speed, memory size, and availability of data. The master can pass data to the workers when they join the network. The paper [39] shows how this can reduce the communication overhead for distributed matrix multiplication. The group management facilitates a very basic support for data–locality.

Programming Model

From a programmer's perspective it is very simple to implement master/worker applications with Condor MW, as one needs to extend only tree classes.

MWDriver The `MWDriver` represents the process of the master node. The user has to implement functions which read the problem description and generate an initial set of tasks. Further one can send some initial data to the workers if desired. After this initialization phase, the only remaining function that needs to be implemented is an event handler that gets called upon the completion of tasks. The user can aggregate the results or create new tasks. Everything else needed for the task distribution is provided by Condor MW; this includes workers joining and leaving the network, the assignment of tasks, and reassignment of tasks if

workers die. The framework offers stubs for checkpointing, but these need to be implemented by the user if desired.

MWTask The `MWTask` class represents the tasks and is basically a serializable data container. The user needs to implement four functions to pack and unpack the task description and the results. Condor MW provides the means to send tasks to the workers.

MWWorker The `MWWorker` class represents the thread that is started on the worker side. It needs to implement a function that accepts an `MWTask`, solves the problem which is described by the task object, and finally writes the result back into the `MWTask`. Further, the `MWWorker` can provide a function to unpack initial data from the master during the initialization phase.

Condor MW adds a layer of abstraction over the communication and thereby offers three different means of communication between the master and the workers, which can by interchanged easily. These are:

- PVM [58],
- Condor remote I/O for shared files, and
- TCP Sockets.

The PVM library supports development form the view point of a parallel virtual machine. The paper "PVM and MPI are completely different" [29] compares the two approaches. The authors of Condor MW had to choose PVM instead of MPI because it provided dynamic process management (processes joining and leaving the network), a feature that is not defined in MPI 1.x.

The authors of [39] recommend using their TCP Sockets implementation. PVM is difficult to support in the condor world of desktop computers, because it needs to be installed on each computer. The Condor remote I/O methods are said to be relatively slow. Therefore, TCP sockets are a reasonable compromise. Developers using the TCP sockets need to be careful with portability issues, however. Floating point numbers for example are not marshaled correctly by Condor MW to an architecture independent format. Instead, they are simply casted to byte arrays and then sent to other nodes.

Condor MW offers a very convenient feature for debugging. As it is difficult to debug distributed applications with commodity debuggers like `gdb`, Condor MW offers a means to run one master and one worker within a single process. In this case, send and receive commands are simple memory copies and whenever one party sends data

to the other party, the thread of execution changes. This way, it is very easy to use stock debuggers to trace the program execution.

As Condor MW is deployed in a desktop computer environment, it is important that nodes can join and leave the network. If workers leave the network, the jobs assigned to them are simply relocated to other workers. The master node is however critical and cannot be replaced. In order to compensate for master failures, the user can implement functions which write dumps of the master's current work queue and the tasks already finished. With this it is pretty easy to restart the computation if the master node crashes.

Another interesting feature is the possibility for the master to determine how many workers it needs at runtime. It can request more workers if needed and release workers if they are not needed.

Conclusion

We can conclude the description of Condor MW by stating that Condor MW is a C++ based framework for master/worker applications with simple topologies (one master, many nodes). It does not support a persistent service mode but is started for each problem independently. Condor MW supports group mechanisms to bind tasks to certain nodes, but these mechanisms are insufficient for our application of distributed sequence database searches.

2.1.2 AMWAT

As in the previous section, we want to begin our analysis of AMWAT [11] with a description of its environment, in this case AppLeS [12].

AppLeS

Application–Level Schedulers pursue an approach very different from Condor. The *"AppLeS [framework] essentially develops a customized scheduler for each application"* [12]. These schedulers are superior to stock schedulers, because they integrate application–specific and system–specific information into the resource selection and provide good schedules within the application. AppLeS does not allocate any resources, but writes schedules that can be implemented with external resource management systems.

In order to calculate good or even optimal schedules, AppLeS uses application templates. These templates provide frameworks for certain types of applications like Jacobi

solvers (solvers for systems of differential equations), master/worker applications, or genetic algorithms. The template used determines the general workflow, how a problem is solved. Thereby, each template defines a parameterized model as a basis for estimating the execution time of an algorithm for a given problem instance.

Let's consider the example of the master/worker template. Obviously, the application runtime depends on several system–specific variables like CPU speed, network latency, and network bandwidth. If these variables are known, it is possible to calculate a first estimation of the application runtime for a given problem instance. The estimation can be improved further by incorporating the dynamic status of the system like the load and availability of resource. This information is provided to AMWAT by the Network Weather Service [51]. It measures the current load of resources and predicts the future load.

With all this data, or reasonable approximations thereof, the application runtime for each possible problem instance can be estimated analytically. A problem can be partitioned in many different ways into subproblems and these can be assigned in many different ways to processors. If we assigned a partition i to a node n, its runtime could be estimated as $T_{i,n} = A_i/P_n + C_{i,n}$, where the parameter A_i represents the problem size, P_n stands for the effective processor speed, considering its utilization, and $C_{i,n}$ denotes the necessary communication time. By convention, the total application runtime is determined by the latest terminating node. For each possible set of nodes and for each possible partition of the job, we can estimate the total runtime and choose the best combination. As the AppLeS knows application specific information, it can regard particular needs, like a high or low ratio of computation to communication demand, and request resources accordingly. Regular schedulers cannot utilize this knowledge.

In order to calculate schedules, AppLeS provides a framework which consists of four components:

1. The *Resource Selector* enumerates several possible resource combinations that can be used for the application's execution. These resources need to respect the minimum resource requirements, the user's access rights, and all other details necessary for a viable schedule.
2. The *Planner* creates a schedule for the given resources.
3. The *Performance Estimator* cooperates with the Planner by analyzing and estimating the runtime of schedules in order to find the best possible schedule.

4. Once the best possible schedule (or a reasonable approximation thereof) has been calculated, it is handed to the *Actuator*, which implements it with the target resource management system.

The paper "Application Level Scheduling of Gene Sequence Comparison on Metacomputers" [64] gives an interesting example of how to improve runtimes with an Application Level Scheduler compared to schedulers which do not incorporate application specific details:

In order to minimize the communication time of an application, developers prefer coarse grained tasks. However, these can produce bad load imbalances, as we have seen in Figure 1.2.4 on page 10. Fine grained tasks prevent load imbalances, but create a bigger communication overhead. A good approach is to use a combination of both strategies. We create and assign coarse grained tasks to workers instantly after application startup, but hold back some fine grained tasks, whose distribution is delayed until the first coarse grained tasks are completed. Their job is to ensure a balanced load among all workers. That way, we can benefit from the advantages of both approaches.

The application needs to determine a good ratio of coarse grained to fine grained tasks to ensure that it produces neither too many nor too few fine grained tasks. In order to calculate that ratio, the AppLeS can use the uncertainty estimations of the predicted network and CPU speeds, which are provided by the Network Weather Service as well. In case of a high uncertainty more small grained tasks might be necessary than in case of a very low uncertainty.

As we see, AppLeS are not only useful to select the right kind of computation resources for our application, but they can be employed for task distribution while the application is running as well. This approach is taken by the AMWAT, Apples Master/Worker Application Template [11] as well.

Task and workflow model

Even though AMWAT addresses the same problem of providing a framework for master/worker applications as Condor MW, its task and workflow model is very different. A comprehensive description of AMWAT's models can be found in [61].

For AMWAT, a problem that needs to be solved can be split into one or many work cycles. A work cycle consists of sending data regarding the work cycle to the workers, then splitting the task into many fixed size work units, which get processed at the workers, and finally collecting the results and aggregating them to a (intermediate)

ALGORITHM 2.1.1. AMWAT Schema from the point of view of a master node.

```
 1. initialize
 2. While another work cycle C needed do 3–10
 3.    transfer cycle input data I(C) to all workers
 4.    break problem into many work units
 5.    Until results of all work units w in C received do 6–9
 6.       If received request for work unit then
 7.          transfer work input data i(C, w) to idle worker
 8.       If received result for work unit then
 9.          collect work output data o(C, w) from worker
10.    decide whether problem is solved, or another work cycle is needed
11. finalize
```

result of the work cycle. With this information, the master can decide whether another work cycle is necessary, or whether the problem has been solved. Algorithm 2.1.1 gives pseudo–code for this approach. For many applications, a single work cycle is sufficient, however, in case of simulations, it may be interesting to continue a simulation until a solution converges.

A major difference between AMWAT and Condor MW is AMWAT's hierarchical network topology with more than one master node, which provides sophisticated means for load balancing (see [62]).

A single master node might turn into a bottle neck because of its limited network bandwidth if it has to serve many workers. Instead, a master node can be a parent of other masters which distribute tasks to their children. Figure 2.1.2 illustrates this idea of a multi–tiered hierarchy. For a master node, it does not matter whether its children are other master nodes or workers. The functionality of master nodes just needs to be augmented by the capability to accept tasks from a parent and to report the results back. The information of who is processing the work is totally transparent for the upper most master.

Because AMWAT can use the services offered by the Network Weather Service, it can schedule tasks in a way that keeps workers equally busy. If a master has n tasks and two children, c_1 and c_2, from which c_1 has twice the speed of c_2, it can send $\frac{2}{3}n$ tasks to c_1 and $\frac{1}{3}n$ tasks to c_2 in order to keep both busy for the same time. By not sending all tasks at once but iteratively on request, the load can be balanced even

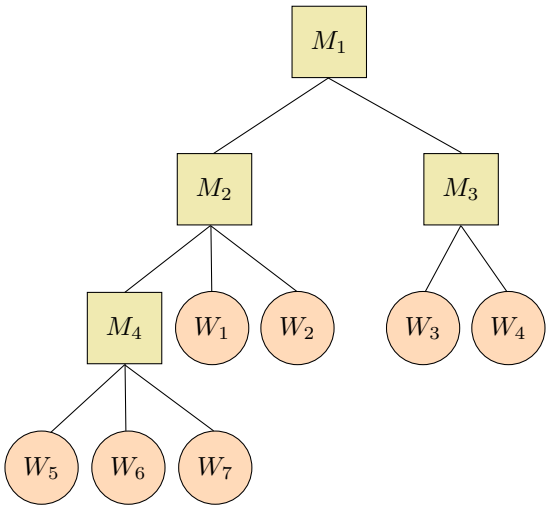


Figure 2.1.2: Multi–tiered master/worker applications

better. Task prefetching, i.e. requesting work before the last work units are finished, diminishes idle time.

The speed of a worker can be estimated by the Network Weather Service based on the performance in the past. The speed of a master with many children can be estimated as the sum of the children's speed. As AMWAT takes even into account that network links may be the limiting speed factor, this approach is well suited for heterogeneous hardware as we can find in a computational grid environment.

Programming model

In order to implement master/worker applications with AMWAT, one needs to implement a small set of C functions for the top master, the intermediate masters and the workers. These are listed in Table 2.1.1.

Tasks are not implemented as serializable classes like in Condor MW. Instead, each task has a unique integer ID. Sending and receiving has to be implemented by the programmer. They are triggered by the framework at synchronized communication points to make sure, that each sending process has a receiving partner and vice versa. This means, if a worker sends a work request message to a master, the framework calls `AMWATApp_SendWorkUnits` on the master node and `AMWAT_ReveiveWorkUnits` on the worker node. These are functions the user has to implement. AMWAT ships several functions that hide specifics of the supported communication technologies (IPC, MPI, PVM and TCP Sockets).

User defined functions
`AMWATApp_Compute(unitId)`
`AMWATApp_FinalizeApp(...)`
`AMWATApp_FinalizeCycle(...)`
`AMWATApp_InitializeApp(...)`
`AMWATApp_InitializeCycle(..., *workUnitsCount)`
`AMWATApp_ReceiveCycleData(whoFrom, ...)`
`AMWATApp_ReceiveResults(whoFrom, *workUnits, workUnitsCount, ...)`
`AMWATApp_ReceiveWorkUnits(whoFrom, *workUnits, workUnitsCount, ...)`
`AMWATApp_SendCycleData(whoTo, ...)`
`AMWATApp_SendResults(whoTo, *workUnits, workUnitsCount, ...)`
`AMWATApp_SendWorkUnits(whoTo, *workUnits, workUnitsCount, ...)`

Table 2.1.1: AMWAT template functions [11]

Conclusion

AMWAT offers an easy to use C based framework for master/worker applications. In steady state, the framework offers an extremely efficient execution of tasks, due to optimized task prefetching with speed estimations. However, synchronization points due to work cycles result in wasted CPU time. The framework does not run as a service and does not support any form of information locality.

2.1.3 Java Distributed System

The task distribution system [1] by the National University of Ireland, Maynooth has no official name [36]. We will use the name "Java Distributed System" (or short: JDS) as it is mentioned in the installation instructions.

The concepts of JDS resemble those we have seen in the two previous frameworks. JDS targets large networks of desktop PCs. The authors argue strongly for using Java to achieve portability [53]. Other advantages of Java are the easy loading of compiled code at runtime, its security features for running code in a sandbox, and the easy high–level remote method invocation.

Task and workflow model

According to [53], JDS employs an n-ary tree of nodes like AMWAT. However, the downloadable environment seems not to support more than one master. Therefore, the following description is solely based on the published paper. Assuming the n-ary

tree model, internal nodes are masters, which are responsible for distributing tasks and aggregating results, while the leaves are workers, which do the actual processing. According to [53] the tree can be rebalanced if nodes join or leave the network. The tree structure allows for scaling to much bigger networks because we do not have the bottleneck of a single master.

Different from AMWAT, the master nodes do not store a list of work units, but generate the work units on demand, whenever clients request more tasks. This has the advantage that the size of work units can be adapted dynamically. Children of a master report the time to process a work unit back to their parent. This calculates an exponentially weighted moving average and adapts the graining of work units so that the average processing time approaches a given value. That way, the user does not need to estimate the runtime of work units for given sizes but passes this job to the framework. As the average is calculated over all nodes, it does not account for different speeds among the nodes. A master whose children are many masters and one worker might overload the worker easily.

Because work units are generated on demand, the user can freely determine when a job has been completed, or, otherwise, keep producing new tasks, even as a reaction to the results received so far. Therefore, the JDS framework allows for the same freedom concerning the task model as Condor MW does.

The major difference from the before mentioned frameworks is that JDS runs as a service. It allows users to submit several jobs which can be processed in parallel, and has an interface to query the status of the jobs' completion. The framework comes with a scheduler that supports fairness criteria and priorities (see [53]), but users can plug in customized schedulers as well.

A job consists of an algorithm in form of executable byte–code and input data. It can be submitted by a user at runtime and will be distributed for processing among the workers. Thereby, the framework can be deployed once and does not need to be modified if users submit new jobs for new types of applications.

The task flow follows the pull model like the before mentioned frameworks. That means, a worker requests tasks instead of getting them assigned from the master automatically. It keeps a buffer of tasks (prefetching) to prevent idle time during network communication. If the master has no tasks, the client sleeps for a certain amount of time and asks again later. This can create delays if the application does not run at its capacity limits.

Programming model

The programming model of the JDS is explained in Thomas Keane's Master's thesis [35] and the developer manual [34]. Java allows it to be much more flexible than any C(++) based implementation because code can be loaded at runtime easily.

The user has to implement two classes for an executable master/worker application.

The `DataManager` runs on the server side and is responsible for generating work units, processing results, and giving status reports. A work unit is a Java `Vector`, which can be serialized and sent over the network by Java means. The user can store into the vector which ever data is suitable for the particular application. All methods that need to be implemented for the server side are `generateWorkUnit()`, `processResult()`, `adjustGranularity()`, `getStatus()` and `closeResources()`. Tasks are stored on disk while being queued in order to reduce memory usage at runtime.

The `Algorithm` class needs to implement only one method, `processUnit()`. It receives a `Vector` parameter for the task description and returns a `Vector` as a result. All communication is hidden from the application developer.

For submitting tasks and monitoring the status, the framework offers a JavaRMI interface.

Conclusion

JDS offers a flexible framework for master/worker applications on desktop PC networks. Its scheduling features are inferior to those of AMWAT but it offers to process many separate jobs from different users in parallel as a service with an external job submission and status query interface.

The authors of JDS quote Bull, et al. [14], who have performed several benchmarks of C, Java and Fortran. According to them, Java is not significantly slower than C for scientific applications. However, Java's RMI implementation is probably much slower than low–level communication via MPI or PVM because it needs to handle more complex mechanisms (classes, synchronization, exception handling) and because it cannot use custom low latency network hardware.

2.1.4 The Organic Grid

"The Organic Grid: Self–Organizing Computation on a Peer–to–Peer Network" [17] shares many concepts with the JDS but tries to employ them in a bigger context like SETI@home with an immense number of desktop PCs. Therefore, simple tree

structures seem not to be efficient enough. The organic grid transfers ideas from peer to peer computing to the problem of master/worker applications.

The organic grid is a sparsely connected network of nodes. Each node tries to keep a fixed number of neighbors to whom it is connected. A user can submit a job to an arbitrary node of the peer to peer network. This node will take the role of a master and find other nodes in its neighborhood to process the job. These nodes can do the actual processing, or, in case the jobs they receive are too big, take the role of a master and acquire other workers themselves. That way, an overlay network with a tree structure is built. Spawning of new masters and workers happens in form of mobile code like it is done at the JDS framework.

Because CPU speeds and network bandwidths vary considerably in peer to peer networks, the overlay tree is restructured at runtime, such that fast nodes move close to the root of the tree. This ensures that the network is no bottle neck and enhances the performance noticeably [17]. Each node is a scheduler for only its immediate children. Therefore, the application scales very well. Once a job is completed, the overlay tree structure will be resolved and the nodes are available for new jobs.

Because different subtrees can have different numbers of CPUs of various speeds, each worker and master keeps a floating average of its speed (and that of its children). With this information, a node can request work from its parent to keep it busy for a given time. This enables task prefetching and a good utilization with load balancing.

As there is no implementation of the organic grid available, we cannot present any programming details.

2.1.5 Other Frameworks

- **Grid Application Framework for Java (GAF4J)**

  IBM's GAF4J [28] is based on Java and the Globus Toolkit [27] in combination with CoG.

  Different from other frameworks, GAF4J does not require the user to implement a certain interface, in order to create tasks that can be executed on other nodes. Instead of that, the user has to write a serializable class and register an instance of this class at a `TaskDefinition` object. Further, one has to specify the minimum resource requirements (CPU speed, free RAM, free disk space) and the input and output files the object is going to use.

Once the `TaskDefinition` object is fully initialized, it can be submitted for execution. A scheduler will use the MDS of the Globus Toolkit to find suitable nodes and processes the tasks in FIFO order. Listeners can be used to collect information about the completion status of tasks.

GAF4J employs Globus technologies to get information about the resources that nodes can provide, and to copy files between nodes. It can be downloaded from IBM's AlphaWorks platform, but its source code is closed.

- **Nimrod/G**

  Nimrod [52] is a platform for parameter sweep applications. These are applications where simulations or calculations have to be repeated with many different parameter combinations for example in order to find averages or optima. Other reasons to compute identical calculations with different parameters are calculations of values at different positions in space.

  Nimrod offers a workflow description language, where the user can describe the possible values of the parameters and necessary copy operations for input and output files.

  Abramson, et al. [2] describe EnFuzion, the commercial sibling of Nimrod, and give a brief explanation of the scheduling. The framework simply assigns jobs in a first–come first–serve order to free processors.

  Nimrod/G [3] is a Globus enabled version of Nimrod.

- **Cactus Grid Application Toolkit**

  The Cactus Grid Application Toolkit [15] offers Cactus thorns (modules) for task farming applications. Tasks are sent and received via HTTP, but extensions for gSOAP [31] and GridFTP [4] are available as well.

- **Berkeley Open Infrastructure for Network Computing**

  The Berkeley Open Infrastructure for Network Computing [10] is the successor of SETI@home [7]. It targets large numbers of desktop PCs and offers a framework for C, C++, and Fortran applications. Users can register and select to which projects they want to donate their computation power. The project offers means for redundant computation to ensure that nobody delivers manipulated results, credit systems to reward the CPU donors, and interfaces to monitor the execution

status. This project clearly targets a different scope of users than the framework to be developed for this thesis.

- **Cilk**

  The Cilk Project [19] pursues a completely different approach than the before mentioned projects. Instead of providing methods for queuing and distributing tasks, Clik is based on a programming language very similar to C which has statements to spawn threads. These threads can be relocated to different processors and processed there with very little work for the programmer. A `spawn` statement forks the program flow and creates two separate executions. The processor will continue processing the first thread until it reaches a join statement and continue working on the second thread afterward. That way, the threads build a binary tree where execution happens in depth–first order. If free CPUs are available, they contribute by executing queued threads. Relocating threads is pretty efficient and based on work stealing. If a processor runs out of jobs, it steals the earliest forked (and therefore often the biggest) not processed thread of a random processor. This ensures that the task is big enough in most cases to justify the overhead of relocation.

2.1.6 Conclusion

We have seen many different frameworks for master/worker applications with various task and programming models. Neither of these frameworks cares for data locality in a way that is suitable for sequence database searches, nor do we know of any framework which supports data locality very well at all.

2.2 Specific implementations

As there seem to be no generic frameworks for sequence database searches, we want to look at some implementations that are specific to BLAST [5].

2.2.1 mpiBLAST

Darling, Carey and Feng describe mpiBLAST [46] as *"an open–source parallelization of BLAST that achieves superlinear speed–up by segmenting a BLAST database and then having each node in a computational cluster search a unique portion of the database"*

[22]. This approach reduces the high overhead of disk I/O, which other single–node implementations of BLAST suffer from, because mpiBLAST can use the combined RAM of all machines and does not need to page to disk. Furthermore, the mpiBLAST approach does not create a big communication overhead because each part of the database can be searched independently from of all other parts.

The input parameters of mpiBLAST are the database and a set of queries. For each query sequence in the input mpiBLAST produces a set of local alignments with scores and a measure of the likelihood that the alignment is a random match between the query and the database (e–value) [22].

In order to perform a distributed query, mpiBLAST needs a segmented database. For that, it offers a wrapper script around the `formatdb` application of the NCBI tools [49], which produces a given number of database segments of equal size.

During the initialization phase, each worker tells the master node, which database segments it finds on its local storage from previous searches. After that, the master broadcasts the set of queries to all workers. This ends the initialization and workers start requesting work units until all units are processed. Each database segment must be compared exactly once by exactly one node against all queries. For the assignment of database segments to the workers, the master uses a simple heuristic. First the master computes the intersection of the database segments the worker has, which requests more work, with the database segments that remain to be searched through. If this set is not empty, the master assigns that database segment, which is replicated on the fewest number of workers. Otherwise it assigns a random non–searched database segment to the worker. In this case, the worker needs to download the database segment first, before it can be processed. After the completion of a fragment search, the worker sends the results to the master, who merges them with the results of other workers and writes the final result to disk when all segments have been processed.

With this algorithm, each database segment needs to be loaded into memory exactly once for a set of queries, and only few segments need to be copied over the network if they remained on the worker nodes from previous searches.

Algorithms Algorithm 2.2.1 and Algorithm 2.2.2 show slightly simplified versions of the pseudo–code given in [22].

The authors of mpiBLAST report superlinear speedups compared to single processor implementations because of the reduction in thrashing effects [22]. However, the speedup deteriorates dramatically if they increase the number of database segments (in-

ALGORITHM 2.2.1. simplified mpiBLAST master algorithm [22].

Let *results* be the current set of BLAST results
Let $F = \{f_1, f_2, \ldots\}$ be the set of database fragments
Let $Unsearched \subseteq F$ be the set of unsearched database fragments
Let $Unassigned \subseteq F$ be the set of unassigned database fragments
Let $W = \{w_1, w_2, \ldots\}$ be the set of participating workers
Let $D_i \subseteq W$ be the set of workers that have fragment f_i on local storage
Let $Distributed = \{D_1, D_2, \ldots\}$ be the set of D for each fragment

```
 1. Unsearched, unassigned ← F
 2. results ← ∅
 3. broadcast queries to workers
 4. While Unsearched ≠ ∅ do 5–14
 5.    receive message from worker w_j
 6.    If message is fragment request then
 7.       find f_i such that min_{D_i ∈ Distributed} |D_i| and f_i ∈ unassigned
 8.       If |D_i| = 0 then
 9.          send f_i to w_j and add w_j to D_i
10.       remove f_i from unassigned
11.       send fragment assignment f_i to worker w_j
12.    If message is search result for fragment f_i then
13.       merge message with results
14.       remove f_i from Unsearched
15. Print results
```

ALGORITHM 2.2.2. simplified mpiBLAST worker algorithm [22].

```
1. queries ← receive queries from master
2. While not search completed do 3–7
3.    currentFragment ← receive fragment assignment from master
4.    If currentFragment not on local storage then
5.       copy currentFragment from shared disk
6.    results ← BLAST(queries, currentFragment)
7.    send results to master
```

dependent of the number of processors), because the master receives a set of matches for each database segment. It has to sort the matches and report only the k best matches. If one splits the database into f fragments, the master needs to receive and analyze $k \cdot f$ results, because the k best global matches might be contained in a single database fragment. Even though $k \cdot f$ results are sent over the wire, only k fragments form the output. The communication overhead increases linearly with the number of database fragments. PioBLAST [41] addresses this issue.

2.2.2 pioBLAST

The pioBLAST [41] implementation attacks the non–search related overhead of mpiBLAST at two points. First, it removes the need for pre–partitioning the database, and second, it optimizes result merging, which contributes a significant proportion of the runtime if the number of processing nodes is big.

For mpiBLAST, the database needs to be partitioned so that each processor gets one partition. That means that the partitioning depends on the number of processors and that changing the number of processors requires a repartitioning. PioBLAST uses the index generated by `updatedb` to assign virtual partitions on the fly. The master node reads the index, and from that, it creates and sends partition information in form of start and end offset in the database file to the workers. These read the database via MPI–IO [45, 30], a parallel I/O interface of MPI–2. In order to use MPI–IO, the authors of pioBLAST had to modify the NCBI Tools slightly.

We have mentioned the weakness of mpiBLAST's result merging. Workers compute alignments with scores and pass the k best of these to the master node. This selects and sorts the k best matches of all workers and formats the output. Contrary to this serial process of mpiBLAST, pioBLAST tries to parallelize the formatting of the global output as much as possible. The actual formatting of output happens at the workers.

With slight modifications of the NCBI Tools the authors of pioBLAST were able to make the workers write the potential formatted output of their database fragment searches into strings in memory. The workers pass the scores of alignments and the string lengths for the formatted outputs to the master, which sorts and selects the best results. With this information it can instruct the workers to write their cached output fragments to the correct offsets in the output file. This is done with MPI–IO as well.

The results are very promising. MpiBLAST spends 95.6% of its total runtime with searching the database on 16 processors. This deteriorates to 10.3% on 61 processors,

while pioBLAST achieves a rate of 92.4% [41]. However, the results were significantly worse with NFS.

Zhu et al. analyzed the performance of mpiBLAST on different file systems with various degrees of data redundancy in [72]. However, this shall not be the focus of this thesis.

2.2.3 Multi–Ring Buffer Management for BLAST

The authors of mpiBLAST and pioBLAST assumed that *"the aggregate memory is large enough to accommodate the database files and intermediate search results"* [41]. This is usually true if several nodes are available for performing the database searches.

Lemos and Lifschitz analyze in "A Study of Multi–Ring Buffer Management for BLAST" [40] the problem of performing many queries on a single machine that cannot store the whole database in memory. They propose using custom page replacement policies that perform better than those of the operating system.

In the buffer management which Lemos and Lifschitz suggest, several threads perform BLAST queries in parallel. The idea of a public–ring is to have one thread, which keeps reading segments from the database and filling the buffer. On the other side, there are p threads, comparing their query sequence against the content of the buffer. When all p threads have searched through a database segment of the buffer, this segment gets released so that the reading thread can replace it with the next database segment.

This approach has two advantages. First, the reading part is done in parallel to processing, and therefore, does not block the CPU from searching through the database. This is easily possible because DMA controllers can perform I/O operations while the CPU does other calculations. Second, the database searches are synchronized. If all active searches access the same few database segments concurrently, the operating system needs to keep much less database segments in memory at the same time than if all searches work on different parts. Therefore, the amount of thrashing is greatly reduced. Further, the utilization of the second and third level cache is much better.

The synchronization, however, has disadvantages as well. The runtime of BLAST searches is linear in the length of the query sequence. If one submits searches for long sequences and other searches for short sequences, the long sequences will delay the results of the short sequences, as these cannot proceed with searching through the next database segment until all other queries are completed as well. This has

negative impact on the average search time. To cope with this problem, the authors of [40] propose using multiple ring–buffers. Each ring–buffer is responsible for different sizes of queries. Thereby, short sequences do not need to wait for the results of long sequences.

2.2.4 BRIDGES GT3 BLAST

The BRIDGES GT3 BLAST [8] implementation extends mpiBLAST with a Globus Toolkit environment such that BLAST searches can be offered as a GT3 grid service. When users submit a big set of queries, this set is partitioned into several subsets. Each subset gets searched with mpiBLAST. PBS or Condor are in charge of the resource allocation for the mpiBLAST processes. The user can submit queries with a web portal, which acts as a client to the grid service.

A similar approach is pursued by SOAP–HT–BLAST [70] which uses the Soap::Lite module of Perl to provide a web service interface.

2.2.5 NCBI BLAST website

The NCBI BLAST website [48] which is currently probably one of the biggest running BLAST applications uses CGI scripts behind a load balancer to allow users to submit their queries. These queries are tagged with an identifier for the user to query results and stored into a replicated SQL database. A scheduler breaks the queries into 10 or 20 chunks each that search different parts of the database. The split points for these 10 or 20 chunks are adapted each day once to provide a good load balancing. After splitting the query, the chunks are assigned to free processors. The assignment is conducted on the basis of which content is expected to be in the buffer cache of the machines. After that, the machines report their results to special nodes, which merge, format, and store them to the database. [9]

2.3 Conclusion

We have seen several frameworks for master/worker applications. They have in common that a programmer needs to implement few functions or interfaces, while the framework provides the control flow for assigning, sending, and receiving tasks. These functions are provided in various degrees of sophistication. Several frameworks assume that the user executes the application for each problem instance independently. Others,

mainly Java base frameworks, allow running the application continuously to processes queries it receives at runtime. Neither of the frameworks supports the concept of data locality well.

Apart from the generic frameworks, we have discussed several implementations which are optimized for BLAST. These implementations do not provide a generic framework that can be used for other database searches but the BLAST search. Neither of the BLAST implementations intents to be executed as a persistent service. Therefore, they create a constant overhead for each search due to application startup and database loading.

This thesis intends to provide a framework that specializes on parallel searches for any kind of sequence database.

3 Approaches to the problem

There are many different ways the framework can be implemented. Some have advantages, others have disadvantages, but often it is impossible to find the single best solution. This chapter discusses several necessary design decisions and the possible alternatives.

3.1 High level Design

We begin the analysis with the high level design, which describes the general components of the framework.

3.1.1 Servicing style

As mentioned before, there are two major ways of processing requests, which we will denote as *persistent* and *transient service*.

A persistent service gets started only once. After that, it stays in memory and processes requests for users as soon as they arrive. Conversely, the transient service gets started on demand for each incoming request and terminates instantly after delivering the result.

Transient services have been used since the early days of the World Wide Web in form of CGI scripts. Each time a user sends an HTTP request to a web server, the server executes a CGI script, a standalone program, which parses the request, processes it and produces output that is sent to the user. The UNIX `inetd` daemon does virtually the same by listening for incoming TCP/IP connections and starting a process to serve the request.

The `inetd` itself runs as a persistent service which is started at boot–up time and runs until the server is shut down. Web servers run often as persistent services as well, because they are subjected to frequent requests that can be handled very quickly. The cost of starting the web server for each request outweighs the benefits of saving resources during times of little usage.

Either servicing style could be applied for this framework. The advantage of a transient service, i.e. CGI style processing, is that it is very easy to warp existing applications into frameworks that provide the servicing functions for incoming requests. It is not even necessary to modify the applications themselves. The framework would listen for requests, preprocess them so that they appear as regular requests to the existing application, execute the application on many nodes for different parts of the database, and finally aggregate the partial results. The framework does not consume resources in times when few queries arrive, and it is easy to develop and debug applications as they can still run as standalone applications, so one can use commodity debuggers to analyze problems.

The primary disadvantage of transient services as opposed to persistent services is the overhead of starting the application for each request. In the example of Spectral Comparison, reading a compressed binary database to memory took approximately the same time as comparing 12 mass spectra to the whole database in memory. The ratio of 1:12 illustrates why a frequent application startup is inefficient. We can keep gathering queries until the number of queries is big enough to justify an application startup, but this mitigates the problem instead of solving it. The application startup times alone justify the need for a persistent service, but there are other advantages of persistent services as well. We can carry over information about the execution of one query to the next. If we know that one processor is slower than the others, because the CPU is slower or the load is higher, we can reduce the fraction of the database this processor has to search through and redistribute the responsibility to other nodes. Furthermore, a persistent service facilitates interleaved processing and communication. While one thread blocks the CPU with searching through the database, another thread can use the network to send results of processed queries and to receive new queries. The worker who searches through the database can keep a buffer of queries and reorder their processing if it helps to reduce the total processing times. Reordering queries can improve the utilization of second and third level caches of processors and thereby reduce the communication on memory busses.

The arguments above should have motivated our desire for a persistent service style.

3.1.2 Application topology

As discussed in Section 1.2.1, the efficiency of a master/worker application depends on the ratio of communication to processing time per query. Increasing the number of

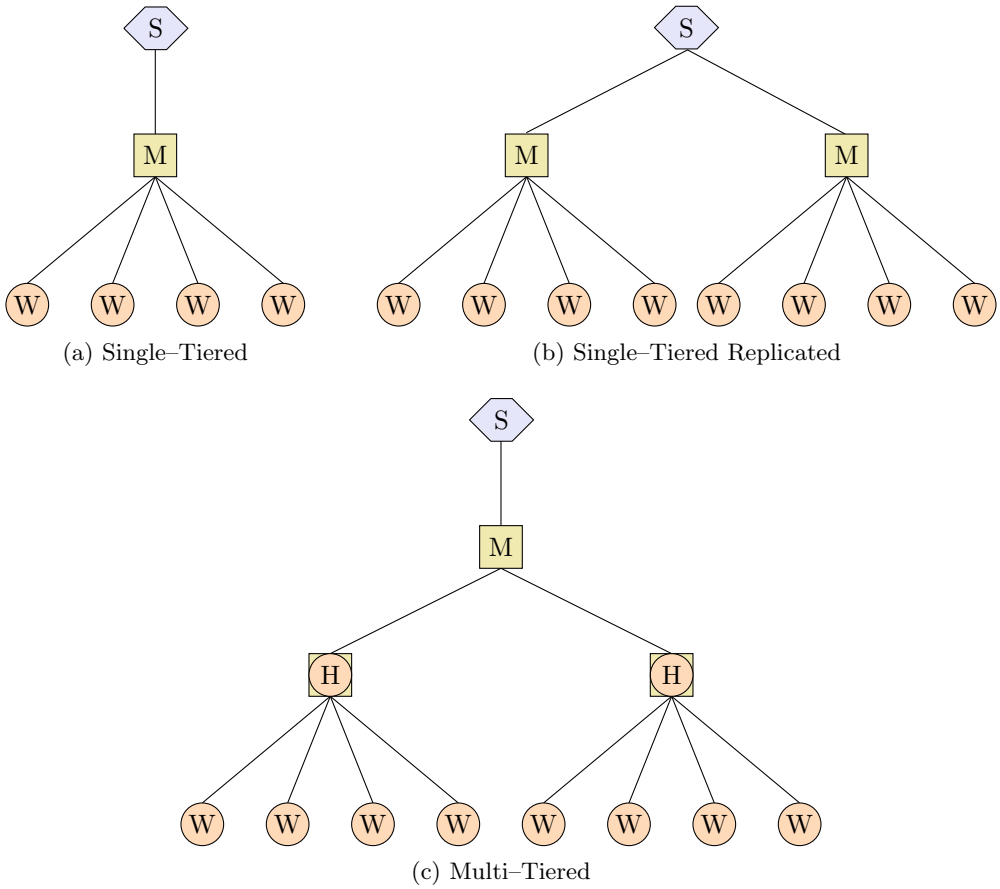


Figure 3.1.1: Topologies of master/worker applications

workers connected to a master increases the overhead as well because more subqueries and subresults need to be sent over the network and the aggregation of subresults to a final result becomes more computationally intense. Therefore, the efficiency decreases with the number of workers. There are alternative topologies that can mitigate this effect.

The *single–tiered* application described above is depicted in Figure 3.1.1a. W denotes workers; M, masters; and S stands for the source of queries, which can be a SOAP interface or a web–site. The obvious advantage of the single–tiered approach is its simplicity, and it works well in most cases. If the database is small, we need only few workers. If the database grows, it takes more time to process a fraction of the database on a worker, therefore the communication to computation ration decreases and we can

	few queries	many queries
small database	single–tiered	single–tiered replicated
huge database	multi–tiered	single–tiered replicated or multi–tiered

Table 3.1.1: Decision matrix for optimal application topologies

add more workers. However, we cannot scale linearly, because doubling the number of workers does not halve the total query processing time. We lose efficiency and need alternative approaches if the database grows extremely big or if the number of queries is very large.

One way to increase the throughput, i.e. to handle many queries, is to replicate the single–tiered structure and to ask the query source to assign queries to masters according to a load–balancing strategy like round–robin. Such a *replicated single–tiered* structure is shown in Figure 3.1.1b. It allows for adding nodes without adding load to the existing masters and thereby reducing response times in case there is an abundance of queries. If the application receives few queries, such that there is never more than one query in the system at a time, this approach does not help at all, because one part of the network is busy, while the remaining part stays idle.

Instead of replicating the single–tiered structure, one can introduce a multi–tiered structure as shown in Figure 3.1.1c. The query source sends the query to a master node. This node breaks the query into subqueries for its children, but these are hybrid nodes (H), which means that they correspond like a worker to their parent node but serve subqueries to their children like a master node. By introducing hybrid nodes, one can restrict the number of children of each node and thereby mitigate the problem of the network being a bottleneck.

Table 3.1.1 shows a decision matrix for application topologies. We will see that the modular structure of the framework, which has not been described yet, supports either type of topology without many modifications. In the following analysis we will assume the single–tiered structure, because it is easy to adapt the results to other topologies.

Another question that arises while discussing topologies is that of where to place the master node(s) if several nodes can be used. Obviously a master node is subjected to much higher network traffic than the workers. Therefore, a master node should be placed close to its children for low latency communication and should have a high bandwidth connection. This question has been analyzed in Gary Shao's PhD thesis [61] and shall not be subject of this thesis. We assume that all nodes share a homogeneous

network of a cluster. We will see later that the current implementation can be extended to calculate optimal placements.

If we consider availability and reliability as issues, the replicated single–tiered network is probably the best approach, because the application can continue working while parts of it are out of order. The problem for all other topologies is that they form a single MPI network. MPI as of version 1.0 does not allow nodes to join and leave the network at runtime, probably the most important disadvantage of MPI compared to alternative middleware solutions that were considered for this framework. MPI 2.0 defines means to allow nodes to join and leave the network at runtime but is neither mature nor disseminated enough to rely on. Therefore, we will not allow dynamic network changes. If one node crashes, the application needs to be restarted. This can be improved later.

3.1.3 Result aggregation

So far we have distinguished between master and worker nodes and assumed implicitly that the workers send their partial results to the master, who is in charge of aggregating the results and computing a final answer to the query.

By doing that, the master does not only have to send many messages to the workers but also has to receive many messages from the workers, which increases the danger of a network bottleneck. If the computation of the final result is expensive, the master's CPU can be a bottleneck as well. In order to compensate for that, we split the tasks of the master into two modules.

From now on, the master is merely responsible for distributing subqueries to the workers, and a separate *aggregator module* is in charge of aggregating the subresults. This increases the flexibility of our application topology.

Aggregator placement

Instead of placing the aggregator module at the master node, we can share the task among all workers. Each worker can run an aggregation module. When the master breaks a query into subqueries, it can assign one aggregator to each query, either based on the performance of workers or simply with a round–robin strategy. Workers process the subqueries but send the results to the aggregator of the respective subqueries instead of sending the results to the master. The aggregator processes the subresults and sends the final result back to the master, once all subresults have arrived and have been processed. The master receives only one result instead of one from each worker.

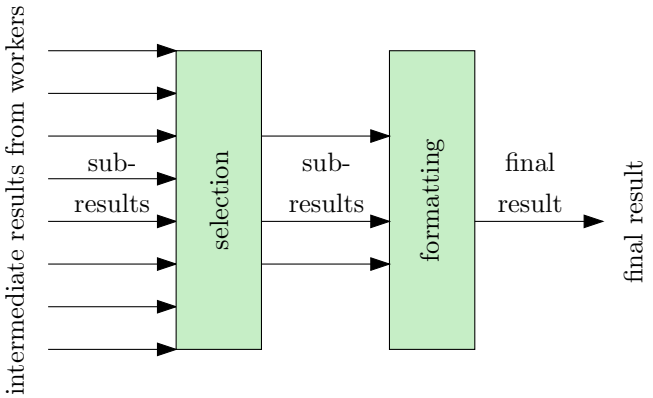


Figure 3.1.2: The aggregation process

Another strategy is placing aggregators on dedicated nodes instead of placing them on the same nodes as the workers. That way, nodes are not subjected to volatile environments where they have to process queries one time and process results another time. This might improve the cache efficiency and might deliver faster results. Suppose that we assign the job of aggregation to the worker in round–robin order and process two queries consecutively. The first query is processed on all workers and aggregated on worker W_1. While all workers but W_1 process the second query, W_1 is busy aggregating results. Therefore, all workers might finish their processing when W_1 starts working on the second query. Therefore, a single node delays the result. If we have dedicated aggregators, this does not happen. It is difficult though to find the right number of aggregators, so that they are neither underutilized nor a bottleneck. The decision needs to be made on a case by case basis.

Placing workers on other nodes than the master comes at additional costs. Usually the query contains information necessary to aggregate the final result, like for example about the output format. Therefore, we need to send the original query to the aggregator, creating one additional and potentially big message in the system for each query. Another potentially big message is the final result from the aggregator to the master.

Aggregation strategy

A second topic in connection with aggregation is the general strategy. The aggregation process usually consists of selecting the best matches of the database and producing a formatted report, as shown in Figure 3.1.2. This report can be optimized for being

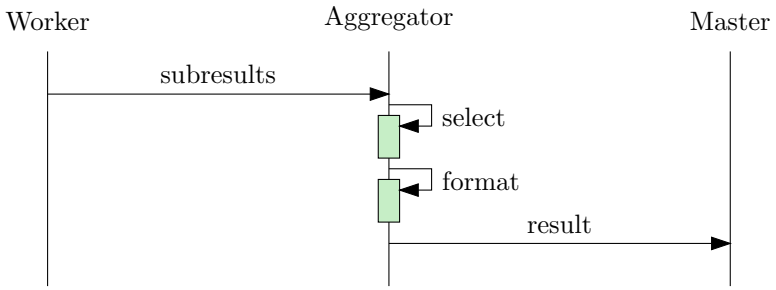
read by humans, i.e. it can be written as ASCII text, it can be written as XML for easy exchange between applications, or it can be even written in binary format if space is crucial. Depending on the actual application, we can optimize the aggregation strategy in many ways. The network might be the bottle neck in terms of bandwidth (too many big messages), latency (too many small messages) but the formatting can create a bottleneck at the aggregator as well.

In pioBLAST [41], the final result contains the spacious alignment of sequences, which is difficult to compute. The authors of pioBLAST have had great success with introducing a two phase aggregation. We begin with the description of the problems of a one phase aggregation and introduce two phase aggregation strategies later. In order to find the best 10 matches for a query, each worker has to send the best 10 matches of its database fragment to the aggregator, to ensure that no result fragment gets lost. If the network has $|W|$ workers, this means that the aggregator has to discard $10 \cdot (|W| - 1)$ results, and if the results have a moderate to large size, this might imply a huge overhead compared to a single processor evaluation. This aggregation is depicted in Figure 3.1.3a.

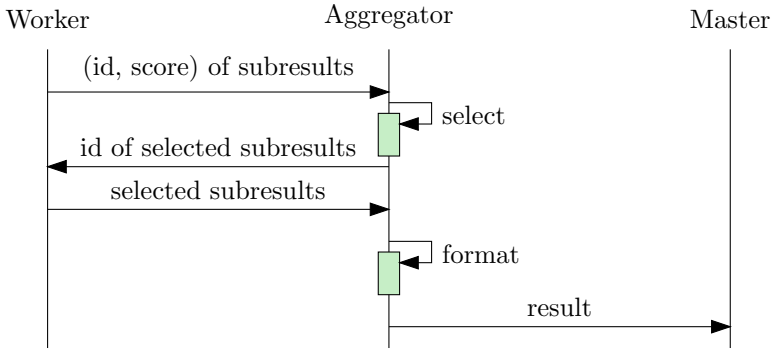
Often a small subset of the information stored in a subresult is sufficient to decide whether a subresult will appear in the final result or not. This subset might consist of nothing but an ID and a score of the matches; the actual results describing the matches can be omitted in the selection process. Using this idea, we can implement a two phase aggregation (see Figure 3.1.3b): During the first phase, each worker sends information that helps the aggregator to decide whether a subresult will appear in the final result. During the second phase the aggregator requests only those full subresults from the workers that are actually needed and completes its job by creating a formatted final result.

The authors of pioBLAST have used this approach not only to distribute the database search but also to distribute the formatting of the result (Figure 3.1.3c). Each worker calculates the formatted result of its top ten subresults and sends the score as well as the size of the output of each subresult to the aggregator. The aggregator selects the best subresults and tells the workers which subresults to write at which offsets in a result file. A parallel I/O module like ROMIO can be used to allow concurrent writing to an output file.

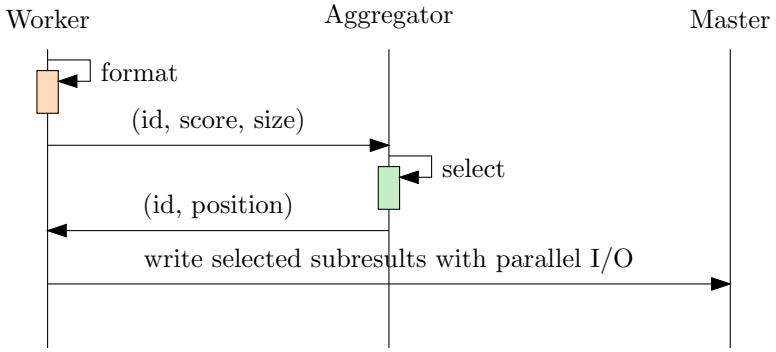
If the subresults are small and the job of generating the final result from these is fast, a user should stick to the standard way of having all work done by the aggregator



(a) Sending intermediate results to aggregator



(b) Preselection at aggregator



(c) Distributed aggregation

Figure 3.1.3: Aggregation strategies

because both alternatives generate additional messages on the network which might slow down the whole application. If subresults are big, the second or third options are preferable.

3.1.4 Internal parallelism

Searching through the database in parallel on many nodes instead of one allows utilizing the CPU power and memory of many computers but requires the overhead of network communication. We can mitigate this effect by performing network communication in the background, because communication and computation require different resources of the computer.

Using threads allows for a better utilization of the CPU because tasks like I/O to the hard drives and network can be conducted in the background. We do not need alternating communication and processing phases if we employ threads.

After splitting the master and aggregator module, we separate the network communication to a separate module now as well. Processing modules like the master, worker, or aggregator module can submit messages to the messaging module which takes care of the delivery and notifies the receiving modules when new messages arrive. Messages are stored in a buffer and sent as soon as the necessary resources are available. This removes blocking waits from the program execution.

MPI

The biggest problem of multithreading is the lack of thread support by MPICH-1.2.x, a very popular MPI implementation. The application hangs in dead locks after some time. However MPICH2-1.0.x and ScaMPI, a commercial MPI implementation, have been tested successfully to support threading. LAM/MPI is reported to support threading as well. All three applications require the programmer to serialize all MPI operations, but this can be achieved easily by the messaging module. It is the only module which is allowed to send and receive messages.

Apart from the requirement of a carefully designed communication, threading puts other burdens on the application and framework developer as well. We will analyze these in the upcoming paragraphs.

Synchronization

Every multithreaded application needs to take care of synchronization. No two threads are allowed to write (or read and write) to the same data structures in memory at the

same time, for the sake of preventing inconsistent states. The common means of solving this problem is the separation of memory ownership between threads and the use of critical sections for serializing access. The latter can create deadlocks in race conditions which are difficult to debug. As the application developer provides just call–back methods to the framework, the framework can take care of serializing calls to these methods. Therefore, an application developer does not need to pay attention to the problems mentioned, as long as the framework is implemented correctly.

Profiling

Analyzing the runtime of MPI applications is difficult, because the performance of one node depends on many factors like its CPU, the network bandwidth, and of course the performance of other nodes. A node might deliver a bad performance because its CPU is not fast enough, but it is likely as well that it spends most time waiting for input from other processors. Frameworks like MPE have been developed to help profiling MPI applications. The user can trace MPI calls to observe the network traffic, but this is seldom enough to understand the cause of bad performance. The often more important function of the profiling environment is to record the entering and leaving of sections of common function like reading the database, communicating, processing queries, formatting results, or idling. This can be displayed visually after the application has terminated. That way the programmer gets a visual idea of what the application was doing at each point in time during execution. However, this kind of visual profiling does not work well, if one cannot get separate profiles for each thread. MPE does not support this and we need a customized solution.

3.1.5 External interface

The one module still missing is an external interface to submit queries at run time. As discussed in Section 3.1.1, we prefer a persistent service, and therefore, we cannot use command line parameters to specify the queries. There are many possible solutions for submitting queries. These range from low level TCP/IP sockets to Corba and SOAP. As SOAP is gaining momentum due to the increased use of web serviced in grids, and as it is supported well for C++ and Java by frameworks like gSOAP2 and Apache Axis, we have chosen SOAP for the external interface. Any other interface can be easily plugged in as well.

As we do not want to expose the application developer to the details of SOAP, we have defined a set of methods that can be used for any database search application.

This includes the two important methods for submitting a query and requesting the result. Anything else is of minor importance at this point and will be discussed later. Queries and results are sent as strings, so that the application developer does not need to implement new XML schemata for a particular database.

3.1.6 Conclusion

We have introduced 5 different modules:

- Master
- Worker
- Aggregator
- Messaging Service
- SOAP Service

Any pair of Master, Worker, Aggregator and SOAP Service can communicate only over the messaging service. This strong encapsulation gives a great freedom for the topology. Figure 3.1.4 shows how modules can be connected on a node. Each node has an instance of an Application Model that provides the application specific functions that are used by the other modules. Furthermore, each node has a messaging module and a subset of Master, Worker, Aggregator and SOAP Service. This subset defines the purpose of the node. We can easily model topologies that run aggregators on any node, on the master node, or on dedicated nodes. The smallest working application runs on a single node, containing all modules mentioned above. Hybrid nodes for a multi-tiered topology as described in Section 3.1.2 can be implemented by running a Master and Worker module on one node. The Worker module just forwards any job to the Master module which serves its children with tasks.

3.2 Query Processing

After this analysis of the high–level organization of the framework, the next chapter is dedicated to the questions of what queries look like and how they are processed. Figure 3.2.1 shows the general idea. We are going to analyze several aspects of this like the regular structure and the flow of queries.

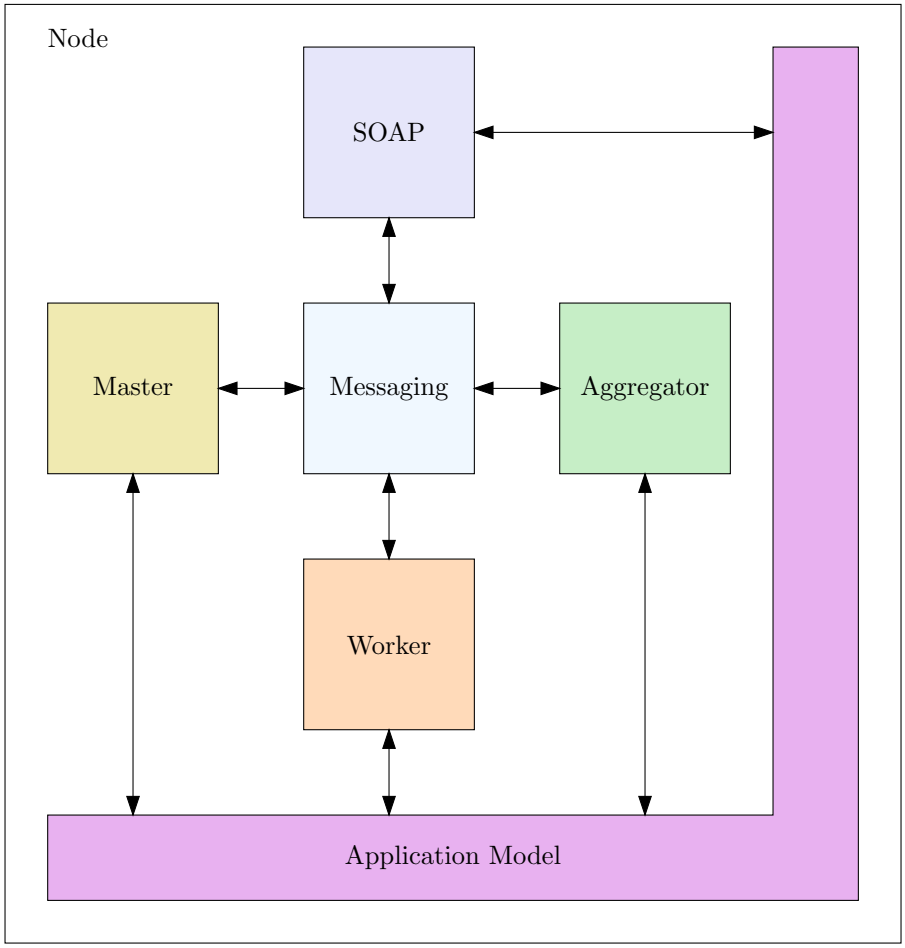


Figure 3.1.4: Framework modules on one node

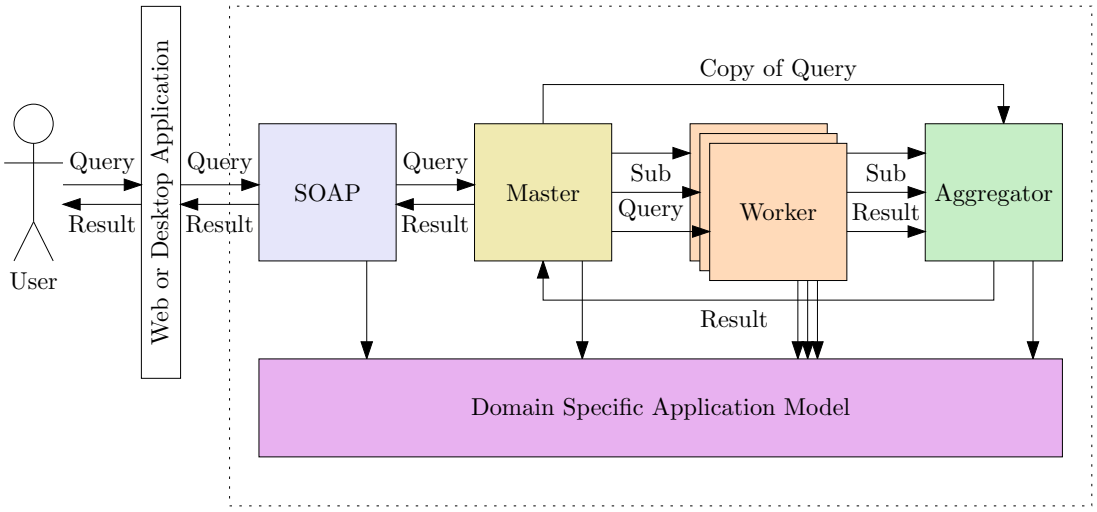


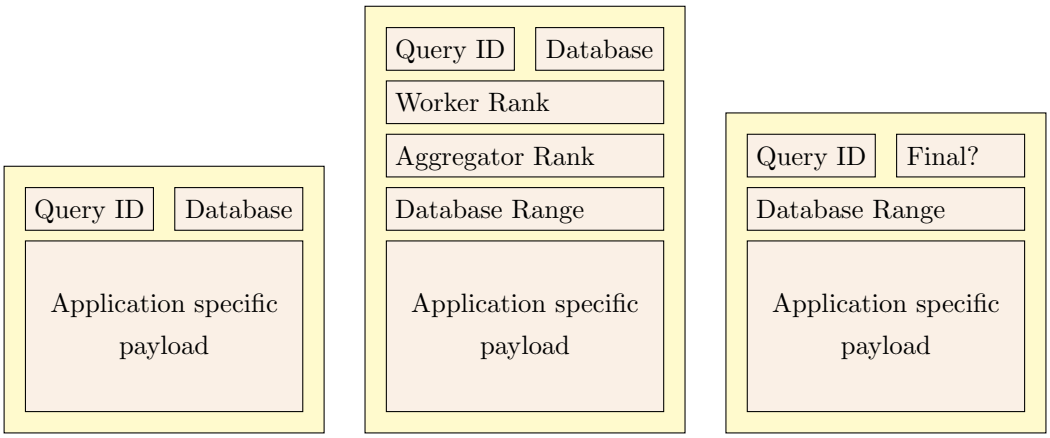
Figure 3.2.1: Query flow

3.2.1 Query structure

A user submits a query to the framework and needs to request the result once the query is processed. As the processing can take a long time, sending a query and requesting the result are two separate operations. In order to match the two, it is necessary to assign unique IDs to queries. If the application supports queries to more than one database, each query needs to contain a reference to the database as well. Apart from this general information, the queries contain the actual query terms as a payload. These can be stored in any form that is suitable to the specific application domain. The user can choose to transmit the payload as strings or use serialization means provided by the framework to compose custom data structures. Figure 3.2.2a illustrates the contents of a query object.

The master creates a set of subqueries for whose aggregation they need to be matched together. In order to do that, the subqueries and subresults need to contain a reference to they parent query. Furthermore, the workers need a reference to the respective database to actually evaluate the query and to understand the content of the payload. In order to determine, which worker should receive a subquery, and to which aggregator the subresults should be sent, the subquery objects have two more fields that determine the respective nodes' MPI ranks.

Results and subresults share the same base class. In order to distinguish results and subresults they contain a flag that is set to `true` in final results. Results do not contain



(a) Query (b) Subquery (c) Result

Figure 3.2.2: Data structures for queries and results

a field for the database name, as it tends to be long and is known to the receivers of results and subresults anyway. The master, who receives final results, knows the query ID to database name relation, because it created the query ID. The aggregators receive a copy of the query before they aggregate results.

3.2.2 Query flow

The regular query flow as depicted in Figure 3.2.1 is pretty simple as it follows a pipeline from the SOAP server, to the master, to the workers, to the aggregator and finally back to the master. Despite its simplicity, the pipeline is flexible enough to support even work cycles. If a master receives a result, it can refrain from storing it. Instead, the master can use the result to create new subqueries for the workers. That way it is possible to implement two phase searches. The first phase can be used to perform a superficial search with fast heuristics whose results can be used as cutoff values for a more detailed second search.

The programmer has to decide whether the additional network overhead outweighs the speedup of a preceding heuristic search.

Pushing, Pulling and Stealing

Three different models have been used in the past to assign jobs to workers; push models, pull models, and work stealing models.

Using *push models*, the master is in control of the points in time when jobs are

sent and to whom they are addressed. This model is usually inferior to pull and steal models because it is more difficult to get good load balancing, but we will see that it turns out to be a good choice for our application.

The domain of *pull models* are volatile environments like of Condor MW [21] where workers join and leave the network at runtime. The master does not need to keep track of the status of workers but can simply resend a task to another worker, if it does not receive a result within a certain amount of time. This eases the management of the set of workers. Another advantage of pull models is the easy load balancing because tasks do not need to be of similar difficulty. Some tasks can take a considerably longer time to finish than others. Workers who finish their tasks early simply request more work, and thereby stay busy until all tasks are finished.

Alas, the pull model does not support our application very well, because subqueries cannot be processed on arbitrary nodes. For processing a subquery, a worker needs to have the respective database fraction in memory, and as mentioned before, we generally assume that loading a database fraction from disk takes more time than processing a query on that fraction. Further, loading a new part usually means unloading another part that is actually assigned to the worker, as memory is limited. Processing one query on a foreign database part, therefore, results in two expensive loading operations.

The same arguments apply to *stealing models*. Their idea is to assign jobs to workers as soon as possible. Whenever a worker runs out of jobs, it chooses a random worker and tries to steal fractions of this worker's unfinished work.

As each subquery can be processed by only one worker in our model, the master can accept incoming queries, break them into subqueries, and assign them instantly to the workers according to the push model. As we cannot send one worker more queries than other workers, the approach being pursued, needs to assign smaller database ranges to those workers, who are known to be slower than other workers, in order to balance the load.

Considering this, we need a feedback from the workers to the master, which reports about the workers' performances. This makes our query flow model actually a hybrid of the push and pull models. Different from pure pull models, where idle workers keep sending pull requests, the workers inform the master of how many queries have been processed and how much time they spent doing that, so that the master can decide whether more work needs to be sent to the workers when new queries arrive.

The workers actually do not need to inform the master explicitly about their finishing

of a query, because the master receives the final result from the aggregator and can conclude from that, that a worker finished its subquery. With this, it is possible to estimate the number of queries in the workers' queues.

This leads us to the question, when a master should send more queries to its workers.

Query queues

A master might create and send subqueries to the workers as soon as a new query arrives, or it might queue incoming queries until a result from the aggregator arrives. Either way is bad.

If the master creates and sends subqueries as soon as new queries arrive, this means that the workers might build up long subquery queues. This is an inefficient use of the aggregated memory, because many subqueries are usually bigger than a single query. Furthermore, it takes away control of the master. If a master realizes that one of its workers is getting slower than the others, it needs to redistribute the responsibilities of the database fractions. However, this update of responsibilities cannot be executed until all previously sent queries are completely processed, as they have database ranges assigned to them. That means, even though a load imbalance is observed, the master cannot change anything for a possibly long period of time.

Conversely, if the master sends subqueries only if a result from a previous query arrives, there is at most one query in the pipeline and workers spend much time just idling.

The solution is to choose a balance between the two approaches: The master tries to keep a certain number of queries in the pipeline. If it falls short that number, more queries are sent. This approach can be enhanced by sending bursts of subqueries, as several subqueries can be combined into one message. The fewer messages are sent the higher is the throughput. Whenever the number of subqueries in the pipeline falls below a lower limit, the buffers are refilled to their maximum capacity.

As the master does not know where the queries are in the pipeline, buffers need to be big enough and limits need to be chosen carefully to make sure that workers never run out of tasks.

As the query queue is highly encapsulated from the remainder of the program, it is pretty simple, to extend the current implementation that stores all queries in memory by features which swap parts of the queue to a database or hard disk. This is necessary, if many queries arrive at peak times and need to be processed at off–peak times.

3.2.3 Scheduling

Schedulers can optimize the observed performance of the database in several ways. In case of a replicated single–tiered application topology, expensive queries can be processed on other nodes than quick queries, in order to prevent computationally expensive queries from creating huge delays for shorter queries. This approach is pursued by the NCBI servers [48] as well.

Virtually any non–preemptive scheduling strategy for single–processor systems could be employed to determine the order in which queries are processed.

- First Come First Served (FCFS) is the simplest mechanism and is currently implemented. It can be considered as a fair scheduler, because queries are processed in the order in which they arrive.
- Shortest Process (Query) Next (SPN) prevents long queries to block the CPUs for smaller queries and thereby reduces the average response time. This approach requires, however, an estimation of the processing time of queries. Depending on the application model, processing times of queries can be similar or diverge a lot. It is the responsibility of the scheduler to take care that long queries do not starve.
- Priority Based Scheduling (PBS) can sort queries in descending order of priorities. Researchers might assign priorities to their queries and thereby allow more important queries to be processed first. Penalties might be added to queries of researchers from different institutions or of users who use the service too excessively.

Either of these scheduling strategies can be implemented by adding a new query queue type that returns queries in different orders, without modifying any other parts of the framework. Currently, however, the FCFS strategy is the only one implemented.

The scheduling strategies mentioned above enforce a certain level of fairness. Other scheduling strategies can actually increase the framework's throughput.

Concurrent processing

Suppose that the database fragment which is assigned to a node exceeds the node's memory size. If memory pages are swapped in a least recently used (LRU) fashion and the database fragment is searched through sequentially for each query, this results in an extremely bad performance. Each time a query is compared against a record of the database this part is swapped out to disk.

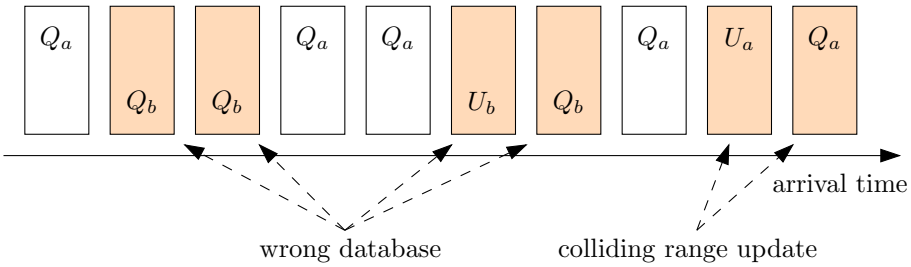


Figure 3.2.3: Prefetching of queries from the queue

We can improve the throughput tremendously by reordering the way queries are processed. If workers evaluate 10 queries concurrently, they can compare each query against the first database entry, then the second database entry, and so on. Thereby, it is guaranteed that at least 90% of all entries are found in memory. This approach can be implemented by a ring–buffer strategy as described in Section 2.2.3.

Even if the whole database fraction of one node fits into memory, this reordering may increase the throughput, as the hit rates on the second level cache are higher.

Query reordering

If the application supports more than one database, a worker can use query reordering in order to improve the performance. Suppose that the application serves two databases, each of which is big enough to fill the entire memory. If queries arrive in alternating order for the two databases, each query requires to swap out a whole database and to reload the other one. It is better to process a burst of several queries on the first database and then proceed to the second database afterward, as this strategy reduces the amount of swapping.

The query queue maintains a list of queries that need to be processed. We need to find criteria according to which we group queries. These criteria shall increase the throughput but prevent the starvation of queries on rarely used databases. Furthermore, they have to be aware of range updates. As we want to group queries on the same ranges of the same database, we do not allow queries on different databases or queries before and after a range update request to be grouped together.

Figure 3.2.3 shows the state of a query queue with queries on two databases labeled Q_a and Q_b, respectively for database a and b, as well as update requests U_a and U_b. For the sake of visual clarity, all labels of operations on database b have been lowered.

The application model gets called by the worker to process the first query in the

queue and can request additional queries in order to process them concurrently. The query queue will do a linear search for appropriate matching queries. Several queries are marked to be disregarded in the example because they address a different database. Further, no query may be considered after update request U_a because all following queries regard different database ranges.

The developer can set additional constraints as of which queries match to the first query. After selecting and processing all white queries in the example, the first Q_b query moves to the front of the query queue and is the next one to be processed. This illustrates, that queries do not starve, as they move to the beginning of the query queue eventually.

3.3 Database Aspects

After this discussion of the high level design and the handling of queries, we will discuss database aspects that are important to the framework.

3.3.1 Database model

As the framework targets as many databases as possible, it is important to make few assumptions about the database model. These are:

- *unique name:* Each database has a unique name. This name may coincide with the file or directory name of the database but can be independent of that as well, as long as the application model knows how to access the database.
- *known number of records:* Each database can determine the number of records n it contains efficiently. If this operation is expensive, the application can count the entries once and cache the result for later requests.
- *consecutively numbered records:* Records in a database are numbered consecutively from 0 to $n - 1$.
- *constant time access to all entries:* If asked to load an entry into memory, the database can access each entry in constant time. Usually some kind of index is necessary for that, which is small enough to be kept in memory all the time.
- *capability to load records:* The application knows how to load records into memory. This includes the assumption that records are either available on a shared network file system or sent by the application specific implementation (all tools necessary for that are included in the framework).

We can see from this list that the framework makes no restricting assumptions about the content of the database, except that records are enumerated, but even if the database has different primary keys, it is easy to introduce one–to–one mappings from the primary keys to numbers.

The task of loading parts of the database into memory is up to the developer of the application specific implementation. The framework decides just which parts should be kept in memory by which node. We need to analyze this partitioning of responsibilities and how these responsibilities can be refined to deliver a high throughput.

3.3.2 Partitioning of responsibilities

One factor that determines the possibilities of partitioning the database responsibilities is the amount of RAM available.

If the total amount of RAM available is much bigger than the size of the database, we might assign database fragments to more than one node and thereby relax the rules, which node can process which queries. This might improve the load balancing.

If the total amount of RAM available is comparable to the size of the database, we should not assign a database fragment to more than one node, because that would produce expensive reload operations if a query for a fragment not in memory arrives.

If the total amount of RAM available is less than the size of the database, we cannot keep it in memory. Often a customized paging implementation as described before can increase the throughput considerably.

From these options the second one is the primary target of our framework. If nodes lack RAM, the developer can implement a page swapping algorithm in the application specific part. We do not consider the first option because it increases the complexity of the application dramatically. Two nodes might be equally suited for processing a subquery, but assigning the subquery to one node, might block this node form processing the next available subquery for which it might be the only node available. Assigning a stream of subqueries to nodes is anything but trivial. Therefore, we stick to the simpler strategy that assigns a fraction of the database to exactly one worker. If this worker turns out to be slower or faster than its peers, the size of its fraction of the database can be augmented or reduced.

3.3.3 Refinement of responsibilities

Degrees of variability

Workers can be bound to database partitions in various degrees, ranging from bindings that never change to very loose bindings.

The easiest approach is a *static* binding of database partitions to workers. During the initialization phase of the framework, a master assigns a database partition to each worker which does not change at all during runtime. This works well if all workers have the same speed and processing queries consumes the same time on all database partitions. As there is neither a demand for workers to measure their performance nor to report it to the master, this approach reduces the overhead of the framework.

A binding where the master assigns database ranges to workers which are rarely changed is considered *semi–static*. This makes sense on a shared cluster that is mostly idle and allows running the framework in the background. If nodes are used by other applications, the framework needs to adapt to the situation and needs to reduce the assigned database partitions on the occupied nodes. For semi–static bindings we request that partitions are always ordered according to the worker IDs. That means that the first worker is responsible for the partition that starts at record 0. The second worker is responsible for the partition right after that. Keeping this order has the advantage that workers often do not need to read a whole new database partition when they receive an update request. Usually it is sufficient to read few entries at the left or right end of their previously assigned database range.

A final approach is using completely *dynamic* bindings, where the master changes the assigned ranges of workers frequently at runtime. Assignments of database ranges to workers are only temporary and last for only one query. This approach is useful for applications that handle many databases of which some are used rarely. In order to use this approach efficiently, the master has to exploit that database ranges can be kept in memory and resuse them for the next queries. However, this is difficult to handle because the master needs to know the content of all query queues at workers and needs to keep track of which database fractions are currently loaded on these workers. Even if this information is available at the master, it is difficult to find an optimal assignment that reduces the number of necessary database reloads. In case an application handles many databases and most queries concern one database, it is usually advisable, to

dedicate one node for less popular databases. Dispatching between this node and the remainder of the network can be done easily on the SOAP level.

Neither of these approaches requires the workers to keep the ranges assigned to them in memory all the time. The application specific implementation can decide whether to keep the database in memory, whether to use ring–buffers, or whether to unload a database fraction completely if it has not been used recently.

Calculating good ranges of responsibility

The ranges of responsibility should correspond to the performance of the workers as a single slow worker is capable of delaying the result of a query. The highest throughput is achieved if all workers process the units assigned to them with the same speed. The master is in charge of finding appropriate ranges of responsibility for each worker based the performance of the workers in the past.

The first question is how to find a good performance indicator for the workers. After that we can focus on calculating good assignments.

A good performance indicator for workers is their performance in the past. Workers who used to be slow are unlikely to be very fast in the future. However, they may be subjected to changing environments if they share the CPU with other processes. We can account for that by continuously measuring the time the worker spent on processing the most recent query. Using only the most recently measured time of a worker as estimation would result in a very volatile value and many range refinements. Using the arithmetic mean instead is not good either, because values from a long time ago are weighted too much. A good compromise between the two strategies is an exponentially decaying average which is calculated according to the formula

$$\bar{D}(t) = \alpha \cdot d + (1 - \alpha)\bar{D}(t - 1) \tag{3.3.1}$$

where $\bar{D}(t)$ describes the averaged value of the duration to process a query at time t, d describes the most recently measured value, and α represents a learning factor. Figure 3.3.1 shows a set of measured processing times and the averages computed according to different formulas. The measured processing times are fluctuating and experience a strong peak in the middle. We can see that the arithmetic mean shows hardly any reaction to the peak but shows the effects still a long time after it vanished. The graph shows further floating averages with different learning factors. We see that a learning factor of 0.4 produces a very volatile curve which would cause a lot of

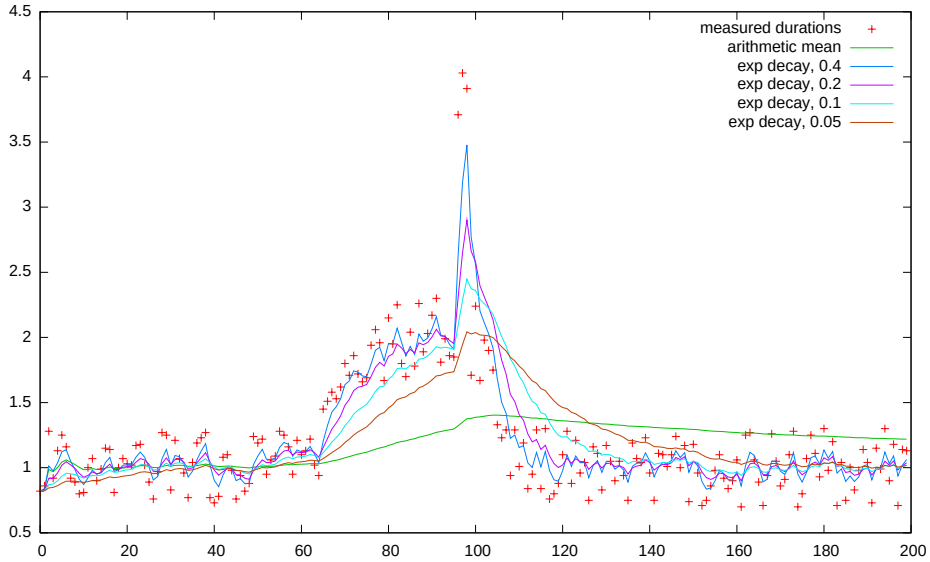


Figure 3.3.1: Average computations for measured processing times

database range updates. Lower learning factors like 0.1 or 0.05 produce good results. They react to the changed situation while being smooth enough to not cause too many range updates.

In order to reduce the communication between workers and the master, which needs to know the workers' performance, the workers are in charge of calculating the average query processing times. Update messages are sent to the master only every k queries, where k is configurable. The master might estimate a worker's performance by measuring the time from sending a subquery to receiving a confirmation that it has been processed but this produces bad values due to query queues, message queues, and network delays.

All workers start with equal shares of the database and send messages to update the master node's information about the average durations the workers spent on processing queries.

If the master has received enough update messages from all workers, it can decide whether it wants to request an update of responsibilities. For that it determines the average duration of all workers and checks whether any worker deviates by more than a relative threshold. This prevents update requests if workers deviate by only 1% from the average.

After deciding that an update is necessary, the master needs to assign new responsibilities. It knows the average duration to process one query $\bar{D}_i$ of each worker i. Further it knows the size of the database range R_i assigned to each worker i. We define the size as the number records in this range. As each worker i processes $|R_i|$ records in total time $\bar{D}_i$, we can denote the average performance of worker i as

$$\bar{P}_i = \frac{|R_i|}{\bar{D}_i}.$$

We define the total database range and the total performance as

$$R = \bigcup_i R_i \quad \text{and} \quad \bar{P} = \sum_i \bar{P}_i.$$

We can assign new ranges R'_i to workers now such that

$$\frac{|R'_i|}{|R|} \approx \frac{\bar{P}_i}{\bar{P}}.$$

The fraction of the database assigned to one worker equals the fraction that this worker contributed to the total performance of all workers in tandem.

After this reassignment of responsibilities, the workers' average durations of processing queries move close together and are more homogeneous. However, depending on the periods for which a worker's performance is decreased by other processes, additional refinements in either direction may be possible. In practice this works pretty well, as we will see in the benchmarks.

The algorithm assumes that all database records have a common length. This is no problem, however. Suppose a database has records sorted by increasing length, such that all short records are at the beginning of the database and long records are at the end of the database. The time to process a subquery can differ considerably between the workers. For the algorithm above, however, the situation appears as if the first worker was faster than the other workers, as it does not care for the reason of this worker searching through records faster than other workers. It will simply assign a bigger range of the database to the first worker and the situation levels off.

The same happens if some records are relevant for queries less often than other records. Irrelevant records can often be processed quicker than relevant records, because it is apparent that they will not deliver good matches. The algorithm will notice

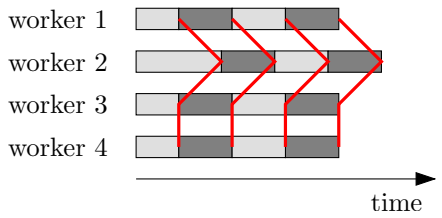


Figure 3.3.2: Persistent delays of one worker

that a worker has many records that are rarely relevant and assign this worker a bigger fraction of the database.

Yet, there are situations that cannot be handled very well by the algorithm.

Suppose one worker has less memory than other workers but the same type of CPU. Suppose further that an equal distribution of database sizes to workers forces this worker to swap memory to disk while all other workers can keep their fraction of the database in memory. In this case, the algorithm will realize the bad performance of one worker and reduce its range, such that it does not swap any more. Now the worker is faster than the other workers, as it has the same CPU but a smaller range of responsibility. Therefore, the algorithm will increase the worker's responsibility in the next step. Either outcome is possible. The assignments are likely to swing into equilibrium. However, it is possible as well that they keep oscillating which results in a very bad performance. As the moving average reacts slowly to the changed situation, the workers are more likely to reach equilibrium as counter–reactions are usually weaker than the refinement that caused them.

Figure 3.3.2 shows another problem. The gray boxes represent subqueries being processed on the workers. While worker 2 was processing the first subquery, the refinement algorithm has leveled off this difference early. However, as we can see from the vertical curves that represent the time when subqueries belonging to one query are finished, worker 2 is never able to catch up with the other workers. This has no serious impact on the throughput of the application, but on the experienced responsiveness as a result cannot be delivered until each worker has finished its work. Worker 2 adds a constant delay to each result.

It is very difficult to handle this problem, as it is not sufficient for the master to know the average performance of the workers, but it needs to know the current queue lengths as well. The master might reduce the database assignment for a worker temporarily until it has caught up with the other workers, but this is very difficult to handle

correctly. Instead, we assume that the load of the framework allows for certain idle periods. During these periods workers are able to catch up. If the load is extremely high, however, this does not work out and the framework performs badly.

3.3.4 Number of databases

The discussion so far has not paid much attention to the question of supporting more than one database at the same time. There are several ways of handling this.

One way is to use the replicated single–tiered topology and assign different databases to different master/worker networks. This is very easy but does not address a problem that is very likely to occur: Databases are unlikely to be equally popular, which means that some databases will receive considerably more queries than others. If the distribution of computational resources does not account for that, some nodes will idle while others are very busy.

Dynamically bound responsibilities have no fixed assignments of databases and can thus swap databases easily. However, we came to the conclusion that this approach is difficult to implement efficient.

The approach chosen is the following: Each worker is responsible for database fragments of several databases at any time. Databases which are used infrequently are swapped to disk if RAM is scarce and therefore do not have any bad impact on the other databases, except that a query to a unusual database can force popular databases to be swapped out to disk. To mitigate this effect, we have introduced a possibility to reorder queries by pre–fetching queries of the same database from the query queue. That way, several queries to the same unpopular database are grouped and the swapping pays off. If just one single query to an unpopular database is in the queue, this query still gets processed because we do not want it to starve. However, it might be a better idea to dedicate a few nodes to a collection of unpopular databases.

3.4 Application Model

We have analyzed all important aspects of the framework so that we can conclude the requirements for the application specific implementation now. We consider an application model an object that handles the application specific aspects of one database. Several databases, even with similar contents, are represented as several instances of the application model.

Each application model is identified by the database name it represents. It needs to support the following functions:

- *breaking a query into subqueries*
- *processing a subquery*
- *aggregating subresults*
- *loading database ranges* and *determining the number of records in the database*
- *transforming a textual query description as sent by SOAP to an internal query*
- *storing and delivering results*

The functions listed above need to be implemented by a class and provided to the framework as call–back methods. The upcoming chapter will discuss the implementation aspects of the framework.

4 Implementation

This chapter introduces the technologies used for this framework and explains the reasoning of using them. After this, we will introduce the framework's API and explain the separate modules of the implementation. Finally, we will show an implementation of a sample application, which was used for the benchmarking as well, and describe how to setup the framework on a cluster.

4.1 Technologies

The decision to use or not to use external libraries is always a tradeoff between reinventing the wheel and creating additional dependencies that are difficult to install. Libraries have been chosen with the following criteria in mind:

- Platform independence
- Licensing
- Popularity

The libraries shall not restrict the application to one operating system. The framework is explicitly targeted and has been tested on the GNU/Linux operating system, but it should work with no or little modifications on any recent UNIX platform.

No proprietary libraries are required, so that the framework can be used and distributed under the GNU Public License.

In case more than one library was available to solve the same problem, we have tried to choose the one with the bigger user base, to ensure that development continues and that is it easy to install the library with most Linux distributions.

4.1.1 Boost

The Boost Framework [13] consists of many modules that support platform independent development in many areas.

From the set of available modules, we have chosen to use Boost's threading support, function binding, concept checks, string handling with formatting, file system abstrac-

tion, and unit tests. We have decided not to use Boost's serialization library but to implement this on our own.

The threading support encapsulates pthreads in a C++ wrapper and provides the threads and locking mechanisms like mutexes and condition variables. Concept checks are an extension of the C++ template mechanism to enforce that classes used as template parameters implement a given interface. The Boost library compensates for shortcomings of the STL in string formatting (`sprintf` style) and file system access. Finally, it provides unit tests and assertion macros.

We have chosen not to use Boost's serialization library, because it does not support platform independent binary serialization. Serialization is either binary and platform dependent or platform independent as ASCII text. The first option is bad for heterogeneous hardware; the latter option is inefficient, as we attempt to keep the network traffic as low as possible.

4.1.2 MPI

There are several implementations for the MPI standard. MPICH and LAM/MPI are probably the best known open source implementations. The framework has been tested successfully with MPICH2 and the commercial ScaMPI implementation.

MPI offers platform independent communication with excellent performances, sometimes surpassing self–written TCP/IP socket implementations. This is possible by bypassing the TCP/IP stack with shared–memory communication where available and supporting custom hardware like Quadrics and Myrinet to provide low latency, high–bandwidth communication[1]. Supartik Majumder's Master's Thesis [43] provides an excellent overview of the principal architecture of MPI libraries like MPICH and LA–MPI.

4.1.3 SOAP

The SOAP protocol allows for remote procedure call by encoding function calls as portable XML messages and sending them over communication protocols like HTTP. The advantage of SOAP is less its performance (SOAP has a huge overhead) but rather its platform independence and easiness. SOAP implementations exist for Java, C(++), Perl, Python, PHP, Ruby, and virtually any existing scripting language of importance.

[1] "[...] the MPI message latency with Myrinet is up to 5 times lower than with Ethernet." [43]

This allows for easy integration of the toolkit into other applications. Programs can easily submit queries to the framework and request the results.

The gSOAP implementation has been chosen because it is easy to use and widely employed in applications around the grid. The Globus Toolkit can extend gSOAP by means of authentication and secure communication.

4.1.4 Apache log4cxx

The Apache log4cxx library resembles the popular Apache log4j library for logging debug and status messages. This has proven to be a valuable tool for debugging distributed applications, as commodity debuggers like `gdb` are difficult to use in this scenario. The log4cxx library comes with macros which ensure that logging statements create hardly any overhead if they are disabled. Enabling and disabling logging can happen at runtime and does not require recompiling the application.

4.2 API and Internals of Modules

The upcoming paragraphs describe the implementation details of the framework. They comprise an explanation of the existing source codes as well as an explanation of code that needs to be written in order to use the framework.

As the previous chapters have given a high level overview of the components of the framework, we will now iterate bottom–up over the framework to give a comprehensive explanation of the modules and how they cooperate.

The framework consists of several modules which are separated into several namespaces according to their functions:

TF::Utils:	Contains several utility functions for the initialization and configuration of the application, benchmarking, and binary I/O.
TF::Serialization:	Contains classes to serialize and deserialize data for network transport.
TF::Thread:	Contains an abstraction of threads.
TF::Messages:	Contains the definition of messages and a messaging service, which is responsible for message delivery.

TF::Query:	Contains the definition of classes related to queries, like queries themselves, subqueries, and results.
TF::Topology:	Contains topology information about how nodes work together.
TF::Coordination:	Contains a class that initializes the nodes by starting masters, workers, and aggregators according to the topology. It provides further access to the different modules at runtime.
TF::Master:	Contains the interface and an implementation of a master.
TF::Worker:	Contains the interface and an implementation of a worker.
TF::Aggregation:	Contains the interface and an implementation of an aggregator.
TF::Interfaces:	Contains a SOAP interface.
TF::ApplicationModel:	Contains the interface of an application model, the application specific part of each implementation.

We inspect these packages now in greater detail. The purpose of the upcoming chapters is to provide some background information and everything necessary to make use of the framework. All source codes of the framework contain verbose Doxygen comments with even more detailed descriptions of the algorithms and concepts.

4.2.1 TF::Utils namespace

The `TF::Utils` namespace defines several utility functions. These include the application startup, configuration, benchmarking, and binary I/O.

- ```
  void TFInit(int *argc, char ***argv);
  ```

  This function initializes the framework by starting MPI, calculating topology information, and initializing the logging. It should be executed before any other function calls or object instantiation of framework classes.

- ```
  void TFFinalize();
  ```

  This function needs to be called by each worker before terminating the program; it releases resources like MPI and flushes all buffers to disk.

- `void TFAbort(int errorcode);`

  This function can be called for aborting the application in case of errors.

- `void TFActivateCrashHandler(char *executable);`

  This function activates a crash handler which uses `gdb` to generate a stack trace of the application when it is about to crash.

Using these functions, a `main` method should look like this:

```
#include "TFUtils.h"
int main(int argc, char** argv) {
    TF::Utils::TFInit(&argc, &argv);
    TF::Utils::TFActivateCrashHandler(argv[0]);
    // actual code
    TF::Utils::TFFinalize();
    return 0;
}
```

Besides these startup methods, the `Utils` namespace provides class `TF::Utils::TFConfig` for reading configuration files.

`TFConfig` looks for a file `tf.config` in the current working directory or the user's home directory. The file follows a simple key/value format with entries like this:

```
# definition of the directory containing the database
database path = /var/tmp/database
```

The `TFConfig` class provides methods for several data types to access the values stored in the configuration file. All methods follow a similar schema:

```
static
std::string getString(
                      std::string key,
                      std::string const& defaultValue = "" );
```

In order to query a value, the user can call

```
TFConfig::getString("database␣path", "~/database");
```

The method will return the `database path` value as defined in the `tf.config` file or the default value `"~/database"` if no configuration file exists or no value is specified in this file.

Finally, the `TF::Utils` namespace contains a class `TFEventLogger` which is used for profiling.

The two functions

```
static inline void start(int thread, int eventNumber);
```

and

```
static inline void end(int thread, int eventNumber);
```

can be used to record the beginning and end of a program state. This recording helps to get an impression of the execution of the programm and to identify bottle necks. The states of each thread are recorded separately. The states of a single thread may be nested but may not overlap otherwise.

The following example illustrates how the event logger is used in the worker implementation.

```
TFEventLogger::start( TFEventLogger::TH_WORKER,
                      TFEventLogger::EV_PROCESSING );
applicationmodel->processQuery(...);
TFEventLogger::end( TFEventLogger::TH_WORKER,
                    TFEventLogger::EV_PROCESSING );
```

The `TFEventLogger` records the times of entering and leaving the `EV_PROCESSING` state, and the `tfprofiler` can generate graphical reports from this information as shown in Figure 4.2.1. Further details about profiling and how to interpret the diagram will follow in section Section 5.1.2.

4.2.2 TF::Serialization namespace

Figure 4.2.2 illustrates the relation between the three classes of the `TF::Serialization` package:

The `TFSerializer` offers stream operators to serialize atomic data types like `int`, `float`, and even `std::string`. By using the `MPI_Pack` and `MPI_Unpack` methods, it serializes and deserializes data in a platform independent, yet space efficient, way.

In order to serialize variables, one can simply stream them into a `TFSerializer` object and convert this to a character array when finished as shown in the following example:

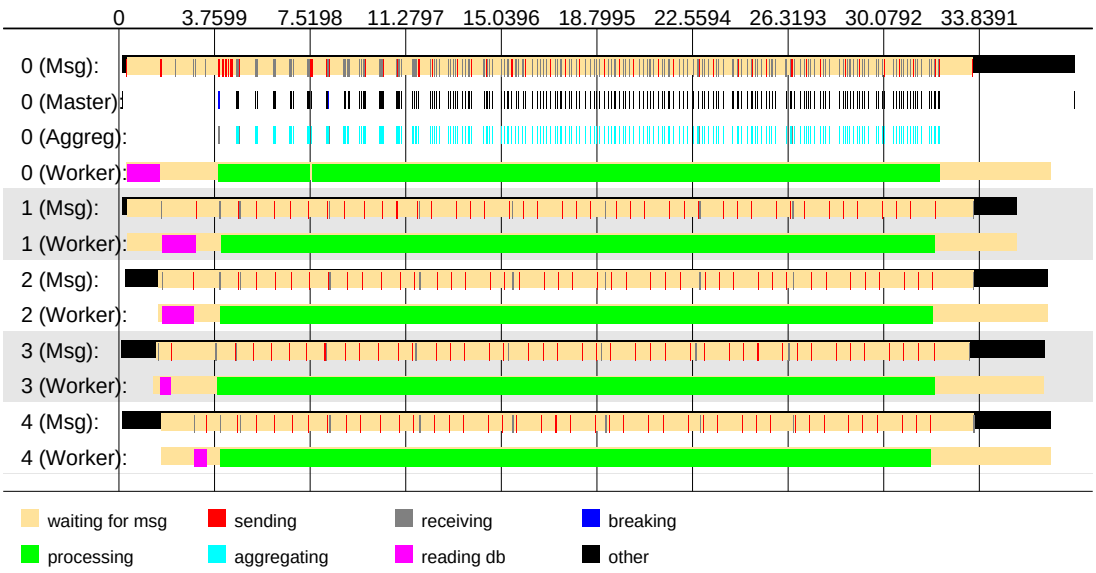


Figure 4.2.1: Example profile

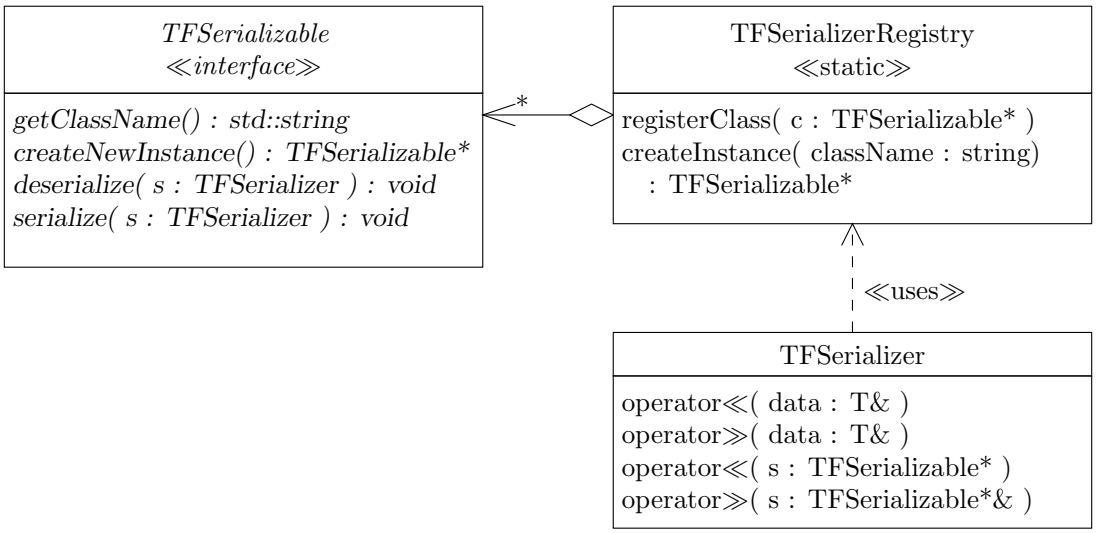


Figure 4.2.2: Serialization class diagram

```
// write values
TF::Serialization::TFSerializer serializer;
serializer << 1 << 2 << -3 << 42;
// store in character array
char * buffer;
int buffer_length;
serializer.serialize( buffer, buffer_length );
// do something with character array
...
// release buffer
delete[] buffer;
```

Deserializing a stream is equally simple:

```
int a, b, c, d;
TF::Serialization::TFSerializer
    deserializer(buffer, buffer_length);
deserializer >> a >> b >> c >> d;
```

In order to serialize and deserialize polymorphic classes we need to store type information about the objects in the stream. For that, the classes need to implement the `TFSerializable` interface, which provides methods to request a class ID, to create a new instance of a class, and to serialize and deserialize an object. As we need to create new instances of serialized classes at runtime, these classes need to be registered at the `TFSerializationRegistry`. This can use the `createNewIntance()` method of the `TFSerializable` interface to create the instances needed.

This is a simple example class that can be serialized:

```
class S : public TF::Serialization::TFSerializable {
public:
    int x;
    S() {}
    virtual ~S() {}

    const std::string& getClassName() const {
        static std::string className("S");
        return className;
```

```
    }

    TFSerializable *createNewInstance() const { return new S(); }

    void deserialize(TF::Serialization::TFSerializer& serializer) {
        serializer >> x;
    }

    void serialize(TF::Serialization::TFSerializer& serializer)
    const {
        serializer << x;
    }
};
```

The registration at the `TFSerializationRegistry` is as simple as

```
TFSERIALIZER_REGISTER( S );
```

Once a class implements the required methods, objects can be serialized with

```
S *a = new S();
serializer << a;
```

and deserialized with

```
S *b;
serializer >> b;
// ...
delete b;
```

4.2.3 TF::Thread namespace

The `TF::Thread` namespace contains only one class `TFThread` which provides a framework for threads with an interface similar to Java's `Thread` class.

A thread can be in either of these states:

- `NOT_RUNNING`, i.e. not started, yet, or already terminated
- `RUNNING`, i.e. started and not terminated, yet
- `STOP_REQUESTED`, i.e. a stop was requested but thread has not terminated, yet

The `TFThread` class has few functions:

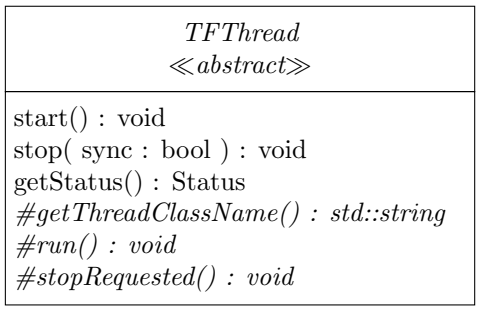


Figure 4.2.3: Thread class diagram

- `void start();`

  This method starts the thread, sets its execution status to `RUNNING`, and returns immediately without waiting for the termination of the thread.

- `void stop(bool sync=true);`

  This method requests the thread to stop. If `sync` is true, it blocks until the `run` method of the thread terminated.

- `Status getStatus() const;`

  This method returns the current execution status.

- `virtual std::string getThreadClassName() = 0;`

  This abstract method needs to return the name of the thread for debugging purposes.

- `virtual void run() = 0;`

  This abstract method needs to implement the actual computation of the thread. The code could look like this:

  ```
  while ( getStatus() == RUNNING ) {
  // do something
  }
  ```

- `virtual void stopRequested();`

  We encourage the use of condition variables to block threads instead of busy idling. The `stopRequested` method can be used to trigger these condition variables in case a user requests the thread to stop. It is executed when `stop` is called.

4.2.4 TF::Messages namespace

With the serialization and thread support we have all tools necessary to implement the message passing between nodes.

Current MPI implementations like MPICH2, LAM–MPI, and ScaMPI require a serialized MPI access in multi–threaded programs. That means that no two threads may access MPI at the same time. The easiest way to guarantee that is to relocate all MPI communication into one thread.

This thread delivers incoming and outgoing messages in the order they arrive, with the exception of system messages which have priority.

In case one module communicates with another module on the same computer, messages do not need to be serialized and sent over the network. Instead, it is sufficient to deliver the message internally.

Messages have to be derived from the `TFMessage` class in order to be sendable. They are serializable objects that contain fields which specify the sender's MPI rank, the receiver's MPI rank, and the modules targeted at the receiver. Each node can send messages to each node, including itself. The target module values can be defined by calculating the sum of a subset of these constants:

- `static const int WORKERMODULE = 0x1;`
- `static const int MASTERMODULE = 0x2;`
- `static const int AGGREGATORMODULE = 0x4;`
- `static const int APPLICATIONMODULE = 0x8;`

Apart from the `TFMessage` base class, the framework defines other base classes to reduce the amount of code that needs to be written for a specific application. These include the `TFObjectMessage` class for sending `TFSerializable` objects, `TFOneValueMessage` for sending atomic values like strings, and `TFKeyValueMessage` for sending key/value pairs of atomic types.

The `TFMessagingService` defines the interface for messaging services, which can be used to deliver messages. There is currently only one implementation that uses MPI for sending messages. The communication takes place in a single thread according to the simple schema shown in Algorithm 4.2.1.

As we cannot use two threads (one that performs a blocking wait for incoming messages, and one with a blocking wait for outgoing messages) due to MPI's restrictions,

ALGORITHM 4.2.1. Main loop of the Messaging Service.

```
1. While true do 2–9
2.    If outgoing message available then
3.       send message;
4.    If incoming message available then
5.       receive and dispatch message;
6.    If either was true then
7.       sleep very briefly;
8.    Else
9.       sleep longer;
```

we have to conduct permanent checks for new messages. However, this creates virtually no load on the processor and is yet very fast if the sleep times are very short.

Figure 4.2.4 illustrates the message flow from a master to a worker on another node. We use this to explain the steps conducted in message passing.

① In the first step, the `TFMaster` implementation creates a message and hands it over to the `TFMessagingService` for delivery. Figure 4.2.5 shows how messages can be sent. The code is important to know because not only the provided modules like the master and workers, but also the application specific implementations, can send messages.

② The `TFMessagingService` accepts the message and stores it into a FIFO queue for delivery.

③ Algorithm 4.2.1 checks in line 2 for outgoing messages.

④ Messages are serialized and then sent with MPI's `MPI_Isend` (line 3). Asynchronous sending is necessary to prevent deadlocks. If two nodes sent messages to each other with a blocking send at the same time, nobody could accept the data, and the processes would block.

`TFMessage` objects can be sent either as two MPI messages, the first containing the message length, the second containing the actual content, or as one message, where `MPI_Iprobe` determines the message length. Even in cases where the network is a bottleneck we cannot measure any differences in the performance.

⑤ The receiving message module checks frequently for incoming messages (line 4–5

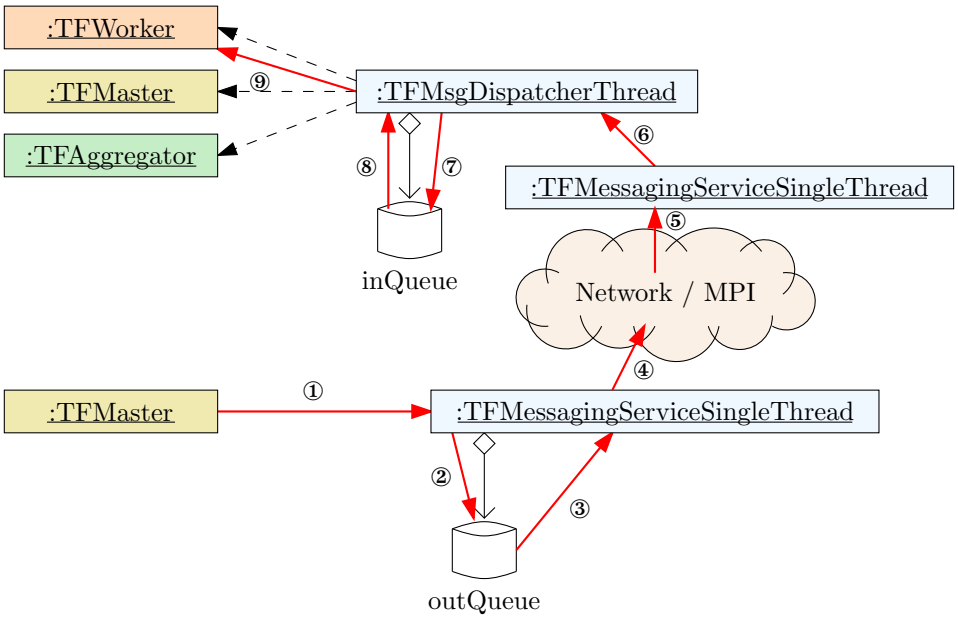


Figure 4.2.4: Message flow from a master to a worker on another node.

```
void sendSubQueryToWorker(
    std::auto_ptr<TF::Query::TFSubQuery> subQuery
  ) {
  TFCoordinator *c = TFCoordinator::getInstance();
  TFMessagingService *ms = c->getMessagingService();

  const int from = TFTopology::rank();
  const int to = subQuery->getTargetWorkerRank();

  auto_ptr<TFMessage> msg(
    new TFSubQueryMsg( subQuery, from, to, TFMessage::WORKERMODULE )
  );
  ms->deliverMessage(msg);
}
```

Figure 4.2.5: Source code for sending messages

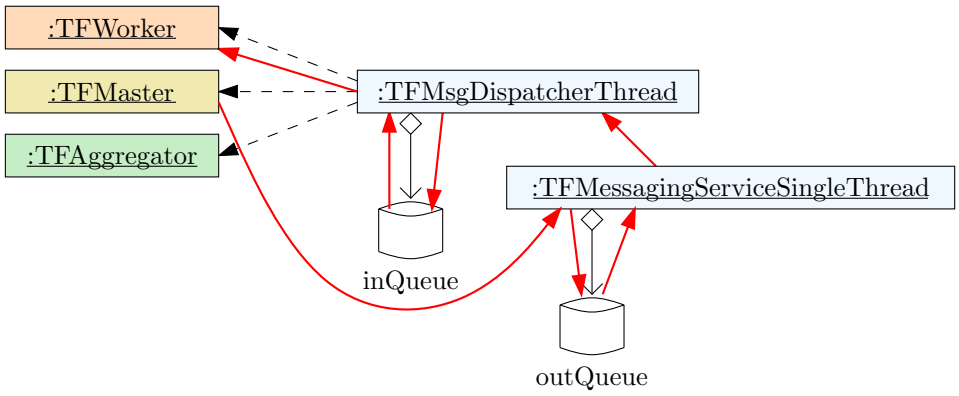


Figure 4.2.6: Message flow from a master to a worker on the same node.

of Algorithm 4.2.1) with `MPI_Iprobe`. This asynchronous check is necessary, because a blocking `MPI_Recv` would lead to deadlocks when each process is waiting for incoming messages.

If a message is available, the messaging service receives it with `MPI_Recv`.

⑥ After that a message is deserialized and passed to the `TFMsgDispatcherThread`.

⑦ The `TFMsgDispatcherThread` buffers incoming messages in a message queue to allow the messages to arrive faster than they can be handled by their respective target modules.

⑧ Messages in the queue are dispatched to their target in FIFO order by the `TFMsgDispatcherThread`, which is executed in a different thread than the `TFMessagingService`.

⑨ The dispatcher passes the messages to the target modules. If more than one module is addressed, a copy of the message is passed for each but the last target module, which receives the original message.

If messages are sent from one module to another module on the same worker, the message does not need to be serialized nor does it need to be sent with MPI. In step ④ we check whether the target is on the same node and skip to step ⑥ if possible. Figure 4.2.6 illustrates this.

4.2.5 TF::Query namespace

The `TF::Query` namespace defines several container classes, including `TFQuery`, `TFSubQuery` and `TFResult` for queries, subqueries, and results. Furthermore, it contains a query ID generator and the query queue discussed in Section 3.2.3.

The contents of the data containers for queries, subqueries and results have been illustrated already in Figure 3.2.2 on page 50. The only difference between Figure 3.2.2 and the implementation is that `TFSubQuery` is derived from `TFQuery`. This makes it easier for the programmer, if subqueries and queries have similar content. It is sufficient to write one class derived from `TFSubQuery` and to use it for sending queries as well. The additional data fields are simply ignored. We postpone the explanation of how to implement custom query classes to Section 4.3.3.

The `TFQueryQueue` interface deserves attention as it is slightly more complicated. The interface is defined as

```
template<class QueryType> class TFQueryQueue;
```

Currently, `template<class QueryType> class TFQueryQueueInMemory` is the only implementation of this interface, but it is easy to write query queues that store data in persistent memory.

All `TFQueryQueue` implementations have to be thread save and implement the following methods:

- `virtual void submitNewQuery(std::auto_ptr<QueryType> query);`
  Appends the query to the end of the query queue.
- `virtual std::auto_ptr<QueryType> requestNextQuery(bool blocking=true);`
  Returns the first element of the query queue. The behavior depends on the value of `blocking`. If set to false, the query queue will check whether it contains any queries and return either the first query or null. If `blocking` is true, the query queue blocks until a query is available for delivery. As this behavior would cause problems if a thread needs to be shutdown, while the query queue is blocking, we need a way to back out of the blocking wait. The function `stop` requests a query queue to stop, which unblocks all blocking requests and lets them return null. The user can use the function `isStopped` to check, whether such an even has happened.
- `virtual void stop();`

Terminates blocking waits.

- ```
  virtual bool isStopped() const;
  ```

  Returns whether a stop request was issued.

- ```
  virtual void waitForQuery() const;
  ```

  Waits until a query is deliverable but does not dequeue it.

- ```
  virtual std::list<QueryType*> requestMoreQueries(
     int max,
     const QueryType* currentQuery,
     SelectorType selector ) = 0;
  ```

  Extracts up to `max` queries from the query queue. Candidates have to be compatible to `currentQuery` and get selected by `selector`. A query is compatible to `currentQuery`, if it belongs to the same database, and no range update request was issued between `currentQuery` and the one inspected. A query `q` gets selected, if and only if

  ```
  selector(currentQuery, q)
  ```

  returns true.
  Selectors are function objects which comply to the following signature:

  ```
  typedef boost::function2<
     bool,
     const QueryType*, const QueryType*
     > SelectorType;
  ```

  That is, they accept two pointers to query objects, the first being `currentQuery`, and return either true or false.
  An example of a complying function is:

  ```
  template<class QueryType>
  bool anything(
     const QueryType *current,
     const QueryType *candidate) {
     return true;
  }
  ```

- ```
  virtual bool empty() const;
  ```

Returns whether the queue is currently empty.

- `virtual int size() const;`

  Returns the current length of the queue.

4.2.6 TF::Topology namespace

The `TF::Topology` namespace defines classes to generate and store topology information. This includes the relationship between masters, workers, and aggregators, as well as the individual responsibilities of the workers for database fragments.

The `TFTopology::init()` method, which is called by `TFInit()`, is responsible for generating the topology and distributing the information to all nodes.

Currently the method generates single–tiered topologies but it is easy to extend this to other topologies or to generate topologies based on the Network Weather Service [71].

The following configuration values determine the behavior of `TFTopology::init`:

- `TFConfig::getBool( "master␣is␣aggregator", true );`

  Defines whether the master node with rank 0 runs an aggregation module.

- `TFConfig::getBool( "master␣is␣worker", false );`

  Defines whether the master node with rank 0 runs a worker module.

- `TFConfig::getBool( "worker␣is␣aggregator", false);`

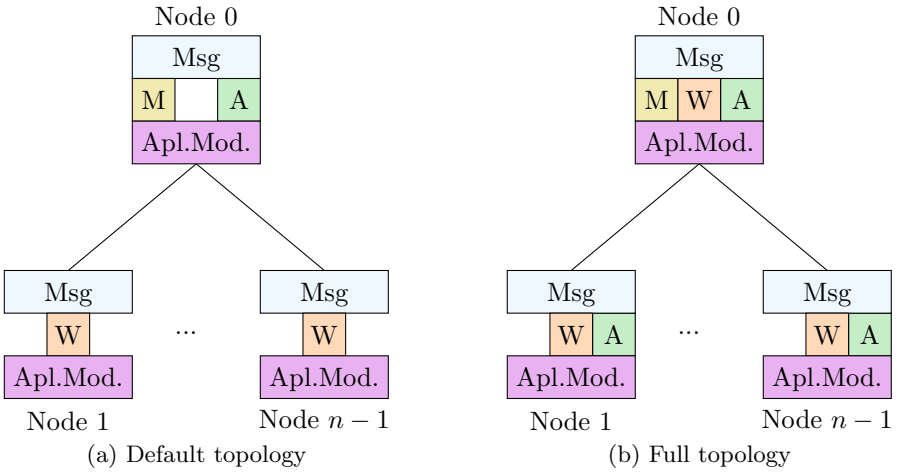
  Defines whether the worker nodes with rank unequal 0 run aggregation modules.

Figure 4.2.7a shows the default topology created by `TFTopology::init` if no special values are configured. The boxes show which modules (messaging service, master, worker, aggregator, application model) are started on the respective nodes.

If result messages are big and aggregation is difficult, a single aggregation node might turn into a bottleneck and it might be sensible to start additional aggregators on the workers as well. Conversely, the master node with rank 0 might be underutilized if the task distribution and aggregation creates little overhead. Therefore, we can start a worker on the first node as well. Such a topology is shown in Figure 4.2.7b.

As the topology is created according to a configuration file, it is easy to try and check which topology gives the best performance. The profiler can help with this task.

Once the topology is established, it can be accessed by the classes shown in Figure 4.2.8. The `TFTopology` class offers information about the node where it is residing.



(a) Default topology (b) Full topology

Figure 4.2.7: Topologies created by `TFTopology::init`

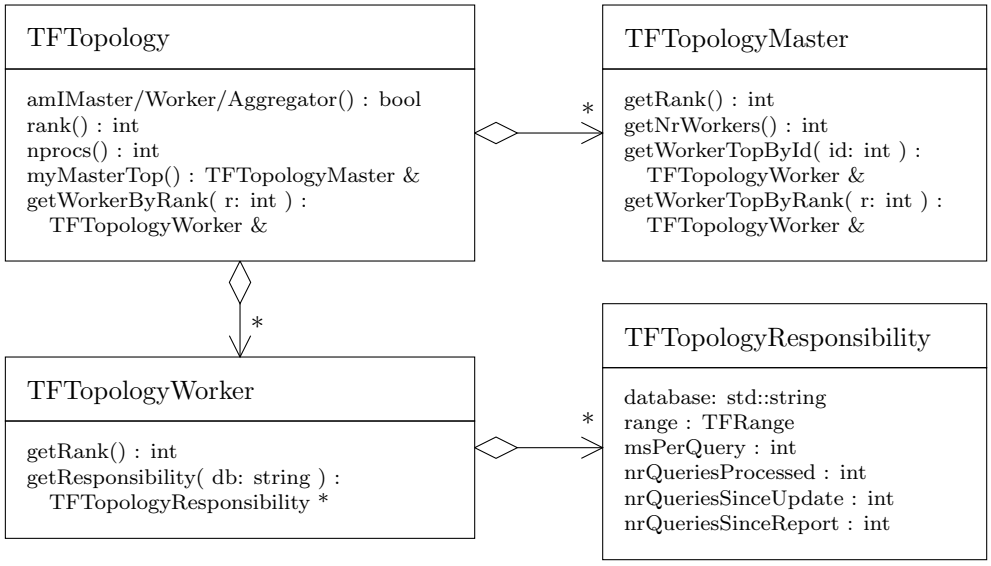


Figure 4.2.8: Topology class diagram

It can tell, whether this node is supposed to run a master, worker, or aggregator module. Furthermore, it can provide the program with information like the node's MPI rank and the number of nodes. Finally, it is a container for objects which store more details about this and other nodes.

The `myMasterTop()` method returns topology information about the node's master. If the current node is running a master module itself, the method delivers information about the this node. If the node is a worker, the method delivers information about the node's parent. The `TFTopologyMaster` class provides rank information and access to the node's workers. These can be accessed by their MPI rank or by an ID (the workers are numbered from 0 upwards).

The `TFTopologyWorker` stores pointers to `TFTopologyResponsibility` objects. Each node can have several responsibilities for various databases. The responsibility objects store the range assigned to the worker, average processing times of queries, and book keeping data about the last range updates and information connected to this.

4.2.7 TF::Coordination namespace

The `TF::Coordination` namespace contains only a class named `TFCoordinator` that serves two purposes. The first of which is to provide a template for the application startup, which instantiates and registers all modules. This is as simple as:

```
TFCoordinator *c = TFCoordinator::getInstance();
auto_ptr<TFApplicationModel>
  model(new MyApplicationModel(DATABASE_NAME));
// and more application models if desired
c->registerApplicationModel(model);
c->initAndStart(SOAP_PORT);
c->waitForTermination();
```

Details about the `TFApplicationModel` follow soon.

The second purpose of `TFCoordinator` is to provide access to the various module instances on a node. For that, it provides the following functions:

- `Master::TFMaster * getMaster() const;`
- `Worker::TFWorker * getWorker() const;`

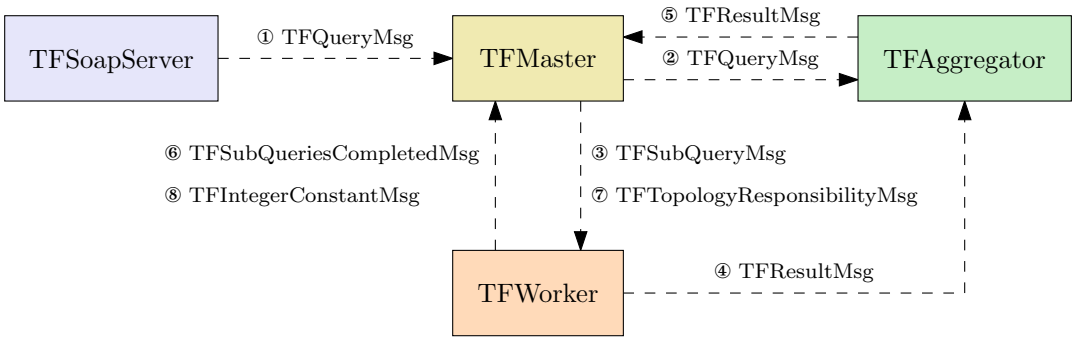


Figure 4.2.9: Message exchange

- `Aggregation::TFAggregator * getAggregator() const;`
- `Interfaces::TFSoapServer * getSoapServer() const;`
- `Messages::TFMessagingService * getMessagingService() const;`
- ```
  ApplicationModel::TFApplicationModel *
      getApplicationModel(std::string database) const;
  ```

The definitions of these interfaces appear in the following sections. Owing to the high degree of modularity each interface by itself is pretty simple.

4.2.8 TF::Master namespace

The definition of the `TFMaster` interface as in the `TF::Master` namespace is driven by the messages the master has to understand. The messages the master needs to send and receive are shown in Figure 4.2.9.

A master implementation like `TF::MasterStandard` gets its main functionality by implementing the following message handlers:

- ```
  virtual void handleNewQuery(
      std::auto_ptr<TF::Query::TFQuery> query);
  ```

  Accepts a new query and triggers a check whether new subqueries shall be sent to the workers. See paragraph “Query queues” of Section 3.2.2 on page 52 for details.
- ```
  virtual void handleNewResult(
      std::auto_ptr<TF::Query::TFResult> result );
  ```
  ```

Asks the application model to store the results and triggers a check whether new subqueries shall be sent to the workers.

- ```
  virtual void handleSubQueriesComplete(
      std::auto_ptr<TF::Query::TFSubQueriesCompleted> comp,
      int const workerRank );
  ```

  Triggers a check whether the database responsibilities need to be refined. See Section 3.3.3 on page 57 for details on the algorithm used.

- ```
  virtual void handleRangeUpdatedNotification( int workerRank );
  ```

  Notification that a worker has processed one request to update the responsibilities of a database fragment.

The `handleNewResult` is responsible for storing results and `getResult` allows retrieving results. Both functions delegate work to the application model.

The message handling can be implemented in two ways. Either messages are processed instantly after being received, or they are stored in a queue, and the `run` method of the master thread does the actual processing. The latter way has the advantage that the Master module does not block the message dispatcher.

4.2.9 TF::Worker namespace

The `TFWorker` interface defines only two message handlers that need to be implemented:

- ```
  virtual void handleNewSubQuery(
      std::auto_ptr<TF::Query::TFSubQuery> query );
  ```

  Accepts a query and puts it into the query queue.

- ```
  virtual void handleNewResponsibility(
      std::auto_ptr<TF::Topology::TFTopologyResponsibility> resp );
  ```

  Accepts a request to update the responsibilities and puts it into the query queue.

The actual processing of queries and responsibility updates happens in the main loop of the worker thread:

- ```
  virtual void run() = 0;
  ```

  Grabs queued requests and delegates them to the application model. Queries can be reordered as described in Section 3.2.3 in order to speedup processing. The restrictions, the reordering must comply with, are completely transparent for the programmer.
  ```

Apart from delegating work to the application model, the worker class is responsible of measuring the processing times and reporting them to the master. The programmer does not need to take care of that.

4.2.10 TF::Aggregation namespace

The `TFAggregator` specifies two message handlers that need to be implemented:

- ```
  virtual void handleNewResult(
      std::auto_ptr<TF::Query::TFResult> result );
  ```
  Aggregates a subresult to the final result and sends it to the master if all subresults have been aggregated. The actual aggregation is delegated to the application model and may need information from the `TFQuery` object like formatting options. A `TFResult` has to be buffered if no `TFQuery` object of that query has arrived yet. Otherwise it can be processed instantly.

- ```
  virtual void handleNewQuery(
      std::auto_ptr<TF::Query::TFQuery> query );
  ```
  Stores the query object as it is needed for the result aggregation. If results have arrived before, this function calls `handleNewResult` for each previously arrived result. The query object is deallocated when all subresults are processed.

In order to implement one of the more advanced aggregation strategies described in Section 3.1.3, one can simply derive an aggregation module that handles custom message types in the `handleOther` method.

4.2.11 TF::Interfaces namespace

The `TF::Interfaces` namespace defines currently only a SOAP interface with few methods in the class `TFSoapServer`. It is should be easy but not necessary to adapt this interface to the needs of specific applications.

- ```
  class ns__QueryStatusObj {
      QueryStatusObj();
      char * queryID;
      char * result;
  };
  ```
  Defines a container that describes the status of a query.

- ```
  int ns__submitQuery(char* database, char* queryDescription,
      ns__QueryStatusObj& result);
  ```

Submits a query in textual format for processing to the framework and sets the `queryID` field of the result to the ID assigned to this query. This function asks the application model to translate the textual query to a `TFQuery` and enqueues it for processing. Of course, the textual query description can contain an ID of a database record that describes the actual query.

- ```
  int ns__queryStatus(char* queryID, char* database,
      ns__QueryStatusObj& result);
  ```

  Returns the query status of the given query by populating the `result` field of the `ns__QueryStatusObj` object. The result string is requested from the application model and can be `"unknown"` for example if no result has not arrived yet. Of course, the result string can contain an ID of a database record that stores the actual result.

- ```
  int ns__shutdownServer(char* shutdownToken,
      int minNumberProcessedQueries, int& result);
  ```

  Requests the server to shut down. When the SOAP server starts up, it creates a file `shutdowntoken` with a very big random number. Knowing this number is considered the authorization to shutdown the service. Unless the master has processed the requested number of queries, this request is simply ignored. The result is 1 in case of a non–matching shutdown token or 0 otherwise.

- ```
  int ns__allDBFragmentsLoaded(int& result);
  ```

  Returns 1 if all clients have loaded all database fragments assigned to them. This is useful for benchmarking.

In order to extend the SOAP interface, one needs to edit the file `SoapTaskFarmer.h` in `src/soap/`, run `generate`, and implement the functions in `TFSoapServer.cpp`.

4.2.12 TF::ApplicationModel namespace

While the previously discussed classes build up the framework that steers the distribution and collection of queries and results, the actual processing of queries is application dependent. The `TFApplicationModel` class defines the interface that needs to be implemented in order to do the processing. One `TFApplicationModel` object needs to be registered at each node for each database.

The following list describes the methods defined for the `TFApplicationModel`. An example of how to implement them follows in the next chapter.
  ```

- ```
  virtual std::auto_ptr<Query::TFQuery>
      queryDescriptionToQuery( char const *description );
  ```
  Called by the SOAP server, this function parses the textual description of a query and returns a query object that is derived from the `TFQuery` class or 0 if the description was not syntactically correct. The programmer has to specify a syntax of queries that is suitable for her or his project.

- ```
  virtual std::vector<Query::TFSubQuery *>
      breakQuery( const Query::TFQuery *query );
  ```
  Called by the SOAP server, this method breaks the query object that was generated before into subqueries for the workers, and sets the `range` and `targetWorkerRank` of these subqueries. The framework will take care of the delivery of these subqueries and deallocate the objects.

- ```
  virtual void processQuery(
      std::auto_ptr<Query::TFSubQuery> query,
      Query::TFQueryQueue<Query::TFSubQuery> & queryQueue,
      std::vector<std::string> & processedIDs,
      Worker::TFWorker & worker );
  ```
  Called by the worker, this method processes the query and sends the results to the aggregator with the rank specified in the `query` object. The `worker` object provides the `sendResult` method for sending the results. Using this method is mandatory for sending results to the aggregator.

  The `queryQueue` allows prefetching queries from the queue in order to process similar queries concurrently.

  The IDs of all processed subqueries must be added to the `processedIDs` vector.

- ```
  virtual void aggregationQueryNotification(
      std::auto_ptr<Query::TFQuery> query );
  ```
  This function is called by the aggregator to inform the application model of a query for which it has to aggregate subresults soon.

- ```
  virtual std::auto_ptr<Query::TFResult>
      aggregateResult( std::auto_ptr<Query::TFResult> subresult );
  ```
  This function is called by the aggregator after `aggregationQueryNotification` and requests the application model to aggregate the subresult. If and only if subresults from all workers have arrived, the application model has to return the final result. Otherwise it has to return 0. The framework takes care of sending the final result to the master.
  ```

- `virtual Query::TFRange getDBRange() const;`
  This function is called by the master to query the range of record IDs in the database. It starts at 0 and ends with the last record. This function can be called often and has to be fast.

- `virtual void setActiveDBRange( Query::TFRange const& range );`
  This function is called by a worker, to ask the application model to load the specified range into memory. Following queries will be about this range.

- `virtual void storeResult( std::auto_ptr<Query::TFResult> result );`
  This function is called by the master and asks the application model to store the query, so that it can be retrieved by `getResult` later.

- `virtual std::string getResult( std::string const & queryID );`
  This function is called by the SOAP server and asks the application model to return a textual result of the query. This textual representation can be formatted so that it is easy to parse in the application which submitted the query. The reason for submitting results as text is to take away the task of writing an XML Schema from the application developer. The result can be stored in a database as well. In this case, the result string would contain an ID of the database record.

- ```
  virtual void receiveMessage(
      std::auto_ptr<Messages::TFMessage> msg);
  ```
  This function can but does not need to be implemented. It is called if anyone sends a message to the application module. This might be necessary for special customizations of the framework like an automatic distributing of files during application startup.

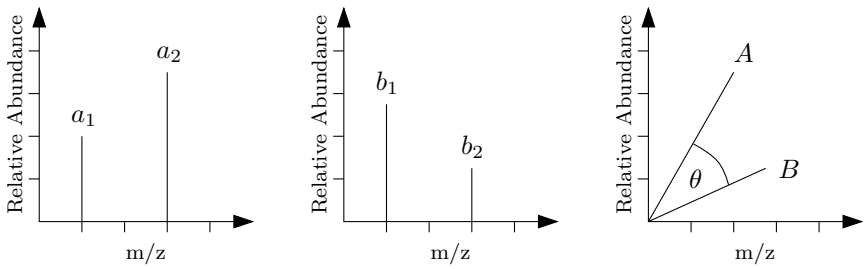
4.3 Sample Application (Spectral Comparison)

This section gives an example of how to use the framework for a mass spectral database.

4.3.1 Problem

The problem of spectral comparison has been mentioned in Section 1.1.3 already.

Dan E. Krane's book "Fundamental Concepts of Bioinformatics" [38] gives an overview about the topic of mass spectrometry. It is primarily used by analytical chemists for protein identification [38]. After separating proteins, they are digested,
  ```



(a) Spectrum A (b) Spectrum B (c) Contrast Angle

Figure 4.3.1: Spectral Contrast Angle for spectra with identical peak positions [69]

i.e. broken into smaller peptides (chains of amino acids) and electrically charged. An electric field accelerates the peptides towards a collector. On their path to the collector, the electrically charged peptides have to pass a strong magnetic field, which bends their flight paths. This allows distinguishing the mass–over–charge ratios (m/z) of the peptides. The collector can measure the intensities of incoming peptides. After a mathematical noise reduction, the mass spectrometer produces an image like Figure 1.1.3 on page 5. A detailed explanation of mass spectrometry can be found in "Protein Sequencing and Identification Using Tandem Mass Spectrometry" [37].

The problem of spectral comparison is to project the difference between two spectra to real numbers. As the real numbers are a totally ordered set, we can use the projection to order mass spectra by their degree of similarity to one target spectrum and thereby find the best matching, i.e. most similar, spectra of a database.

4.3.2 Algorithm

The algorithm used in this implementation for determining the similarity of two spectra is based on the spectral contrast index, described by Wan, et al. [69]. The purpose of this chapter is certainly not to do research on optimal similarity measures, but to demonstrate the features of the framework and how easy it is to employ it for custom search algorithms.

More advanced algorithms like the kernel trick [26], Spectral Convolution [56, 57], and k–mutation Spectral Similarity [56, 57] exist, but the simplicity of the spectral contrast index, both in terms of implementing and in terms of understanding it, justifies our choice.

The basic idea of spectral contrast angles is to consider a spectrum as a vector in a

multidimensional space. Figure 4.3.1 illustrates this with two peaks. We assume for now that the positions of corresponding peaks are identical in the spectra which we compare. Spectrum A contains two peaks with relative abundance a_1 and a_2. The relative abundance of these two peaks defines the multidimensional vector A. In this example, the vector is only two dimensional and has the coordinates (a_1, a_2). Vector B is defined likewise.

We can use the angle θ as a measure of similarity between the two spectra. For identical spectra, the angle is $0°$, and it can grow up to $90°$ for very dissimilar spectra. It cannot grow beyond $90°$ because all points lie in the first orthant, i.e. their coordinates are all positive. We know that the dot product between two vectors $A = (a_i)$ and $B = (b_i)$ is defined as

$$\langle A, B\rangle = |A| \cdot |B| \cdot \cos\theta \tag{4.3.1}$$

and

$$\langle A, B\rangle = \sum_i a_i b_i. \tag{4.3.2}$$

From this it is easy to conclude that

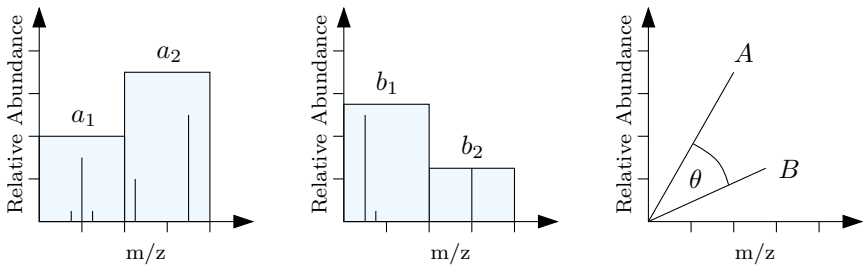
$$\cos\theta = \frac{\sum_i a_i b_i}{|A| \cdot |B|} = \frac{\sum_i a_i b_i}{\sqrt{\sum_i a_i} \cdot \sqrt{\sum_i b_i}}. \tag{4.3.3}$$

As the values of θ are in $[0, 90]$, the values of $\cos\theta$ are in $[0, 1]$. The smaller the angle, the bigger is the $\cos\theta$ value. We can use this to measure the similarity between spectra.

So far, we have assumed that the two spectra we compare have peaks at the same m/z positions. This is no valid assumption for a comparison of arbitrary mass spectra. We cannot even assume identical m/z positions when comparing two different mass spectra of one single peptide as waterloss, different isotopes, and measurement uncertainty can create peaks in one spectrum that do not appear in the other spectrum.

Aggregating several peaks in close proximity to bins allows compensating for small differences in peak positions. Figure 4.3.2 shows two bins in each diagram. The height of a bin, i.e. its value, is determined by the sum of all peaks that are covered by the bin range. It is easy to find better algorithms that prevent the hard gaps between bins, but this is beyond the purpose of this implementation.

In order to find the best match to a mass spectrum, one has to iterate over all spectra in the database and compare each spectrum to the query spectrum. The best match



(a) Spectrum A (b) Spectrum B (c) Contrast Angle

Figure 4.3.2: Spectral Contrast Angle for spectra with different peak positions

is the spectrum with the highest $\cos\theta$ value. It is obvious that this algorithm is well suited for being executed in parallel. We can split the database into fragments and assign different fragments to the workers. They compare the spectra of the fragments to the query spectrum and deliver the best local result. In the aggregation phase, we can select the best local result and deliver it as the globally best match.

The following chapter illustrates the steps necessary to integrate the algorithm into the framework.

4.3.3 Implementation

First, we need to implement queries, subqueries, and results. Because the subquery class is derived from the query class, we can define a domain specific subquery that contains all information of queries and subqueries. Therefore, it is sufficient to implement a single class.

The following code defines a subquery with all necessary functions. The payload in this case is a `SCSpectrum`, which implements the serializable interface.

```
class SCSubQuery : public TF::Query::TFSubQuery
{
public:
    SCSubQuery();
    SCSubQuery(const SCSubQuery& that);
    virtual ~SCSubQuery();
    virtual TFSerializable* createNewInstance() const;
    virtual Query::TFSubQuery* clone() const;
    virtual const std::string& getClassName() const;
```

```
    virtual void deserialize(
        Serialization::TFSerializer& serializer);
    virtual void serialize(
        Serialization::TFSerializer& serializer) const;
    static const std::string className;
    SCSpectrum<float> spectrum;
};
```

As the `TFSubQuery` is derived indirectly from `TFSerializable` we can consider this as an example of how to implement serializable classes as well.

The following code implements the `SCSubQuery` class:

```
#include "TFSerializerRegistry.h"

const std::string SCSubQuery::className("SCSubQuery");
TFSERIALIZER_REGISTER(SCSubQuery);

SCSubQuery::SCSubQuery() : TF::Query::TFSubQuery(),
spectrum("", 0, 0, SCSpectrum<float>::ListType()) { }

SCSubQuery::SCSubQuery(const SCSubQuery& that)
: TF::Query::TFSubQuery(that), spectrum(that.spectrum) { }

SCSubQuery::~SCSubQuery() {}

TF::Query::TFSubQuery * SCSubQuery::clone() const
{ return new SCSubQuery(*this); }

TF::Serialization::TFSerializable*
    SCSubQuery::createNewInstance() const
{ return new SCSubQuery(); }

const std::string& SCSubQuery::getClassName() const
{ return className; }

void SCSubQuery::deserialize(
```

```
    Serialization::TFSerializer& serializer ) {
        TFSubQuery::deserialize(serializer);
        spectrum = SCSpectrum<float>::deserialize(serializer);
}

void SCSubQuery::serialize(
    Serialization::TFSerializer& serializer) const {
        TFSubQuery::serialize(serializer);
        spectrum.serialize(serializer);
}
```

The query class can be generated by

```
typedef SCSubQuery SCQuery;
```

The result class looks virtually the same except that it has a different payload and is derived from `TFResult`. For that reason, we do not repeat it at this place.

This is all we need in order to send queries, subqueries and results between nodes. The next step is to define the application model:

```
class SCApplicationModel
    : public TF::ApplicationModel::TFApplicationModel {
protected:
    std::map< std::string, int >
        nrRemainingResults;
    std::map< std::string, TF::Examples::SC::SCResult * >
        preliminaryResults;
    std::map< std::string, TF::Examples::SC::SCResult * >
        finalResults;
    SCDatabase<float> database;
public:
    SCApplicationModel(std::string const & databaseName);
    ~SCApplicationModel();
    // all virtual methods of TFApplicationModel follow here
};
```

The `nrRemainingResults` field maps query IDs to the number of subresults to be received from the workers. It is used only by aggregators. The `preliminaryResults` and

`finalResults` fields hold results that are currently being assembled in the aggregator or stored on the master node for the SOAP server.

The only remaining part of the search application for mass spectra is the implementation of the methods defined in the application model. These can look as follows:

```
SCApplicationModel::SCApplicationModel(
    std::string const & databaseName)
    : TFApplicationModel(databaseName) {
    database.readDatabase(databaseName);
}
```

The constructor initializes the index of the database and stores the name of the database. The user can query the name later with the `getDatabaseName()` method of the `TFApplicationModel`.

The transformation of a textual query representation to an internal query representation is very application specific and shall be omitted here for that reason. The structure looks like this:

```
std::auto_ptr<Query::TFQuery>
SCApplicationModel::queryDescriptionToQuery(char const *description)
{
    auto_ptr<SCQuery> q( new SCQuery() );
    // fill only application specific fields of q
    return std::auto_ptr<Query::TFQuery>(q);
}
```

Breaking the query into subqueries is more interesting:

```
std::vector<TF::Query::TFSubQuery *>
SCApplicationModel::breakQuery( const TF::Query::TFQuery *_query ) {

    const SCQuery *query = dynamic_cast<const SCQuery *>(_query);
    TFTopologyMaster & topology = TFTopology::myMasterTop();
    int nrWorkers = topology.getNrWorkers();

    vector<TFSubQuery *> result;
    result.reserve(nrWorkers);
```

```
    for ( int i=0; i<nrWorkers; ++i ) {
        SCSubQuery *sq = new SCSubQuery();
        sq->setTargetWorkerRank(
            topology.getWorkerTopById(i).getRank() );
        sq->setRange( topology.getWorkerTopById(i).getResponsibility(
            query->getDatabaseName() )->getRange() );
        sq->spectrum = query->spectrum;
        result.push_back(sq);
    }

    return result;
}
```

This example shows how to get information about the workers from the topology. The framework provides the number of workers and their ranges of responsibility.

We see furthermore that we are in charge of populating the target worker rank as well as the ranges assigned to the workers.

The next callback happens on the workers to process the queries. The following example shows how to prefetch 5 additional queries from the query queue if available. This mitigates thrashing in case the application handles several different databases.

```
void SCApplicationModel::processQuery(
    std::auto_ptr<TF::Query::TFSubQuery> _query,
    TF::Query::TFQueryQueue<TF::Query::TFSubQuery> & queryQueue,
    std::vector<std::string> & processedIDs,
    TF::Worker::TFWorker & worker) {

    // this is where we put all queries to process:
    std::list<TF::Query::TFSubQuery *> queries;
    queries = queryQueue.requestMoreQueries(
        5, _query.get(), TF::Query::anything<TF::Query::TFSubQuery> );
    queries.push_front(_query.release());

    std::list<TF::Query::TFSubQuery *>::iterator i;
    for ( i=queries.begin(); i!=queries.end(); ++i ) {
        auto_ptr<SCSubQuery> query(dynamic_cast<SCSubQuery*>(*i));
```

```
        *i = 0;

        auto_ptr<SCResult> result(new SCResult());
        result->match = -1000000000;

        for ( int x=query->getRange().begin();
                x<=query->getRange().end(); ++x ) {
                SCDatabaseIndex<float> & indexEntry =
                    database.getEntries()[x];
                double value = indexEntry.getSpectrum().compareTo(
                    query->spectrum, 10,
                    SCSpectrum<float>::SpectralContrastAngle );
                if ( value > result->match ) {
                        result->bestIndex = x;
                        result->match = value;
                }
        }
        worker.sendResult(
            auto_ptr<TF::Query::TFResult>( result ), query.get() );

        processedIDs.push_back( query->getQueryID() );
    }
}
```

This code shows how a worker processes the query passed as the first parameter, plus up to five additional queries from the query queue. We do not impose any restrictions on the selection of additional queries by choosing the `anything` filter, except to choose only queries from the same database that have arrived before any responsibility update requests for that database. See Section 4.2.5 for details.

Once the worker has gathered the queries to analyze, the application specific processing is performed. The results are sent to the aggregator using the `sendResult` method of the worker. This is necessary, because the worker module populates several values of the result message, so that the programmer does not need to care about them.

Finally, the IDs of all queries processed get stored into the `processedIDs` vector.

Before the aggregator module passes any results to the application module, it waits

for the arrival of the query object from the master and hands it over to the application module. This can decide to store the query object for the aggregation process or to discard it. The following example code uses the notification to initialize a field that stores the number of subresults we are expecting for the query.

```
void SCApplicationModel::aggregationQueryNotification(
    std::auto_ptr<TF::Query::TFQuery> query ) {

    nrRemainingResults[ query->getQueryID() ] =
        TFTopology::myMasterTop().getNrWorkers();
}
```

The next step is the aggregation of partial results:

```
std::auto_ptr<TF::Query::TFResult>
SCApplicationModel::aggregateResult(
    std::auto_ptr<TF::Query::TFResult> result ) {

    SCResult * partialResult =
        dynamic_cast<SCResult *>(result.get());
    const std::string queryID = partialResult->getQueryID();

    SCResult *prelimResult = 0;
    if ( preliminaryResults.find(queryID) ==
         preliminaryResults.end() ) {
        // create new result entry
        prelimResult = dynamic_cast<SCResult *>(result.relese());
    } else {
        // update existing result entry
        prelimResult = preliminaryResults[queryID];
        if ( partialResult->match > prelimResult->match ) {
            prelimResult->match = partialResult->match;
            prelimResult->bestIndex = partialResult->bestIndex;
        }
    }
    preliminaryResults[queryID] = prelimResult;
```

```
    if ( (--nrRemainingResults[queryID]) == 0 ) {
        // we are done -> send result and clean up
        preliminaryResults.erase(queryID);
        nrRemainingResults.erase(queryID);
        return std::auto_ptr<TF::Query::TFResult>(prelimResult);
    }
    return std::auto_ptr<TF::Query::TFResult>(0);
}
```

The procedure is pretty simple. First, it checks whether this is the first subresult for the query. If this is true, the new subresult can be stored as the currently best one. Otherwise, it needs to be compared against the currently best subresult. Finally the procedure stores the result and checks whether the most recent subresult was the last one to arrive. In this case, it returns the final result.

All remaining functions are very simple. Examples for the two database related methods are:

```
Query::TFRange SCApplicationModel::getDBRange() const {
    int nrEntries = database.getEntries().size();
    return Query::TFRange(0, nrEntries-1);
}

void SCApplicationModel::setActiveDBRange(
        Query::TFRange const& range) {
    database.loadRange( range );
}
```

Storing and delivering results is implemented like this:

```
void SCApplicationModel::storeResult(
        std::auto_ptr<TF::Query::TFResult> result ) {
    std::string queryID = result->getQueryID();
    finalResults[queryID] =
        dynamic_cast<SCResult *>(result.release());
}

std::string SCApplicationModel::getResult(
```

```
        std::string const & queryID) {
    if ( finalResults.find(queryID) == finalResults.end() )
            return "unknown";

    SCResult *result = finalResults[queryID];
    int bestIndex = result->bestIndex;
    float match = result->match;
    std::string name =
        database.getEntries()[bestIndex].getIdentifier();
    return str( format("finished\n%1%\n%2%\n%3%")
                  % name % bestIndex % match );
}
```

We have considered the database a black box so far and will not go into the details as this is very application specific and can be implemented in many ways. The example implementation assumes that the database can be found either on a shared network drive or on local disks. In order to allow for fast loading of data, the database is stored in binary format, which is smaller and reduces the parsing time. A pregenerated file index allows for determining the offset of each record in the binary database file which makes it easy seek to records if fractions of the database need to be loaded into memory.

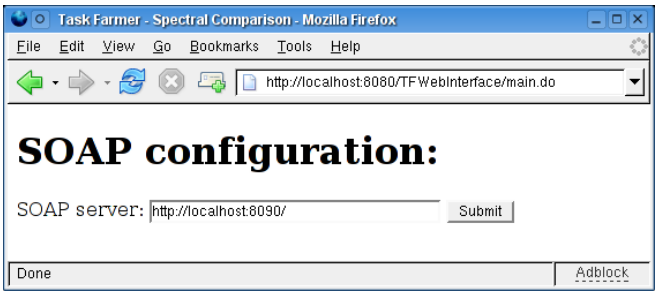
4.3.4 Web application

The purpose of the web application is to show that external applications can access the framework easily over SOAP. The web application is written in Java for the Struts framework. There is currently no database backend to store queries or results, as this is only a proof of concept implementation. We use the session to store data instead.

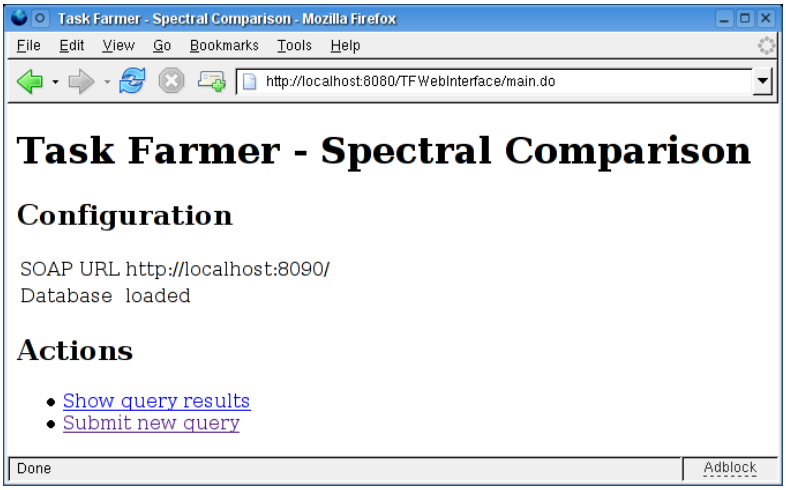
In order to access SOAP services with Java, we can use the Apache Axis library. This creates stubs to call the methods from the `.wsdl`[2] files generated by gSOAP.

The web interface is pretty simple and shown in Figure 4.3.3 and 4.3.4. First, the user is asked to specify the URL of the SOAP server. Then, he or she can submit a query and gets forwarded to a page that claims that the result is not available, yet. Refreshing this page shows the result as soon as the query got processed. The graph shown displays the query and the best match overlaid. As we can see, the query was not found in the database in the example.

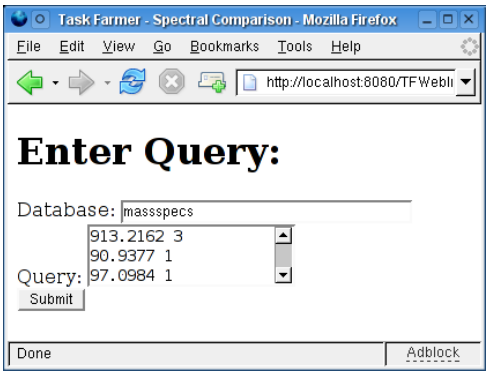
[2]Web Service Definition Language



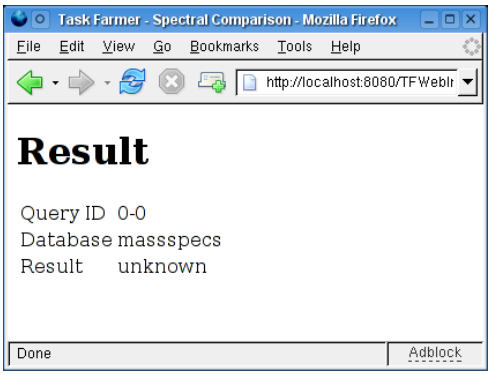
(a) Specifying SOAP URL



(b) Main menu



(c) Entering a query in Mascot DTA format



(d) Initially the result is unknown

Figure 4.3.3: Web interface

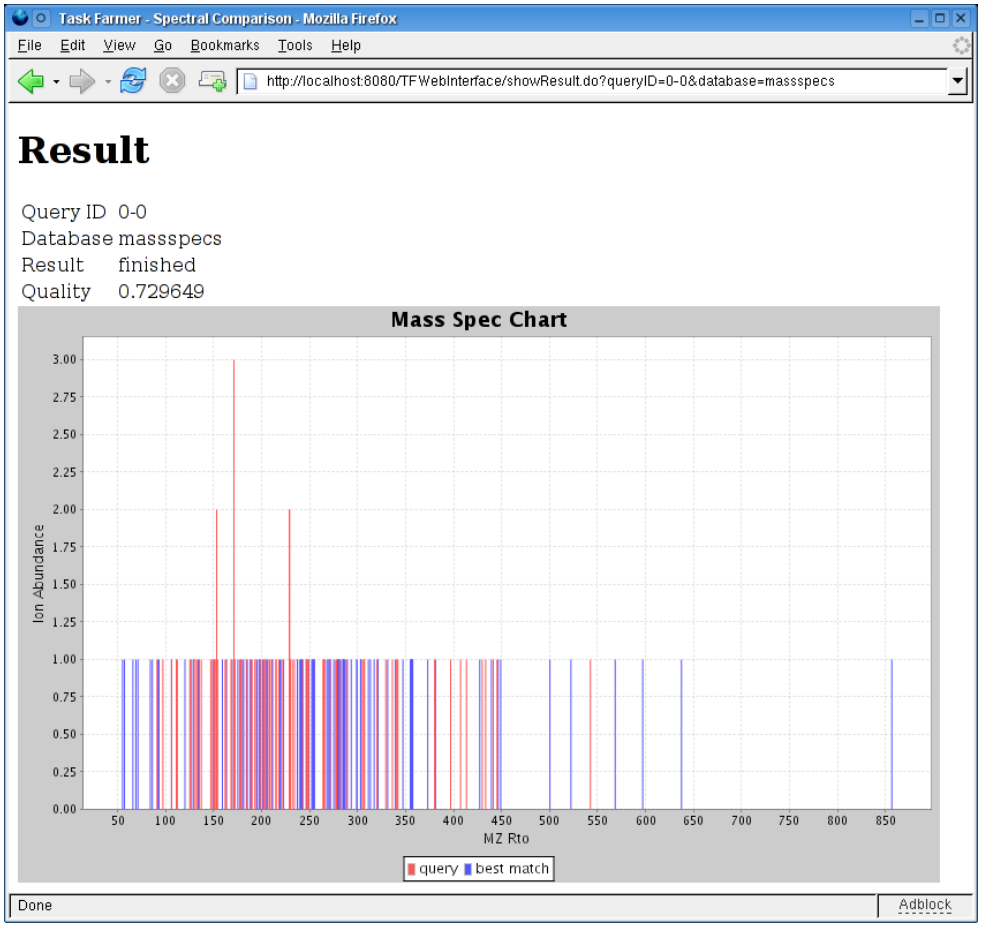


Figure 4.3.4: Web interface II

The following code shows that almost all SOAP details are hidden behind the generated SOAP stubs. Therefore, it is very easy to employ the framework in virtually any kind of front–end.

```
NewQueryForm newQueryForm = (NewQueryForm) form;
StringHolder queryID = new StringHolder();
StringHolder submissionTime = new StringHolder();
StringHolder expectedCompletion = new StringHolder();
StringHolder result = new StringHolder();
try {
    String soapUrl =
        (String) request.getSession().getAttribute("SOAPURL");
    SoapTaskFarmerPortTypeProxy proxy =
        new SoapTaskFarmerPortTypeProxy(soapUrl);
    proxy.submitQuery(
        newQueryForm.getDatabase(), newQueryForm.getQuery(),
        queryID, submissionTime, expectedCompletion, result );
    // new query ID is saved in queryID.value
} catch ( RemoteException e ) {
    // ...
}
```

4.4 Installation and dependencies

As mentioned before, the framework needs several external libraries. These are:

- pthreads, included in all UNIX distributions
- boost, `http://www.boost.org/`
- log4cxx, `http://logging.apache.org/log4cxx/`
- a multi–threading capable MPI implementation like MPICH2, `http://www-unix.mcs.anl.gov/mpi/mpich2/`

Boost and log4cxx can be compiled and installed with default arguments. For MPICH2 it is important to activate the `--enable-threads=serialized` switch when calling `configure`.

The steps of building the task farming framework are

```
make -f Makefile.cvs
./configure
make
```

The `configure` script checks which MPI compilers are available and decides whether to use the MPI 1 or MPI 2 syntax for the application initialization. ScaMPI for example uses the MPI 1 `MPI_Init` command despite being thread safe. For MPICH, we need MPICH 2 with `MPI_Init_thread`.

In order to compile a self–written application that uses the framework, one has to provide the include files of the `mpitf` directory and link against the following libraries:

- `mpitf`
- `boost_thread`
- `boost_filesystem`
- `log4cxx`

5 Analysis

All results of this analysis originate from benchmarks on the PSC2 cluster of the University of Paderborn, Germany. The specifications of this cluster are:

- Fujitsu Siemens Computers hpcLine
- 96 Primergy server nodes, each with
  - 2 Intel Pentium-III, 850 MHz,
  - 512 MByte memory
  - 4 GByte local hard disk
- 96 Dolphin PCI/SCI interfaces, 500 MByte/s SCI-Link bandwidth
- nodes are connected as a 12 x 8 2D torus with distributed switches
- PVFS based file server for the database
- Linux 2.4.7
- GCC 3.4.4
- ScaMPI

5.1 Methodologies and Tools

A set of logging facilities and report generation tools have been written in order to analyze the performance of the framework. These tools can be divided into three categories:

1. Query submission, a tool which submits queries to the search engine,
2. Query processing and logging, the search engine itself which processes queries and logs events for later analysis,
3. and finally one tool which controls the execution and generates reports.

We will discuss these parts now briefly.

5.1.1 Query submission

We can consider two different ways of submitting queries to the framework.

In *all–at–once* mode, the query submitter waits until the framework has loaded all database fragments on all workers. When they are loaded, the query submitter sends all queries at once. This shows the total throughput of the application and simulates the way mpiBLAST works, as mpiBLAST does not support a persistent servicing mode. We disregard the time of loading the database, as the relative fraction of the total runtime decreases the more queries we submit.

In *triggered benchmark* mode, we define epochs. In each epoch, queries are sent at a homogeneous rate. This way, we can simulate the usage of the application in a web environment, where a permanent flow of queries arrives. During peak times, the number of queries per time unit can be significantly higher than at night. If the epochs are long enough, the framework should arrive at a steady state, unless it receives more queries than it can process. Interesting aspects of this benchmark include the response times in inhomogeneous systems where some nodes are exposed to a higher load than others.

5.1.2 Logging

Logging events on many nodes is complicated as their clocks are usually not synchronized to a margin of a second or less. Instead of that, we use a common event as time 0. This event is the reception of the first query. Unless subqueries are very big, this event should be very close in time on all nodes. The PSC cluster has latency times in the range of $225\mu s$.

The workers log for each subquery the time of completion, relative to the reception of the first query, and the duration to process the subquery. The master logs the time of reception and completion of each query and the times and ranges of responsibility updates. Further, each node logs the state of its threads.

We can inspect now Figure 4.2.1 on page 71 again. The profile shows 4 different MPI processes indicated by the white and gray background. Each MPI process consists of several threads. In the example, node 0 runs a messaging service, a master, an aggregator, and a worker. The state of these processes is shown as a function of the time. The profile shows roughly 35 seconds of the execution.

At the very beginning, the nodes read the database into memory. In the example, the MPI processes read iteratively, because all MPI porcesses were executed on the same computer. We see that after the initialization phase all workers are constantly busy processing queries. We can further see that the messaging services of the nodes

are hardly used. Therefore, the network was no limiting factor in this example; in fact it was the CPU.

Node 0 contains a master, a worker, and an aggregator. Yet, we see that its worker finished at the same time as the other workers. This means, that the CPU consumption of the master and aggregator module was negligible.

5.1.3 Execution control and report generation

Two perl scripts `allatonce.benchmark` and `triggered.benchmark` control the execution of benchmarks.

The all–at–once benchmark is configured by a script named `allatoncescript`. Its format is very simple; each line contains two integers. The first integer describes the number of queries to send, whose type is determined by the second number. The interpretation of query types is task of the `consoleclient` application which issues the queries.

The triggered benchmark is configured by a script named `triggerscript`. Each line describes an epoch and contains one integer and one real number. The first number determines the length of the epoch in milliseconds, the second number the frequency in Hertz with which queries are sent. This file is read and executed by the `consoleclient`.

The benchmark scripts support the execution of several iterations in order to determine the variances between the values measured. They further support the execution of scripts with different numbers of processors. These values can be defined with the `-c` (CPU number) and `-i` (iteration) parameters. A setting `-c 1,2,5,10 -i 3` runs three iterations with each 1, 2, 5, and 10 MPI processes.

Apart from starting the framework and the query submission, the benchmark scripts analyze the log files of the framework and generate several reports (usually in form of gnuplot scripts). We will describe these reports when we encounter them the first time.

In the upcoming chapters we will analyze several aspects of homogeneous environments with one database, the refinement of responsibilities in inhomogeneous environments, and the use of multiple databases.

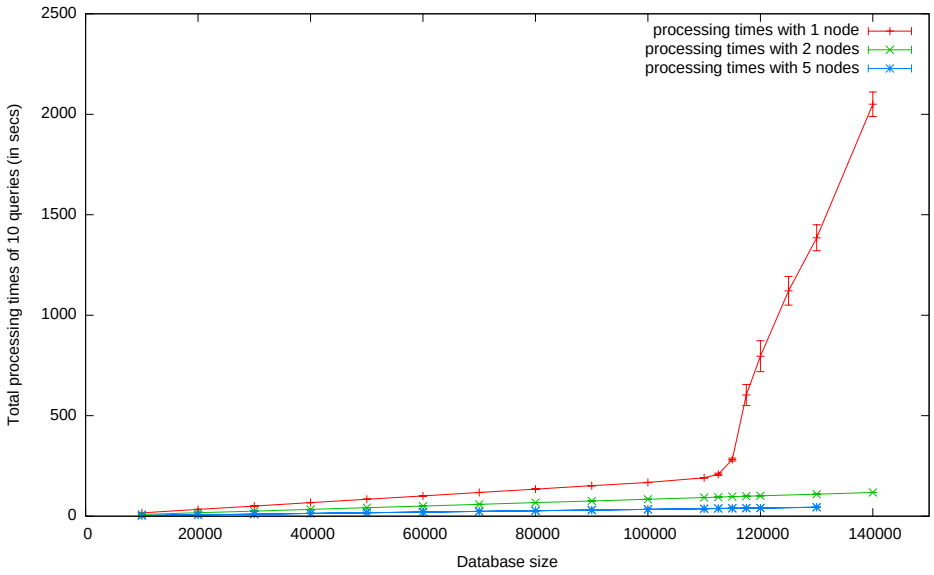


Figure 5.2.1: Thrashing effect

5.2 Homogeneous Environments with One Database

The analysis begins with homogeneous environments with one database as these represent the case which is probably encountered most often.

5.2.1 Thrashing effects

First, we want to verify the initial assumption that *the trashing effect has a huge impact on the performance of database searches if the databases exceed the RAM of a node.* For that, we have sent 10 queries with small spectra to databases of different sizes. Each experiment was executed three times. The average as well as the standard deviation are shown in Figure 5.2.1. The three curves represent the total processing time of all 10 queries on 1, 2, and 5 processors. The abscissa shows the different databases sizes, ranging from $10,000$ up to $140,000$. The ordinate shows the average total processing time.

We can see at the curve for a single node that the application scaled linearly with the number of database records up to $110,000$ records. From then on, the runtime grew tremendously and became increasingly variable between the different experiments because a single node was unable to keep the database in memory.

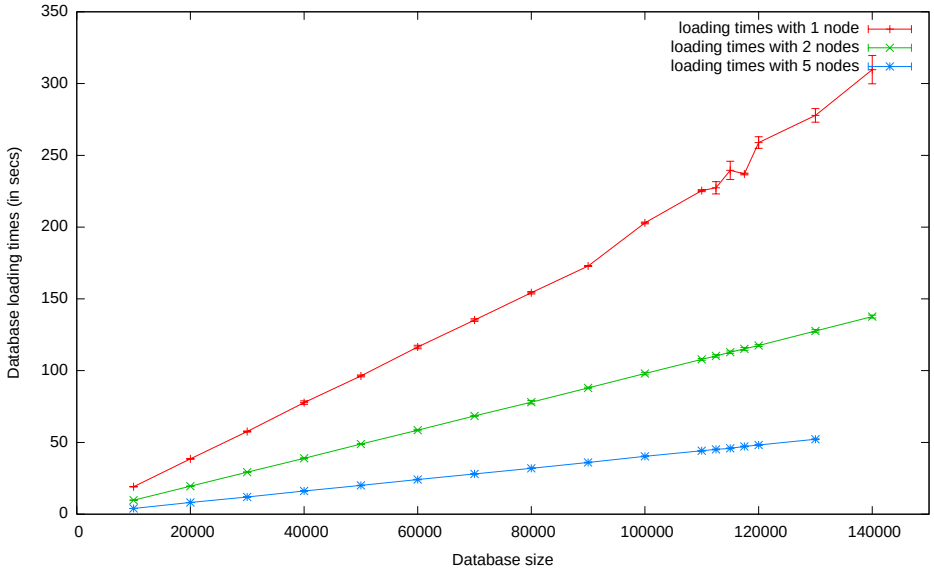


Figure 5.2.2: Database loading time

Using more nodes decreased the database fraction and therefore the memory consumption of each node. We see that the spike did not appear at 110,000 records; it got shifted to the right.

5.2.2 Database loading time

As databases are huge, we believe that *loading the database into memory contributes a considerable amount of time to the total processing time.* Therefore, we improve the runtime of the framework considerably by running it as a persistent service. In order to check this hypothesis, we have analyzed the database loading time of the previous benchmark.

Figure 5.2.2 shows the average time load the database fragments on each node from the file server into memory. As before, we have analyzed different database sizes. The curves show for each data point the average among all workers and all iterations. Table 5.2.1 lists the average loading times L, standard deviation in loading times σ and the speedup S for different numbers of nodes.

For 100,000 records in the database, the ratio of database loading time to processing a query on one node was 12.08 : 1, which strongly supports our idea of keeping

DB Size	L_1	σ_1	L_2	σ_2	S_2	L_5	σ_5	S_5
10000	19.1	0.14	9.7	0.08	1.97	4.0	0.03	4.79
20000	38.5	0.30	19.5	0.10	1.97	8.1	0.08	4.74
30000	57.6	0.38	29.3	0.22	1.97	12.0	0.08	4.81
40000	77.8	1.20	38.9	0.35	2.00	16.2	0.18	4.81
50000	96.3	0.61	48.9	0.30	1.97	20.0	0.17	4.81
60000	116.4	1.20	58.5	0.49	1.99	24.2	0.21	4.82
70000	135.2	0.89	68.4	0.41	1.98	28.1	0.22	4.82
80000	154.3	0.81	78.1	1.20	1.98	32.0	0.16	4.82
90000	172.9	0.12	88.0	0.56	1.96	36.0	0.24	4.80
100000	202.9	0.61	98.0	0.87	2.07	40.3	0.30	5.03
110000	225.4	0.69	107.8	0.66	2.09	44.2	0.22	5.10
112500	227.4	4.26	110.3	0.86	2.06	45.1	0.24	5.04
115000	239.6	6.37	112.9	0.76	2.12	45.9	0.26	5.22
117500	237.0	0.56	115.1	0.97	2.06	47.2	0.27	5.02
120000	258.9	4.04	117.5	0.65	2.20	48.2	0.40	5.37
130000	277.8	4.73	127.6	0.89	2.18	52.2	0.28	5.32
140000	309.7	9.84	137.6	1.01	2.25			

Table 5.2.1: Database loading time

the database in memory. Depending on the actual applications and their computing complexity the value can be lower or higher than in our experiment.

As the speedup of loading the database on 5 processors was close to 5, the file server was not the bottle neck of the benchmark. Our implementation uses STL streams and we cannot rule out that other approaches are more efficient.

5.2.3 Speedup

Next, we have to analyze the speed gains of the framework. As the framework needs to send few messages and search huge databases, we expect *close to 100% efficiency* for many nodes if the database fits into the main memory of one node. If the database exceeds the main memory of a single node, we expect *superlinear speedups* because we mitigate the need of swapping. These superlinear speedups cease to exist, however, if the aggregated RAM of all nodes is still too little for the whole database. We disregard concurrent query processing for now.

We begin with analyzing the speedup and efficiency if the whole database fits into the main memory of a single node. For that, we have benchmarked the execution of

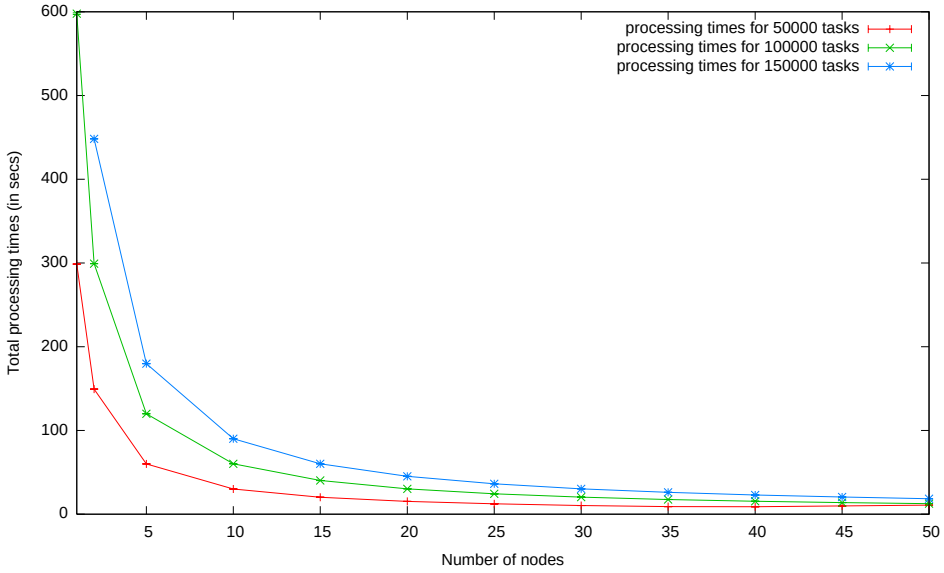


Figure 5.2.3: Processing time

20 queries on different numbers of nodes. Each experiment was executed three times on a database of $50,000$, $100,000$, and $150,000$ records.

Figure 5.2.3 shows the execution time of 20 queries with different numbers of nodes. Each data point represents the average of three experiments. We have used vertical bars to show the standard deviation of the values, but in this homogeneous environment there was hardly any variance as we can see in Table 5.2.2.

The columns in the tables are:

CPUs – Number of CPUs used

T_s – Processing time for all 20 queries on a database of size s (averaged over three experiments)

σ_s – Standard deviation in the processing time of the experiments.

S_s – Speedup

E_s – Efficiency

The speedup and efficiency are shown in Figure 5.2.4 and 5.2.5. We can see that the efficiency achieved good values of approximately 95% and 97% even with 50 nodes

CPUs	T_{50k}	σ_{50k}	S_{50k}	E_{50k}	T_{100k}	σ_{100k}	S_{100k}	E_{100k}	T_{150k}	σ_{150k}	S_{150k}	E_{150k}
1	298.7	0.01	n/a	n/a	597.8	0.95	n/a	n/a				
2	149.4	0.04	2.00	0.999	299.2	0.02	2.00	0.999	448.3	0.03	2.00	1.000
5	60.1	0.03	4.97	0.994	119.8	0.01	4.99	0.998	179.7	0.01	4.99	0.998
10	30.1	0.02	9.92	0.992	60.2	0.01	9.93	0.993	90.1	0.02	9.95	0.995
15	20.2	0.01	14.78	0.985	40.2	0.03	14.87	0.991	60.2	0.02	14.90	0.993
20	15.3	0.01	19.54	0.977	30.3	0.03	19.71	0.986	45.2	0.01	19.84	0.992
25	12.4	0.02	24.17	0.967	24.3	0.02	24.60	0.984	36.3	0.03	24.70	0.988
30	10.4	0.01	28.81	0.960	20.4	0.01	29.28	0.976	30.3	0.01	29.56	0.985
35	9.1	0.04	32.89	0.940	17.6	0.02	34.01	0.972	26.1	0.01	34.34	0.981
40	8.9	0.02	33.47	0.837	15.5	0.00	38.58	0.964	23.0	0.02	38.98	0.974
45	9.9	0.03	30.18	0.671	13.9	0.01	43.07	0.957	20.5	0.02	43.66	0.970
50	10.9	0.03	27.49	0.550	12.6	0.01	47.42	0.948	18.5	0.01	48.46	0.969

Table 5.2.2: Benchmark details

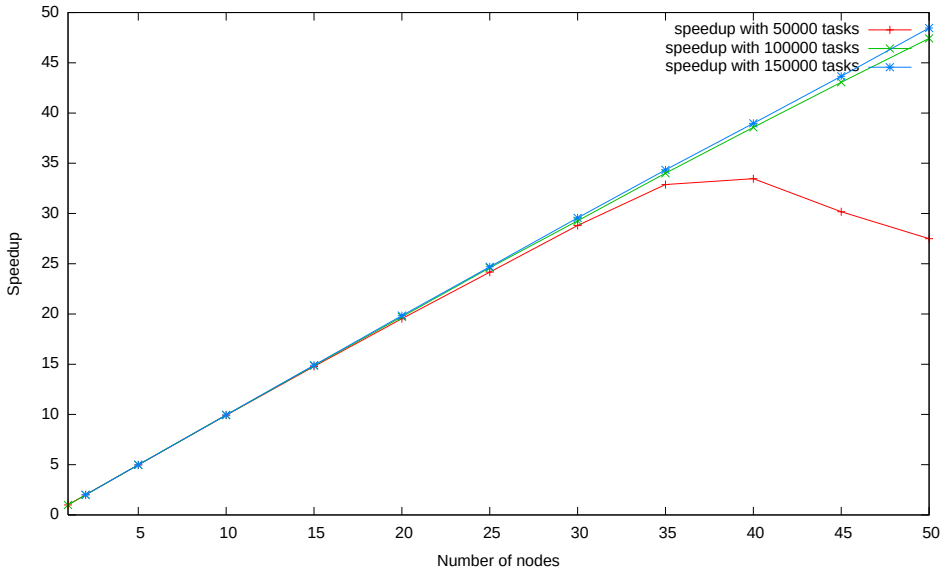


Figure 5.2.4: Speedup

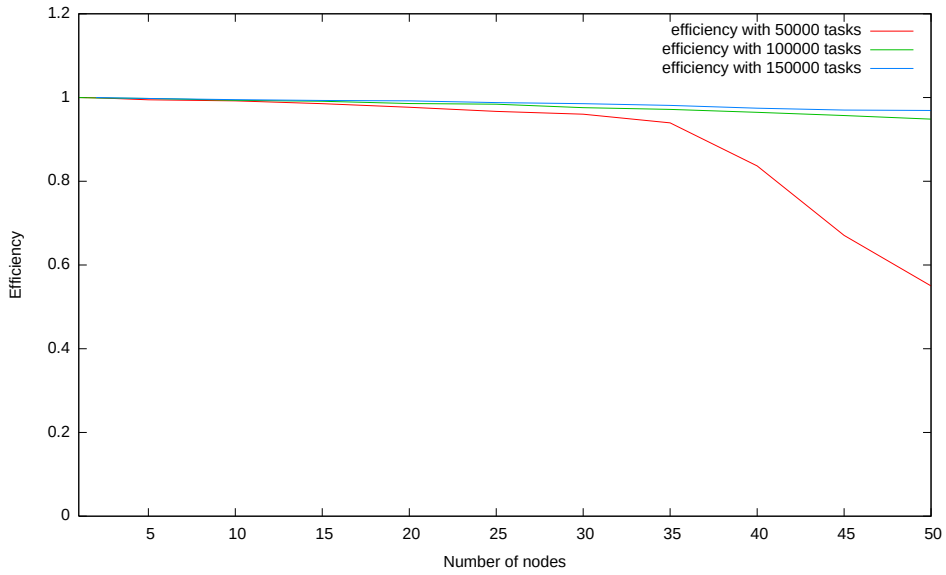


Figure 5.2.5: Efficiency

for databases of 100,000 and 150,000 records. However, the efficiency of queries on databases of only 50,000 records fell down to 55% with 50 nodes, which made them slower than queries processed with 40 nodes. In this case, the network became a bottleneck. We can see that 40 nodes brought the search time down to approximately 445 milliseconds for each query. Thus, there is hardly any need to increase the number of nodes in order to decrease the processing time. In order to increase the throughput and process many queries it is better to replicate the topology instead.

When we calculated the efficiency for 150,000 database records, we have not used the value measured for one processor. The reason is that we intended to disregard thrashing issues. As we can see, databases of 50,000 and 100,000 records achieved a speedup of approximately 2.0 for two processors. Therefore, the estimation

$$P_{150k}(1 \text{ CPU}) = 2 \cdot P_{150k}(2 \text{ CPUs}) \tag{5.2.1}$$

was appropriate, if we assume that one node had sufficient RAM to keep the whole database in memory. In our analysis, however, a node was not able to keep the whole database in memory, so we had to use this simplification.

Next, we use the actually measured processing time of 20 queries on a database of 150,000 records with a single node. Instead of the previously assumed 896.6 seconds for

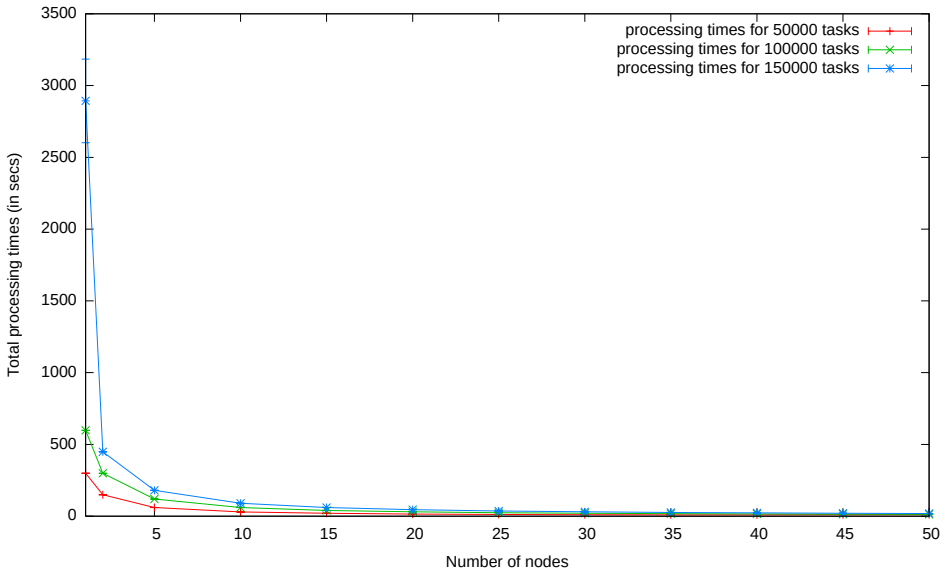


Figure 5.2.6: Processing time

processing 20 queries, we measure 2893.2 seconds with a significant standard deviation due to the thrashing (see Table 5.2.3). The thrashing slowed down the application by a factor of approximately 3.2. As we have seen in Figure 5.2.1, increasing the database size increases the thrashing effect even more.

If we consider the speedup from this point of view, we get completely different values that confirm our thesis of superlinear speedups as we can see in Figure 5.2.6, 5.2.7, and 5.2.8.

CPUs	T_{50k}	σ_{5k}	S_{50k}	E_{50k}	T_{100k}	σ_{10k}	S_{100k}	E_{100k}	T_{150k}	σ_{15k}	S_{150k}	E_{150k}
1	298.7	0.01	n/a	n/a	597.8	0.95	n/a	n/a	2893.2	291.10	n/a	n/a
2	149.4	0.04	2.00	0.999	299.2	0.02	2.00	0.999	448.3	0.03	6.45	3.227
5	60.1	0.03	4.97	0.994	119.8	0.01	4.99	0.998	179.7	0.01	16.10	3.219
10	30.1	0.02	9.92	0.992	60.2	0.01	9.93	0.993	90.1	0.02	32.10	3.210
15	20.2	0.01	14.78	0.985	40.2	0.03	14.87	0.991	60.2	0.02	48.06	3.204
20	15.3	0.01	19.54	0.977	30.3	0.03	19.71	0.986	45.2	0.01	64.02	3.201
25	12.4	0.02	24.17	0.967	24.3	0.02	24.60	0.984	36.3	0.03	79.69	3.188
30	10.4	0.01	28.81	0.960	20.4	0.01	29.28	0.976	30.3	0.01	95.39	3.180
35	9.1	0.04	32.89	0.940	17.6	0.02	34.01	0.972	26.1	0.01	110.82	3.166
40	8.9	0.02	33.47	0.837	15.5	0.00	38.58	0.964	23.0	0.02	125.77	3.144
45	9.9	0.03	30.18	0.671	13.9	0.01	43.07	0.957	20.5	0.02	140.87	3.130
50	10.9	0.03	27.49	0.550	12.6	0.01	47.42	0.948	18.5	0.01	156.38	3.128

Table 5.2.3: Benchmark details II

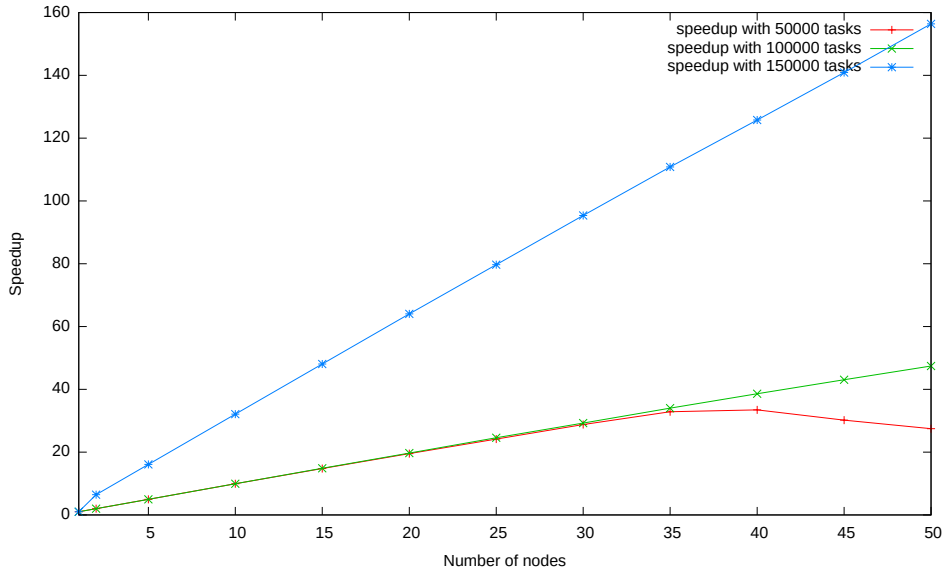


Figure 5.2.7: Speedup II

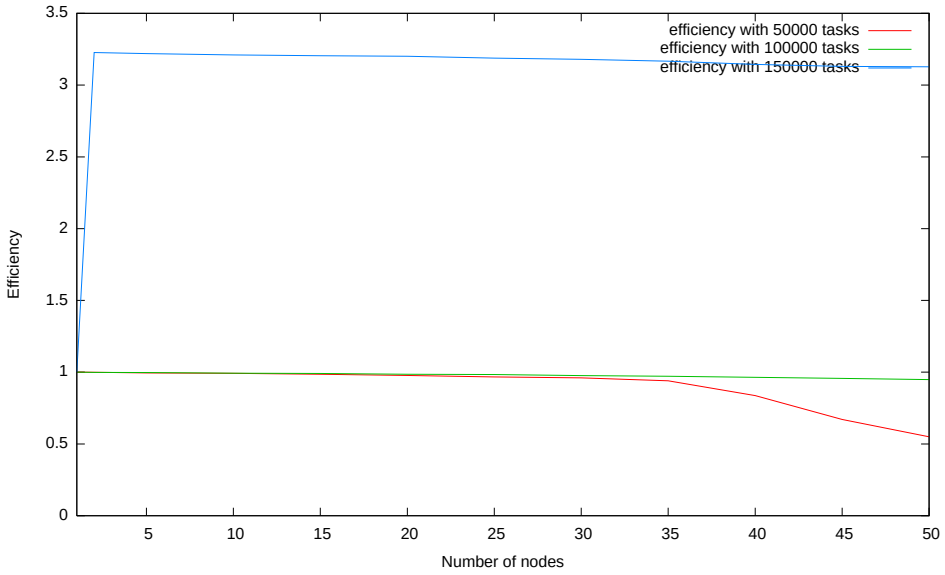


Figure 5.2.8: Efficiency II

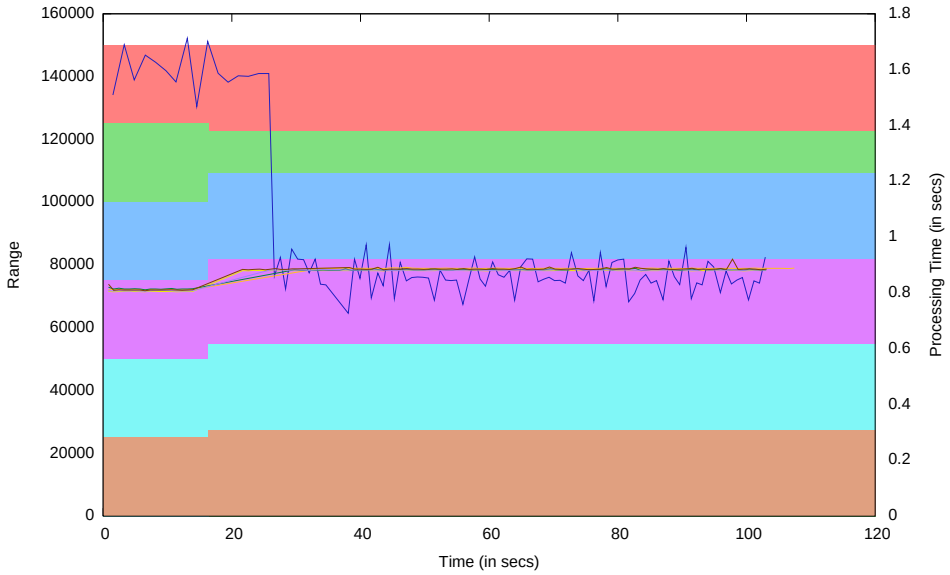


Figure 5.3.1: Refinement of responsibilities in case of a slow node

5.3 Refinement of Responsibilities

5.3.1 Slow node

The first example of how the framework refines responsibilities addresses the problem of a single node being slower than the other nodes. Figure 5.3.1 shows two overlaid graphs.

The colored bars are associated with the left axis. They show the ranges of the database assigned to the workers. Each bar represents by its upper and lower border the upper and lower end of the database fragment the worker is responsible for. The curves belong to the right axis and show the time a worker spent to complete a subquery.

In the benchmark shown in Figure 5.3.1, we have created a permanent load on one node by running additional processes. As a result, we see that the time to process one subquery was doubled on one node at the very beginning. After 10 queries, the framework noticed the imbalance in processing times, and reduced the range of responsibility of this slow node (at approximately 17 seconds). As the query queues of the workers were still filled, it took some time until the workers were able to execute the range updates. After that, all workers needed approximately the same time to process subqueries, even though one node was still suffering from the higher load. As

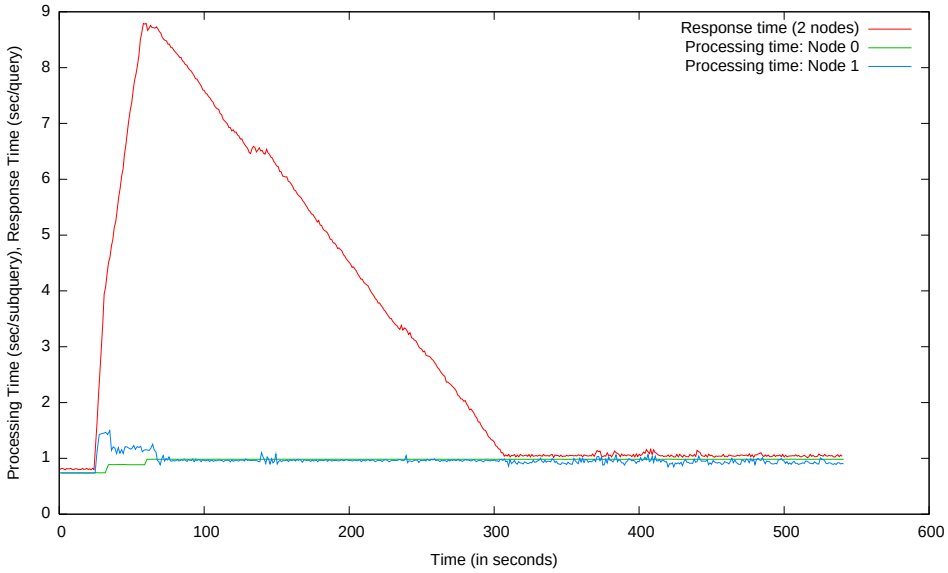


Figure 5.3.2: Response times in case of a slow node

the slowest node determines the throughput of the application, the range refinement is crucial for good runtimes.

Response times in case of a slow node

So far we have analyzed only the processing times of subqueries on the workers in case one node was slow and all queries were sent at once.

Figure 5.3.2 shows an example, where we submitted one query per second (in triggered benchmark mode) to the framework which was running on two nodes, one of which slowed down. The figure shows that the imbalance of processing times was fixed quickly, but that the total processing time of queries, which was determined by the slower node, rose beyond one second. This created a backlog of queries that needed to be worked up later.

5.3.2 Thermal problems

During the evaluation phase of the thesis, the cluster suffered from thermal problems due to the summer heat which the air conditioning system was not able to compensate for. The nodes of the PSC cluster are equipped with thermal sensors which throttle the CPU speed to 50% if they get too hot.

At the time when Figure 5.3.3 was recorded, the system was running at the border

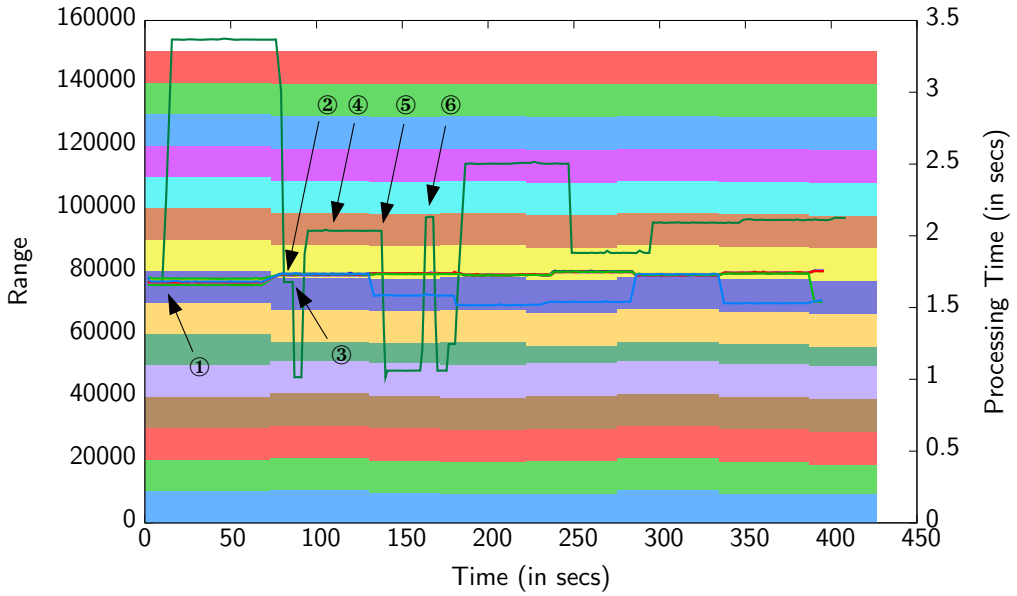


Figure 5.3.3: Refinement of responsibilities in case of thermal problems

of the temperature threshold. At the very beginning all nodes were running at the same speed, but after a few seconds at ①, one node passed the temperature threshold and became throttled to 50%. This resulted in twice as long processing times. A query took 1.68 seconds to process at full speed and 3.37 seconds while throttled.

Shortly before ②, the framework conducted a load balancing and assigned a smaller range to the slow node. As its query queue was still filled, this had no immediate consequence. At ②, the processor temperature fell below the threshold and operated at full speed.

Executing the range update at ③ made the processing time fall much below that of other nodes, because the system recorded query times of the slowed down node for a long time, while it was running at full speed at ③.

Shortly after executing the update, at ④, however, the node passed the threshold again and was again slower than the other nodes. We did not anticipate the node to achieve exactly the same execution speed because the average included some time of executing at full speed as well.

At ⑤ and ⑥, we see a surprising effect. First, the node fell below the temperature threshold at exactly the same time, when it executed the range update. Shortly after that, at ⑥, however it fell below the temperature threshold again, and ran at full speed.

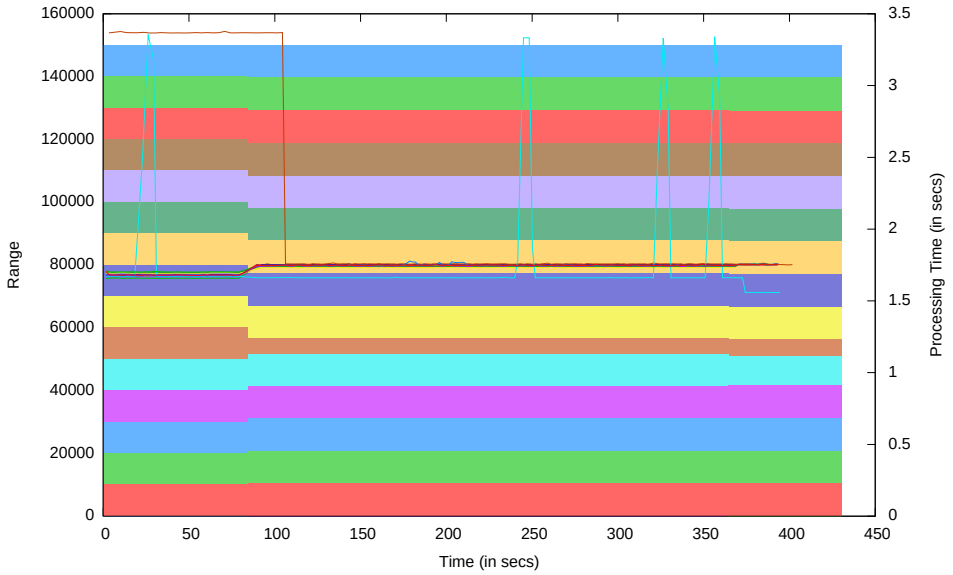


Figure 5.3.4: Refinement of responsibilities in case of thermal problems II

Here we see that the framework assigned the node an even bigger fraction than at ④, even though it was slower than the other nodes for most of the time.

The reason for this reveals if we check the parameters of the execution. The framework sent query updates every 25 queries and was set to a learning factor of 0.05. This means that the first value measured after an update made up for $(1 - 0.05)^{25} = 27.7\%$ of the floating average that was used for calculating the new database ranges. As the node was significantly faster than the average for a short time, and this accounted for almost a third of the average value, the framework assumed the node to be node faster than it was in fact.

We see that the learning factor of 0.05 was too low for an update frequency of once every 25 queries. The default values request 50 queries at a learning factor of 0.05. This reduces the importance of the first measured value to 7.7% – approximately the same importance as the last measured value (5%).

Figure 5.3.4 shows a second analysis with several nodes suffering from thermal problems. This time, the learning factor was increased to 10%. As we can see, the framework was capable of bringing the processing times into pretty close proximity. The framework adapted the database size of one consistently slow node. Only one node stayed faster than the other nodes. The reason for this is that its average processing

time deviated by less than 5% from the average of all average processing times. The framework considers this as tolerable.

Figure 5.3.5 shows the profile of this execution. During the first period of uneven load balance, we see that one slow node forced the other nodes to idle. This is an effect of the query queue length control. The master sent no additional subqueries until less than 5 results were pending. Most nodes processed their queries faster and had to wait until the master sent more queries. After rebalancing, all nodes were utilized nicely.

5.3.3 Inhomogeneous database partitions

The next problem we want to inspect is that of inhomogeneous databases. Previously we used a randomly generated database, but in reality we do not expect databases to have records of uniformly distributed lengths. Instead, they contain records of different species or cells with different length distributions.

In order to analyze this problem we have generated a database with records increasingly sorted by their size. The database contained $150,000$ records; the first having a size of 100 peaks, the last having a size of 1000 peaks.

If we search through the database with 5 nodes and disable load balancing, the last node has to search $\frac{1}{5} \cdot 150,000$ records which are on average more than 5 times bigger than the ones of the first node.

Figure 5.3.6 shows the database partitions and processing times in the case of disabled refinement of partitions.

Figure 5.3.7 shows the effect of partition refinement. The first refinement showed an over reaction. The framework considered the first node being the fastest and added more records to its responsibility. However, these additional records were bigger than the ones used for benchmarking the node before. Therefore, the resulting database range needed more time to be searched through than anticipated. However, this got balanced after just two additional refinements.

The refinement reduced the time to process 200 queries from 1696.4 to 1146.9 seconds. This is a drop by 32.4%. The average time to process the last queries dropped from 8.48 seconds to 5.01 seconds, a gain of 40.9%. An alternative to range refinements for inhomogeneous databases is to permute the records randomly as it creates partitions of comparable sizes.

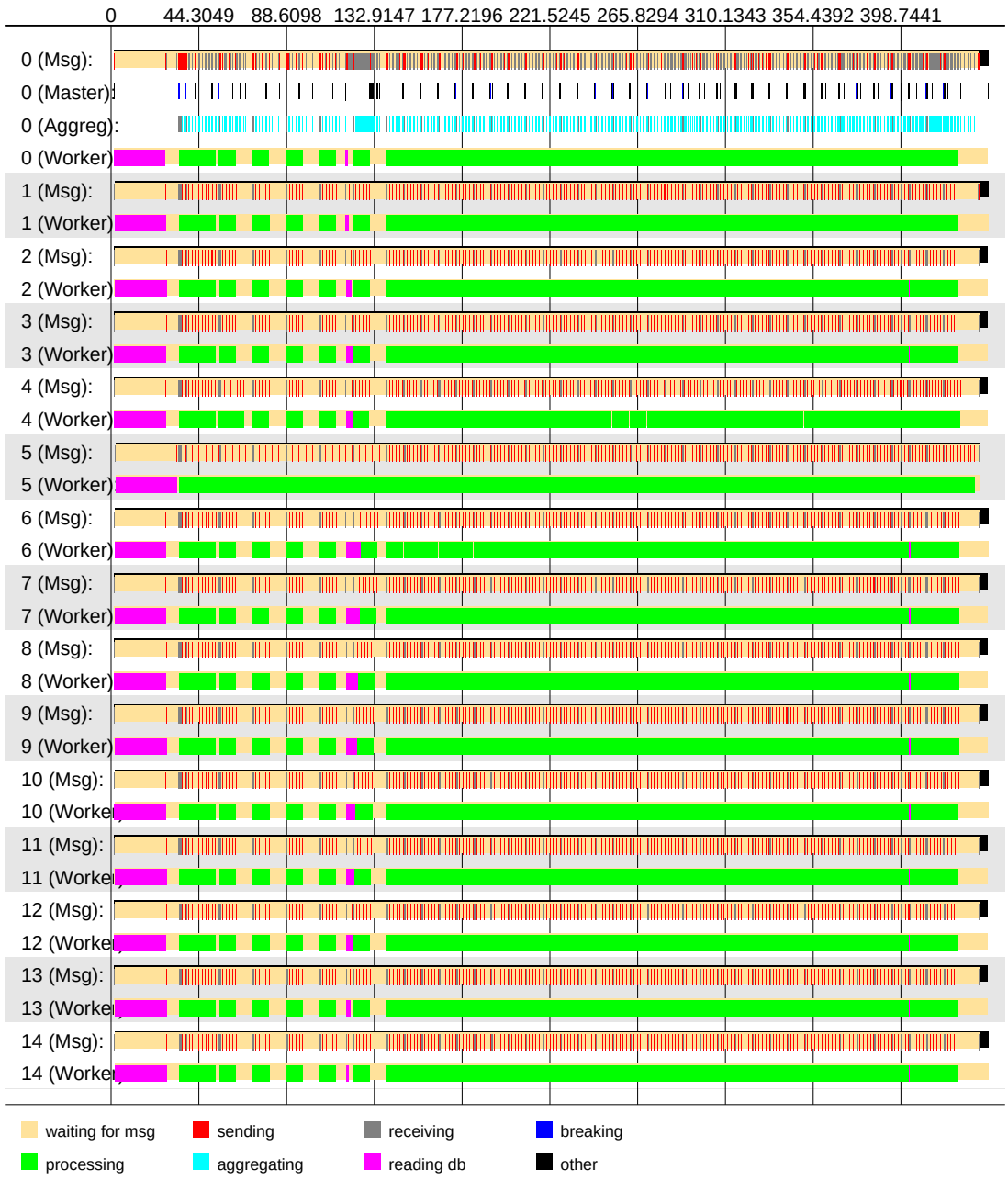


Figure 5.3.5: Refinement of responsibilities in case of a slow node

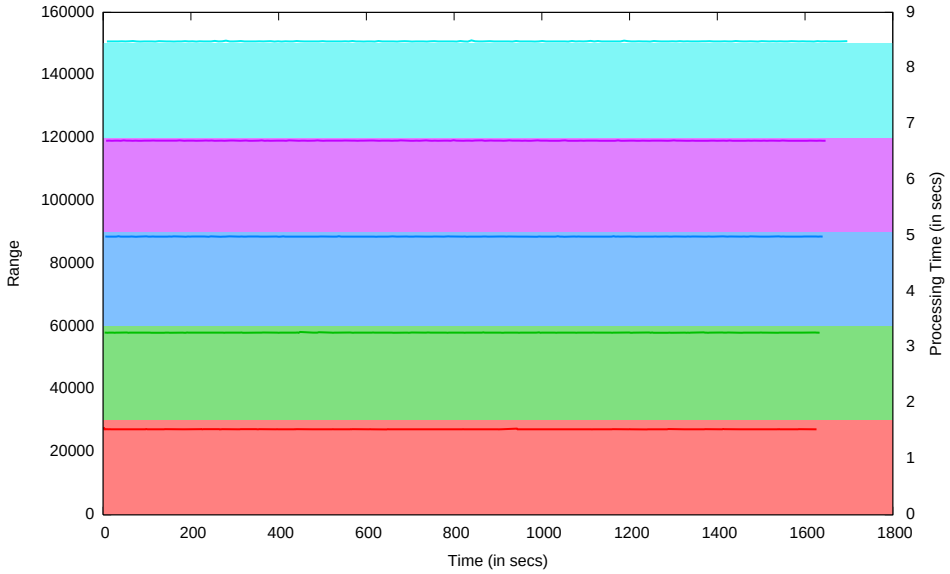


Figure 5.3.6: Processing times of inhomogeneous database without refinement

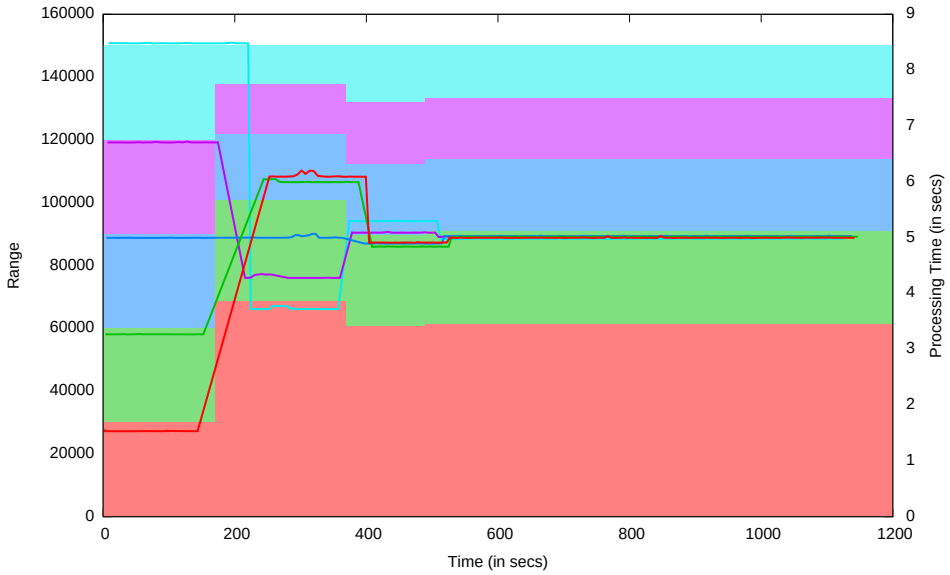


Figure 5.3.7: Processing times of inhomogeneous database with refinement

Experiment	P_1	σ_1
no reordering:	1582.2	236.1
with reordering:	804.0	81.4

Table 5.4.1: Total processing times with query reordering

5.4 Multiple Databases

Our final analysis considers the use of multiple databases. For that, we use two (identical) databases of 65,000 records each. The combined number of records exceeds the RAM of a single node as we have seen before. We expect that *query reordering improves the time to process the stream of queries* due to fewer context switches. As one database fits into the RAM of the node, but not both, we need to swap pages whenever we switch the database.

We have sent 5 times 20 queries, alternating to database 1 and 2. In the first experiment the processing node was not allowed to reorder queries, in the second experiment it was allowed to prefetch and process 4 queries at a time. This resulted in the following processing order of queries:

```
Step 1: 1
Step 2:  2   4   6   8
Step 3:    3   5   7   9
Step 4:                  10    12    14    16
Step 5:                     11    13    15    17
Step 6:                                          18    20
Stel 7:                                             19
```

Odd queries regarded database 1, even queries regarded database 2. The queries were processed row by row, i.e. in the first step only one query on database 1 was processed. In the second step, 4 queries (2, 4, 6, 8) on database 2 were processed. In the third step, 4 queries (3, 5, 7, 9) on database 1 were processed.

Table 5.4.1 shows the processing times for 20 queries averaged over 5 iterations as well as their standard deviation. We can see that allowing to process 4 queries concurrently cuts down the processing time by 49.2%.

In this example, we have reduced the number of context switches from 19 to 6. Allowing the application to prefetch more queries could improve the processing time

even further. However, creating long query queues can result in bad performance in case of uneven processor loads.

In order to check whether the processing time deteriorates over time because of paging and memory fragmentation, we have increased the number of queries to 80; all other parameters remained the same.

Figure 5.4.1 shows the processing times of separate queries. The values depicted are not the ones recorded by the framework. As queries are processed concurrently, it is not possible to assign processing times to individual queries. For each batch, we have assigned the ratio of its total processing time to the number of its queries as the queries' processing times. Their finishing times are artificially distributed evenly in the processing time of the batch.

We can see from Figure 5.4.1 that there is no tendency of a deteriorating processing time after several context switches. The high processing times during the last two batches are due to the fact that the expensive context switch could not be amortized among 4 queries.

Figure 5.4.2 shows the processing without query reordering. We see that the average query times are much higher compared to Figure 5.4.1.

We have introduced two inversions into the series of alternating database queries

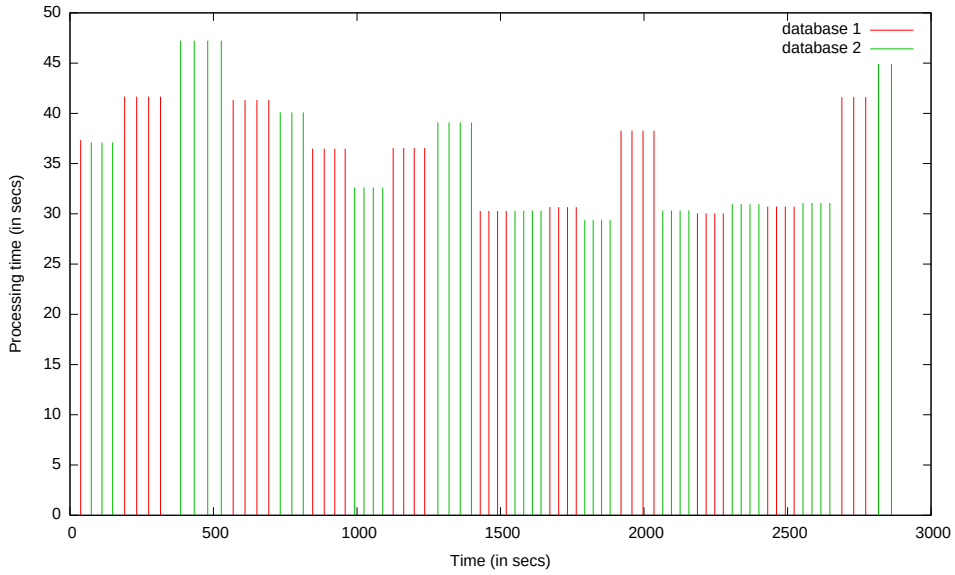


Figure 5.4.1: Processing times of separate queries with query reordering

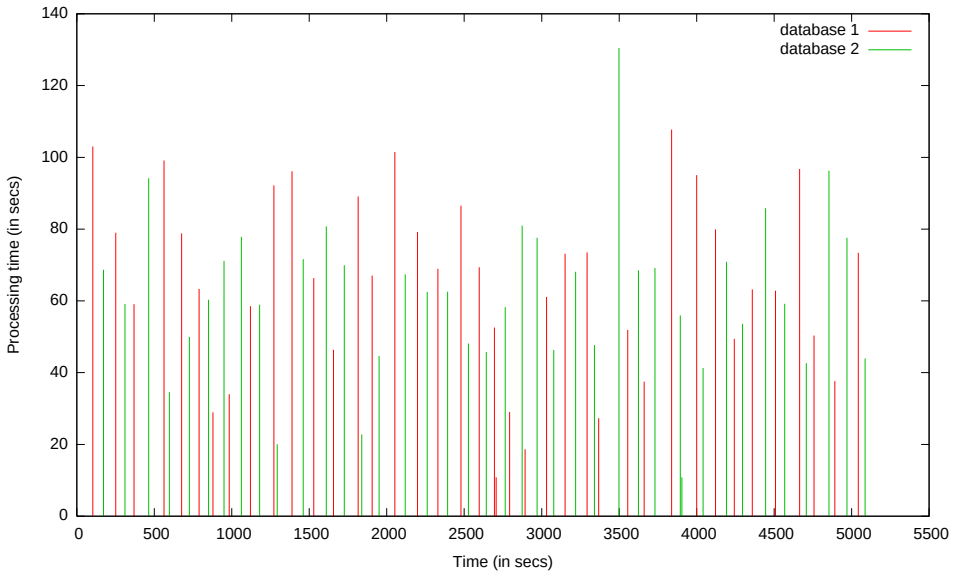


Figure 5.4.2: Processing times of separate queries without query reordering

where two consecutive queries were sent to the same databases. These were processed at approximately 2700 seconds and 3900 seconds in Figure 5.4.2. We see that both times the second query of these tuples was processed much quicker than the first one. This supports the strategy of grouping queries.

From what we have seen, we can conclude that prefetching is a good approach for the support of several databases.

5.5 Conclusion

The benchmarks shown in the previous sections confirm that the framework scales well up to many nodes and is capable of handling load imbalances on nodes in relatively stable environments. In case of very volatile environments (as in the case of temperature problems), the framework performs less well and simple master/worker frameworks without support for data–locality might perform better. An experimental analysis of this is necessary however. Finally, we were able to show the benefits of query reordering.

6 Conclusion

We have shown that many biological databases need to iterate over all records in order to find matching database entries to a query. This process is time consuming due to the size of the databases and gets even slower if the database does not fit into the memory of the searching computer.

We have introduced a framework to distribute the work of searching through biological databases between several nodes that work in tandem. This framework consists of a generic part and database domain specific plugins. It provides the infrastructure to search through databases efficiently while attempting to keep as many parts of the database in memory as possible.

We have applied the framework to the problem of finding similar mass spectra with the Spectral Contrast Angle approach and demonstrated good scalability up to many nodes and superlinear speedups in case of big databases.

Owing to the easy integration of the framework with SOAP and the good performance results, the framework is well suited for many kinds of biological databases. The framework requires little effort to enable database searches for clusters and with MPICH-G2 even for the grid.

The framework can and will be improved in several areas in the future. Recovery mechanisms can be introduced by storing queries and results in databases. Using the serialization features of query and result objects, it is easy to backup the respective data in a database. Load imbalances due to very uneven and strongly fluctuating processing times at the workers can reduce the performance of the framework. Replacing the master with a new implementation that assigns jobs to workers by sending a range refinement request immediately followed by a subquery allows for simulating the mpiBLAST approach, which might perform better in this case.

The very next step is to integrate the framework into the Illinois Bio–Grid Mass Spectrometry Database to provide searchable proteomic MS data.

We consider the framework ready for deployment. Owing to the benefits we have shown throughout this thesis, we recommend researchers in the field of bioinformatics to evaluate the utilization of the framework.

The source codes and additional documentation can be found at:

`http://www.dominicbattre.de/mastersthesis/`

Bibliography

[1] A Java based heterogeneous distributed computing system. URL `http://www.cs.may.ie/~tkeane/distributed/`.

[2] D. Abramson, A. Barak, and C. Enticott. Job management in grids of mosix clusters. In *Parallel and Distributed Computing and Systems*, 2003.

[3] D. Abramson, J. Giddy, and L. Kotler. High performance parametric modeling with nimrod/G: Killer application for the global grid? In *14th International Parallel and Distributed Processing Symposium*, pages 520–528, May 2000.

[4] B. Allcock, J. Bester, J. Bresnahan, and et al. Effcient data transport and replica management for high-performance data-intensice computing. In *18th IEEE Symposium on Mass Storage Systems and 9th NASA Goddard Conference on Mass Storage Systems and Technologies*, April 2001.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 215:403–410, 1990.

[6] S. F. Altschul, T. L. Madden, A. A. Schaffer, J. Zhang, Z. Zhang, W. Miller, and D. J. Lipman. Gapped BLAST and PSI-BLAST: a new generation of protein database search programs. *Nucleic Acids Research*, 25:3389–3402, 1997.

[7] David P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *5th IEEE/ACM International Workshop on Grid Computing*, November 2004.

[8] Micha Bayer, Aileen Campbell, and Davy Virdee. A gt3 based blast grid service for biomedical research. In *Proceedings of the UK e-Science All Hands Meeting 2004*, 2004.

[9] Kevin Bealer. Personal communication, May 2005.

[10] Berkeley Open Infrastructure for Network Computing. URL `http://boinc.berkeley.edu/`.

[11] Francine Berman, Gary Shao, and Jim Hayes. The AppLeS Master/Worker Application Template. URL `http://apples.ucsd.edu/amwat/`.

[12] Francine D. Berman, Rich Wolski, Silvia Figueira, Jennifer Schopf, and Gary Shao. Application-level scheduling on distributed heterogeneous networks. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 39, 1996.

[13] Boost. URL `http://boost.org/`.

[14] J. M. Bull, L. A. Smith, L. Pottage, and R. Freeman. Benchmarking java against c and fortran for scientific applications. In *Java Grande*, pages 97–105. ACM Press, 2001.

[15] Cactus Grid Application Toolkit. URL `http://www.gridlab.org/WorkPackages/wp-2/`.

[16] Michael Cameron, Hugh E. Williams, and Adam Cannane. Improved gapped alignment in blast. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 1(3):116–129, 2004.

[17] Arjav J. Chakravarti, Gerald Baumgartner, and Mario Lauria. The Organic Grid: Self-Organizing Computation on a Peer-to-Peer Network. *IEEE Transactions on Systems, Man, and Cybernetics*, 35(2), March 2005.

[18] L. Paul Chew, Dan Huttenlocher, Klara Kedem, and Jon Kleinberg. Fast detection of common geometric substructure in proteins. In *Proceedings of the third annual international conference on Computational molecular biology*, pages 104–113, 1999.

[19] Cilk. URL `http://supertech.lcs.mit.edu/cilk/`.

[20] Condor high throughput computing. URL `http://www.cs.wisc.edu/condor/`.

[21] Condor MW. URL `http://www.cs.wisc.edu/condor/mw/`.

[22] Aaron E. Darling, Lucas Carey, and Wu chun Feng. The Design, Implementation, and Evaluation of mpiBLAST. In *Cluster World Conference and Expo*, June 2003.

[23] EMBL-EBI Nucelotide Sequence Database. URL `http://www.ebi.ac.uk/embl/`.

[24] EMBL Statistics. URL `http://www3.ebi.ac.uk/Services/DBStats/`.

[25] Fasta format. URL `http://ngfnblast.gbf.de/docs/fasta.html`.

[26] Yan Fu, Qiang Yang, Ruixiang Sun, Dequan Li, Rong Zeng, Charles X. Ling, and Wen Gao. Exploiting the kernel trick to correlate fragment ions for peptide identification via tandem mass spectrometry. *Bioinformatics*, 20(12):1948–1954, 2004.

[27] Globus toolkit. URL `http://www.globus.org`.

[28] Grid Application Framework for Java (GAF4J). URL `http://alphaworks.ibm.com/tech/gaf4j`.

[29] William Gropp and Ewing Lusk. PVM and MPI are completely different. *Future Generation Computer Systems*, 1999.

[30] William Gropp, Ewing Lusk, and Rajeev Thakur. *Using MPI-2: Advanced Features of the Message-Passing Interface*. MIT Press, 1999.

[31] gSOAP. URL `http://gsoap2.sourceforge.net/`.

[32] ht://dig. URL `http://www.htdig.org/`.

[33] Illinois BioGrid Mass Spectrometry Database. URL `http://illinoisbiogrid.org/MSDB`.

[34] Thomas Keane. Java distributed system: Developer manual. Technical Report NUIM-CS-TR-2003-03, National University of Ireland, Maynooth, 2003.

[35] Thomas Keane. A general-purpose heterogeneous distributed computing system. Master's thesis, National University of Ireland, Maynooth, July 2004.

[36] Thomas Keane. Personal communication, May 2005.

[37] Michael Kinter and Nicholas Sherman. *Protein Sequencing and Identification Using Tandem Mass Spectrometry*. Wiley-Interscience, 2000.

[38] Dan. E. Krane and Michael L. Ramyer. *Fundamental Concepts of Bioinformatics*. Pearson Education, 2003.

[39] Sanjeev Kulkarni. An intelligent framework for master-worker applications in a dynamic metacomputing environment, May 2001.

[40] Melissa Lemos and Lifschitz Lifschitz. A Study of a Multi-Ring Buffer Management for BLAST. In *14th International Workshop on Database Systems Applications (DEXA'03)*, 2003.

[41] Heshan Lin, Xiaosong Ma, Praveen Chandramohan, Al Geist, and Nagiza Samatova. Efficient Data Access for Parallel BLAST. In *International Parallel and Distributed Processing Symposium*, 2005.

[42] D. J. Lipman and W. R. Pearson. Rapid and sensitive protein similarity search. *Science*, 227:1435–1441, 1985.

[43] Supratik Majumder. High performance mpi libraries for ethernet. Master's thesis, Rice University, September 2004.

[44] Mass spectrum of tetrachlordibenzofuran. URL `http://de.wikipedia.org/wiki/Bild:Msintro04.gif`, GNU-FDL licence.

[45] J. May. *Parallel I/O for High Performance Computing*. Morgan Kaufmann Publishers, 2001.

[46] mpiBLAST. URL `http://mpiblast.lanl.gov/`.

[47] MrBayes: Bayesian Inference of Phylogeny. URL `http://morphbank.ebc.uu.se/mrbayes/`.

[48] Ncbi blast web site. URL `http://www.ncbi.nlm.nih.gov/BLAST/`.

[49] NCBI Tools. URL `ftp://ftp.ncbi.nlm.nih.gov/toolbox/ncbi_tools/`.

[50] S. B. Needleman and C. D. Wunsch. A general method applicable to the search for similarities in the amino acid sequences of two proteins. *Journal of Molecular Biology*, 48:443–453, 1970.

[51] Network weather service. URL `http://nws.cs.ucsb.edu/`.

[52] Nimrod: Tools for distributed parametric modelling. URL `http://www.csse.monash.edu.au/~davida/nimrod/`.

[53] Andrew Page, Thomas Keane, and Thomas J. Naughton. Adaptive scheduling across a distributed computation platform. In *3rd International Symposium on Parallel and Distributed Computing*, 2004.

[54] W. R. Pearson. Rapid and sensitive sequence comparison with FASTP and FASTA. *Methods in Enzymology*, 183:63–98, 1990.

[55] PeptideAtlas. URL `http://www.peptideatlas.org/`.

[56] Pavel A. Pevzner, Vlado Dancik, and Chris L. Tang. Mutation-tolerant protein identification by mass-spectrometry. In *RECOMB '00: Proceedings of the fourth annual international conference on Computational molecular biology*, pages 231–236, New York, NY, USA, 2000. ACM Press.

[57] Pavel A. Pevzner, Zufar Mulyukov, Vlado Dancik, and Chris L. Tang. Efficiency of database search for identification of mutated and modified proteins via mass spectrometry. *Genome Research*, 11(2):290–299, Feb 2001.

[58] PVM – Parallel Virtual Machine. URL `http://www.csm.ornl.gov/pvm/`.

[59] S.T. Rao and M.G. Rossman. Comparison of super-secondary structures in proteins. *Journal of Molecular Biology*, 76:211–256, 1973.

[60] RCSB Protein Data Bank. URL `http://www.rcsb.org/pdb/`.

[61] Gary Shao. *Adaptive Scheduling of Master/Worker Applications on Distributed Computational Resources*. PhD thesis, University of California, San Diego, 2001.

[62] Gary Shao, Fran Berman, and Rich Wolski. Master/slave computing on the grid. In *9th Heterogeneous Computing Workshop*, May 2000.

[63] F. F. Smith and M. S. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147:195–197, 1981.

[64] Neil T. Spring and Richard Wolski. Application level scheduling of gene sequence comparison on metacomputers. In *International Conference on Supercomputing*, pages 141–148, 1998.

[65] Craig A. Stewart, David Hart, Donald K. Berry, Gary J. Olsen, Eric A. Wernert, and William Fisher. Parallel implementation and performance of fastDNAml - a

program for maximum likelihood phylogenetic inference. *Proceedings of SC2001*, November 2001.

[66] Tom Strachan and Andrew P. Read. *Human Molecular Genetics 2*. Wiley-Liss, 1999.

[67] Todd Tannenbaum, Derek Wright, Karen Miller, and Miron Livny. Condor – a distributed job scheduler. In Thomas Sterling, editor, *Beowulf Cluster Computing with Linux*. MIT Press, October 2001.

[68] The Message Passing Interface (MPI) standard. URL `http://www-unix.mcs.anl.gov/mpi/`.

[69] Katty X. Wan, Ilan Vidavsky, and Michael L. Gross. Comparing similar spectra: from similarity index to spectral contrast angle. *Journal of the American Society of Mass Spectrometry*, 13:85–88, 2002.

[70] Jiren Wang and Qing Mu. Soap-HT-BLAST: high throughput BLAST based on Web services. *Bioinformatics Applications Note*, 19(14):1863–1864, 2003.

[71] Richard Wolski. Dynamically forecasting network performance using the network weather service. *Cluster Computing*, 1(1):119–132, 1998.

[72] Yifeng Zhu, Hong Jiang, Xiao Qin, and David Swanson. A case study of parallel i/o for biological sequence search on linux clusters. In *IEEE International Conference on Cluster Computing (CLUSTER'03)*, December 2003.