First publ. in: Mediterranean Journal of Computers and Networks 3 (2007), 2, pp. 43-51

PARALLEL AND DISTRIBUTED DATA PIPELINING WITH KNIME

C. Sieb *, T. Meinl, M. R. Berthold
ALTANA Chair for Bioinformatics and Information Mining, Department of Computer and Information Science, University of Konstanz, Germany

ABSTRACT

In recent years a new category of data analysis applications have evolved, known as data pipelining tools, which enable even non-experts to perform complex analysis tasks on potentially huge amounts of data. Due to the complex and computing intensive analysis processes and methods used, it is often neither sufficient nor possible to simply rely on the increase of performance of single processors. Promising solutions to this problem are parallel and distributed approaches that can accelerate the analysis process. In this paper we discuss the parallel and distribution potential of pipelining tools by demonstrating several parallel and distributed implementations in the open source pipelining platform KNIME. We verify the practical applicability in a number of real world experiments.

Keywords

Parallel, Distributed, Data Analysis, Pipelining, KNIME.

1. INTRODUCTION

In recent years the amount of data generated on a daily basis is growing at a mind boggling rate. The information and knowledge hidden in this data can only be discovered by sophisticated and high performance analysis methods [10, 9, 23]. It has become increasingly clear that visual exploration techniques and interactive methods can help analysts better understand the extracted information and guide the mining process by their domain knowledge [22, 20]. However, for this to be successful it is crucial to enable easy and intuitive access to the vast variety of analysis tools available nowadays. A recently emerging category of tools for such types of data analysis are the so-called data pipelining tools that enable analysts to dynamically create interactive data analysis workflows. Some of today's popular data pipelining tools are Pipeline Pilot [17], Insightful Miner[12], InforSense KDE [11], D2K [21], DataRush [19] and the recently released, open source data mining platform KNIME ("Konstanz Information Miner", [6, 2]).

*Corresponding author: *E-mail:* sieb@inf uni-konstanz de

All Rights Reserved No part of this work may be reproduced, stored in retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise - except for personal and internal use to the extent permitted by national copyright law - without the permission and/or a fee of the Publisher

In these tools, the pipeline is formed from consecutively connected processing units called nodes. The raw input data can be read from various data sources, such as text files and databases. Usually the data is transformed into table-like internal representations. These tables are then passed along the pipeline to other nodes, which handle pre-processing such as normalizing numerical values, filtering rows based on specific criteria or joining tables from different branches of the workflow. Subsequent nodes then apply machine learning or data mining algorithms to build models based on the input data. Popular methods include decision trees, rule sets or support vector machines for labeled data or clustering algorithms and pattern mining for unlabeled data [23]. Finally, nodes providing tools for interactive visualization help to explore the results.

These steps can of course also be applied by a handwritten script or program. However, by using a graphical representation of the pipeline and the nodes that process the data, the purpose of the workflow becomes much more obvious and the transfer of knowledge among a group of analysts is improved significantly. The flow is intuitive and self-documenting due to its visual representation and even users who do not know much about programming can analyze data quite easily and are able to quickly modify existing pipelines to their own liking. An example of a simple data analysis workflow using KNIME is shown in Fig. 1.

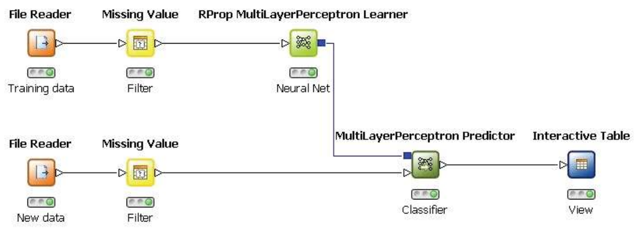


Figure 1. A simple workflow in KNIME

The File Reader node in the upper left area reads pre-classified training data from a file followed by a node, which subsequently filters out rows containing missing attribute values. The upper branch follows by creating a neural network model. Model flows are distinguished by the differently shaped ports. In the lower branch another file reader reads in a new, previously unclassified data set that is to be classified by the model created in the upper branch. After filtering the missing values, the predictor node takes the trained neural network model as well as the new data and classifies it. The following table view visualizes the predicted values.

Contrasting the appealing properties of visual workflow layouting, there are also some minor drawbacks to this concept. First, due to the fact that the workflows must be able

Konstanzer Online-Publikations-System (KOPS)
URL: http://www.ub.uni-konstanz.de/kops/volltexte/2008/6485/
URN: http://nbn-resolving.de/urn:nbn:de:bsz:352-opus-64858

to deal with all kinds of input data, the used data structures must be generic and usually end up offering a rather high abstraction level. Therefore, the used data structures are usually not optimized for specific tasks. Second, usability features like progress indication require additional system resources. This becomes a problem especially if huge amounts of data (hundreds or thousands of megabytes) are analyzed. Because the speed of single processor machines does not catch up with the growing amount of data, parallel and distributed approaches are promising techniques to speed up the data analysis process.

Data pipelining tools provide a lot of parallel potential to accelerate the workflow execution. Even though the general ideas in the field of parallel and distributed computing are not new as such, it is important to analyze the capabilities that arise from the nature of those pipelining tools. In this paper, we present an analysis of these capabilities and map them to potential parallelization and distribution techniques, respectively.

The rest of this paper is organized as follows. In Section 2, we provide a short overview of the field of parallel and distributed computing with respect to data pipelining and data mining. In Section 3, we introduce the used data analysis platform KNIME. The main part of this article in Section 4 describes various ways to parallelize and distribute processing data and tasks in KNIME, and in Section 5 details are provided regarding some experimental results. Finally, we briefly contrast this work to other tools' approaches.

2. PARALLEL AND DISTRIBUTED COMPUTING

In the last decades a lot of research and development work has been done in the area of parallel and distributed computing. The two principal concepts we focus on are Symmetric Multi Processing (SMP), in which global memory is shared among several CPUs or cores, and distributed computing, where independent processing units do not share anything but are connected by ordinary or high speed networks [13]. Depending on the specific manner and speed of the communication, different approaches are applied to parallelize computing tasks.

In general, SMP approaches are easier to implement as the communication and synchronization of parallel processes can be performed via global memory (shared memory, pipes, semaphores, monitors, and others), which avoids the necessity of explicitly transferring information to other computers. In special cases parallelization can even be performed by a compiler, which distributes independent runtime code (e.g. loop-cycles) to the available CPUs. Furthermore, communication between processors is much faster. One disadvantage is that the scalability potential is limited due to the central bottleneck represented by the global memory. For this reason, today's SMP architectures employ hierarchical memory/CPU structures, which however increase the complexity and communication time among processors.

Communication and synchronization in distributed systems is performed via messages. In these systems, messages must be sent and received explicitly in either a synchronous or asynchronous way. For standardization purposes a general interface has been defined, known as the Message Passing Interface (MPI) [15].

Major consequences arise from these two architectures and the application that should be parallelized. First, it is important to define what should be parallelized. Either the data to be processed can be distributed, the task itself, or both. One special issue in data mining applications is the distribution of the search space in cases where it is much bigger than the data itself [18, 14].

In data pipelining tools it is possible to exploit "Pipeline Parallelism". Tools that support stream processing of data [19] can execute a data item in one processing unit and immediately forward the result to the next unit. While the next unit processes the result, the previous unit can continue to process another data item [8].

The second aspect in parallel and distributed computing is the granularity of the data subsets or subtasks to be distributed. In the case of fast communication (i.e. SMP or fast connected clusters) it is possible to distribute small portions of data. This potentially enables optimal balancing of the load among the available processors. In distributed systems with slow networks the communication overhead would neutralize the benefit of such fine-grained parallel execution.

One last aspect is the already mentioned workload balancing among the participating computers or CPUs. If the complexity of the task and its parts can be determined in advance, static load balancing can be performed. However, in many data mining tasks the size of the problem and the structure of the search space are not known in advance. A famous example is frequent itemset mining or pattern mining in general [1, 3, 25]. In this case the work load must be distributed dynamically during application runtime. Therefore, a dynamic load balancing system is needed to detect which computers have a high work load and which are underloaded. Work packages are then subsequently reassigned from one computer to another [24].

Following this brief overview of parallel and distributed computing, the next section describes the KNIME data pipelining platform. We will see that several aspects mentioned in this section play an important role for later parallelization and distribution.

3. KNIME - A DATA PIPELINING PLATFORM

The Konstanz Information Miner (KNIME) is a Java-based data mining platform with a graphical user interface that is based on Eclipse [5]. A workflow in KNIME consists of several nodes belonging to various categories (readers, manipulators, learners, predictors, writers), which are connected via ports. A connection can either transfer data or generated models, which describe extracted information from the input data such as learned predictors or models. A node may have several input and/or output ports and several successor nodes, but only one predecessor node per input port. In general, a node has a configuration dialog in which the user can set various parameters, e.g. which file to read, how many cross validation runs should be performed or how large the constructed decision trees can grow. A node can be in any one of three states:

- not executable: not all of the input ports are connected to predecessors or the node is not configured correctly,
- ready for execution: all of the input ports are connected and the node is configured correctly, or
- executed: the input data/models have been processed and the results are available at its output port(s).

After a workflow has been built and its nodes have been configured properly, the user can either execute the whole workflow or only selected nodes. All necessary predecessors are executed automatically. For further details on KNIME's features see [6]. Internally, data is stored in a table-based format, built up of rows and columns. Each column has a specific type, e.g. strings, integers, doubles or more sophisticated types such as bit vectors, molecules or images. The data is not processed in a stream like way, i.e. the data is not forwarded in a constant flow. Rather, each node processes the whole data and afterwards forwards the entire results. This is much better suited to data mining tasks, as many algorithms need the whole data in advance.

In order to process huge amounts of data, the tables are not completely kept in memory but buffered on disk. As this slows down data transfer, the user may change this default behavior separately for each node provided sufficient memory is available. KNIME also includes the concept of *metanodes*. These nodes can be used to encapsulate sub-workflows to be reused at other locations inside the main workflow. Metanodes can encapsulate specific subtasks and thus hide complexity from the main flow. Metanodes can also be nested inside other metanodes. Besides these basic advantages, metanodes can also be used to represent loops in a workflow, e.g. for cross validation or feature selection (see Fig. 3 for an example). The metanode itself then deals with (repeatedly) executing the inner workflow and aggregating its results. This concept not only makes the workflow easier to understand than direct loops but is also more straightforward in terms of implementation. The framework simply executes a normal node while implementation of the particular node itself takes care of splitting the input data into partitions, running the small sub-workflow several times and aggregating the results at the end of the run (e.g. the cross validation sub-flow).

In this section we introduced general aspects of KNIME and described specific concepts representing potential capabilities for parallelization. The next section illustrates the parallel and distributed implementation of these concepts in more detail.

4. PARALLEL DATA PROCESSING IN KNIME

There are several ways to parallelize a typical workflow. The most simple and obvious one is the parallel execution of different branches in the workflow. Each node that is ready for execution can run in its own thread. The next, more advanced approach, is to process the data rows of the input table in parallel. The most sophisticated way of parallelization is to execute whole sub-workflows in parallel, e.g. the different iterations of a cross validation. Some algorithms allow for parallelization themselves, however this aspect is usually quite independent of the workflow. In the next subsections we will explain the different approaches in more detail and also take a brief look at what a programmer must do if (s)he wants to write a node that is capable of processing data in parallel.

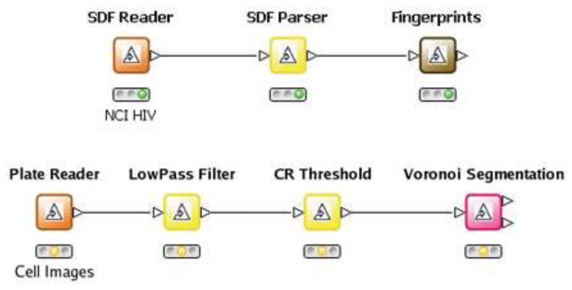


Figure 2. Two small workflows with threaded nodes

4.1 Parallel execution of independent nodes

As already mentioned, the most obvious method of parallel data processing is to execute several independent nodes at the same time. In Fig. 1 the two "File Readers" as well as the "Missing Value" nodes can be executed concurrently. The "Multilayer Perceptron Predictor", however, has to wait until its two predeccessors are finished.

In order to avoid overloading if too many nodes are ready for execution, KNIME uses a thread pool that is limited in size and reuses threads. The user can specify how many threads should be used at the same time. From the programmer's point of view, nothing needs to be done to allow parallel node executions. They are automatically handled by KNIME's workflow manager: it keeps track of queuing and executing the nodes in a workflow.

4.2 Parallel processing of data inside a single node

A considerable number of nodes (especially pre-processing nodes) perform computations based on single rows independently from the other rows. Examples of this type of node are shown in Fig. 2, which parse molecular representations and convert them into internal ones (e.g. the "SDF parser"), or nodes that manipulate image data (e.g. "Low Pass Filter" or "Voronoi Segmentation"). In KNIME these nodes are called "Threaded Nodes" and implementing them is not very different from implementing normal nodes. The framework takes care of splitting the input table into chunks; each chunk is processed in a separate thread, which is taken from the thread pool already mentioned in the previous section.

In order to achieve an equally distributed load among all threads on the one hand and a low overhead on the other, a suitable balance between the size and the number of chunks is important. Currently four times as many chunks as available threads in the pool are created. The abstract model provided by the framework calls a method in the concrete subclass for each row of the input table, which then returns the new cells that are appended to the row. In the final phase the results are merged and the complete output table is built. The programmer needs only be aware that the code is called by several threads concurrently. Therefore, synchronized blocks should be avoided and write-access to common data must be used carefully. Apart from that, the implementation resembles the normal node's API. Almost the same usage model applies in cases where the number of output rows differs from the number of input rows or when the structure of the output table is completely different from the input table structure. Again, the framework invokes the special implementation for each row of the input data. This time however, a (possibly empty) set of complete rows must be returned. Again, the framework takes care of merging the final results. The threaded nodes have a slight overhead that comes from splitting the input tables and merging the results. This depends largely on the size of a row (the number of columns and the size of the objects in the cells) and on I/O speed. Generally, however, this overhead does not significantly impair performance as we will demonstrate later.

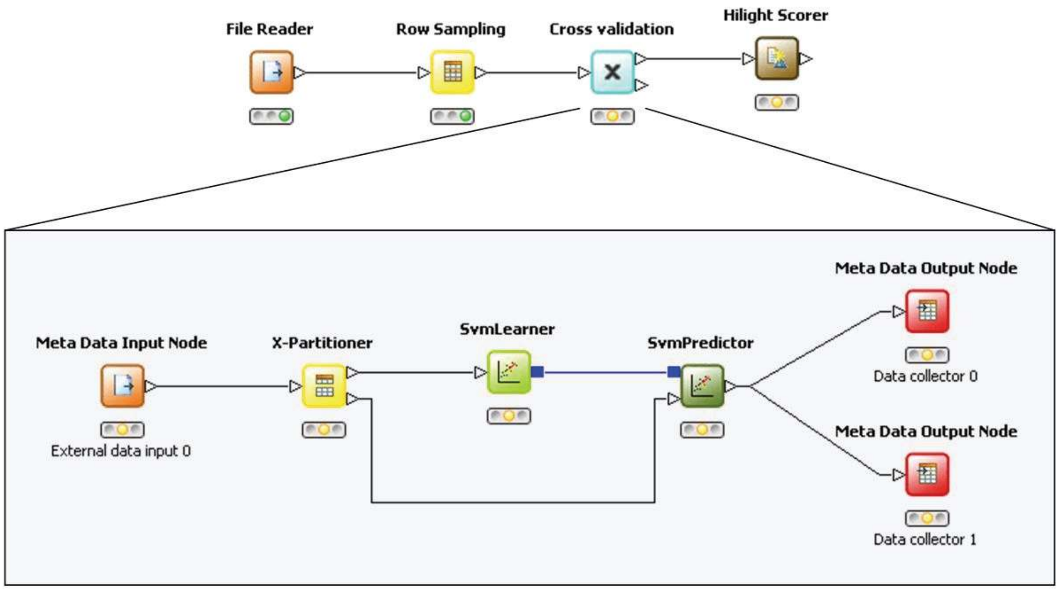


Figure 3. A workflow with a cross validation node and its internal sub-workflow

4.3 Parallel processing of sub-workflows

In Section 3 we explained the concept of metanodes, which encapsulate sub-workflows. Metanodes such as the looper or cross validation node implement additional functionality. Whereas the first node simply executes the inner flow a predefined number of times and merges the results, the latter also splits the input data into different partitions and aggregates the results at the end. Fig. 3 shows an example of how a cross validation node can be used. In the outer flow data is read, pre-processed and then fed into the cross validation node. Its internal flow contains four predefined nodes: the input node, which simply transports the data from the outer flow; two output nodes, which collect the results at the end; the X-Partitioner, which is responsible for splitting the data into training and tests sets. The user has to insert at least two nodes that build a model based on the training data and classify the test data using the model. The classification results must then be transferred to the output nodes. There are two output nodes, as the Cross Validation node has two outports, one for a short table with the errors of each iteration and one with the complete prediction results of all iterations In the example we use the SVM Learner node, which trains a support vector machine.

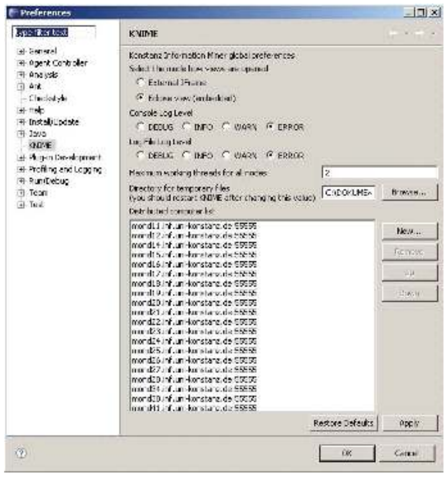
Using support vector machines is relatively time-consuming, even for small datasets, and becomes increasingly so if there are 10 or more iterations in a cross validation node. In such a case, all iterations are independent of each other, besides the fact that partitioning into training and test data must be identical, i.e. in each iteration another 9 of 10 partitions are used for training and the remaining partition is used for testing. Therefore, it is quite natural to parallelize execution of the single iterations. This time, however, the programmer of such a parallel metanode has a small amount of extra work to do.

The nodes inside the sub-workflow are not aware of the fact that they may be executed by several threads at the same time, i.e. they are not thread safe. For this reason it is necessary for our approach to create one copy of the sub-workflow for each iteration. This is accomplished inside the cross validation node, which also pre-executes each copy of the flow so that all nodes up to the partitioner node are executed and retain their individual state.

If an iteration is fully prepared and saved, it is put into a queue. Because the order of the results may play an important role, each single job has an index. As soon as a thread becomes available, execution is started. To assist this process, a thread pool is created: a sub-pool of the global pool. In the dialog of the particular metanode the user can specify how many threads the node should use to execute the internal workflow. Such a sub-pool shares the threads with its parent pool and thus may not use more threads than are already configured for the parent. After 9 of the 10 iterations (in our example) have been submitted to the queue, the cross validation node executes the 10th iteration by itself. This is necessary to ensure that the GUI representations show an executed state and the nodes have data tables at their output ports after the cross validation has finished. If this did not happen, the user's view would be inconsistent showing an outer cross validation node in its executed state but its inner nodes in an unexecuted state.

Finally, the cross validation node takes the results of the queue and merges them into the final output tables. As the cross validation thread does not use its reserved processing time while waiting for the results, it signals its "waiting state" to the thread pool which responds by creating an additional thread. To sum up, the following steps are necessary to implement a parallel metanode:

1. Create a submission queue into which the prepared sub-workflows are inserted and executed. A default implementation that works with threads taken from a sub-pool is provided by the framework.
2. Prepare the internal workflows, save them in a temporary directory and insert them into the queue. Methods for saving are also provided by the framework. The queue handles loading and executing the flow and returns the results of the execution.
3. Execute one (the last) iteration in the same thread in order to update the GUI components.
4. Collect the results from the queue, merge and return them.



Preferences

KNIME

Konstanz Information Miner global preferences

Select the mode how views are opened

External JFrame

Eclipse view (embedded) [✓]

Console Log Level

DEBUG INFO WARN [✓ ERROR]

Log File Log Level

DEBUG INFO WARN [✓ ERROR]

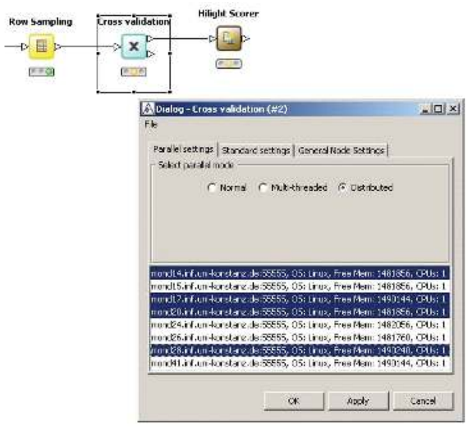
Maximum working threads for all nodes: 2

Directory for temporary files (you should restart KNIME after changing this value): C:\DOKUME~

Distributed computer list:

```
mondl11.inf.uni-konstanz.de:55555
mondl12.inf.uni-konstanz.de:55555
mondl14.inf.uni-konstanz.de:55555
mondl15.inf.uni-konstanz.de:55555
mondl16.inf.uni-konstanz.de:55555
mondl17.inf.uni-konstanz.de:55555
mondl18.inf.uni-konstanz.de:55555
mondl19.inf.uni-konstanz.de:55555
mondl20.inf.uni-konstanz.de:55555
mondl21.inf.uni-konstanz.de:55555
mondl22.inf.uni-konstanz.de:55555
mondl23.inf.uni-konstanz.de:55555
mondl24.inf.uni-konstanz.de:55555
mondl25.inf.uni-konstanz.de:55555
mondl26.inf.uni-konstanz.de:55555
mondl27.inf.uni-konstanz.de:55555
mondl28.inf.uni-konstanz.de:55555
mondl34.inf.uni-konstanz.de:55555
mondl30.inf.uni-konstanz.de:55555
mondl41.inf.uni-konstanz.de:55555
```

Figure 4. Registered distributed computers



Dialog - Cross validation (#2)

Select parallel mode: Normal Multi-threaded [✓ Distributed]

```
mondl14.inf.uni-konstanz.de:55555, OS: Linux, Free Mem: 1481856, CPUs: 1
mondl15.inf.uni-konstanz.de:55555, OS: Linux, Free Mem: 1481856, CPUs: 1
mondl17.inf.uni-konstanz.de:55555, OS: Linux, Free Mem: 1490144, CPUs: 1
mondl20.inf.uni-konstanz.de:55555, OS: Linux, Free Mem: 1481856, CPUs: 1
mondl24.inf.uni-konstanz.de:55555, OS: Linux, Free Mem: 1482056, CPUs: 1
mondl26.inf.uni-konstanz.de:55555, OS: Linux, Free Mem: 1481760, CPUs: 1
mondl28.inf.uni-konstanz.de:55555, OS: Linux, Free Mem: 1490240, CPUs: 1
mondl41.inf.uni-konstanz.de:55555, OS: Linux, Free Mem: 1490144, CPUs: 1
```

Figure 5. Selected computers from the list of computers that are accessible

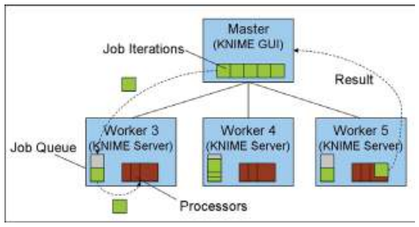


Figure 6. The master/worker approach

4.4 Distributed processing of sub-workflows

In addition to the previously described threaded parallelization, KNIME also enables the distributed execution of sub-workflows. Similar to the threaded version an implementor of a metanode can use the distributed functionality in a transparent manner, and does not need to pay attention to the details on distribution.

In the following, we describe the way KNIME distributes several iterations of the cross validation sub-workflow across distributed computers. The general idea is to set up a KNIME server component on each participating computer. The server has the full functionality of KNIME but runs without the user interface. Instead the server component accepts remote workflow jobs for execution. All computers intended to be used for distributed runs have to be registered to the KNIME global preferences as illustrated in Fig. 4.

Once a workflow has been created that includes a parallel executable metanode (in our case the cross validation node) it can be configured for distributed computing by selecting the appropriate option. Once this option is selected the user can choose the participating computers from a list of accessible registered computers (see Fig. 5).

After the run has been started, a single iteration of the cross validation procedure is assigned to each computer (similar to the threaded approach). This assignment contains the XML description of the sub-workflow and all the necessary data. For this reason, big data files can massively slow down communication between the computers. Therefore, the complexity of the computation inside the sub-workflow must be proportional to the data size.

The distribution scheme follows the classical master/worker approach where the master is represented by the computer on which the GUI is installed. Each worker supports the full KNIME functionality and exploits the threaded parallelism described in the previous sections. For this reason, each worker maintains a job queue from which jobs are assigned to the processors. Fig. 6 demonstrates this approach.

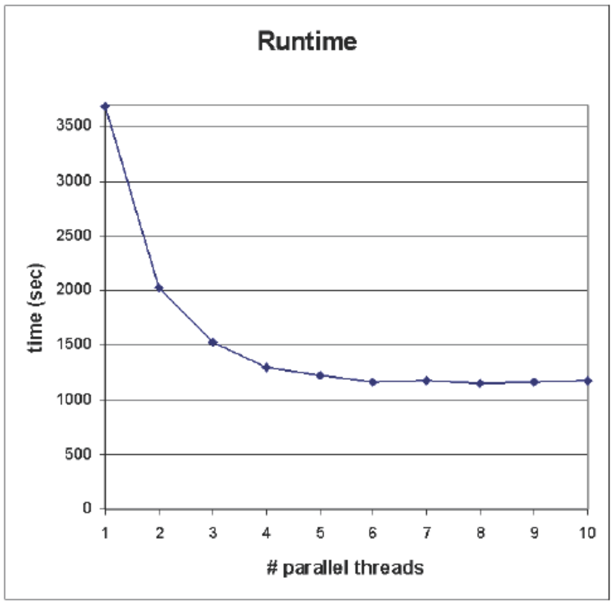
The master assigns to each worker as many jobs as the worker has available processors. Once a worker reports a result to the master node a new job (if available) is sent to the worker. This procedure is repeated until all iterations of the cross validation node have been executed and sent back by the workers. The master node then merges the partial results (as in the threaded approach).

5. EXPERIMENTS

In this section we present some experimental results on the parallel and distributed approaches presented above. The first approach in section 4.1 discussed the parallel execution of independent nodes. As this inherent parallelism is quite obvious and realized in nearly all pipelining tools we focus on the threaded processing of data inside a node (see Section 4.2) and the threaded and distributed approach of processing metanodes (see Sections 4.3 and 4.4).

The threaded tests have been run on a Tyan Transport VX50 (B4881) with 8 Dual Core AMD Opteron 870 CPUs running at 2.00 GHz. The system has 32GB of memory organized in a Non-Uniform Memory Architecture (NUMA) and runs the Red Hat Linux operating system and Sun's Java 1.5.

The first test measures a flow of nodes that process their data in parallel chunks as described in Section 4.2. The used flow (shown in Fig. 2) was taken from a research project in the field of cell-assay image classification[4]. The flow is the pre-processing part of a bigger flow. It reads 384 cell-assay images (Plate Reader), performs low pass filtering, then image thresholding to remove the background from the images (Otsu thresholding), and finally segments the cell-assays into single cell images by Voronoi segmentation. All these steps (except the image reader) process the images (each image represents a data row) independently and thus, have been implemented as threaded nodes. The graphs in Fig. 7 show the runtime for 1 to 10 parallel threads (40 chunks have been created for 10 parallel threads).



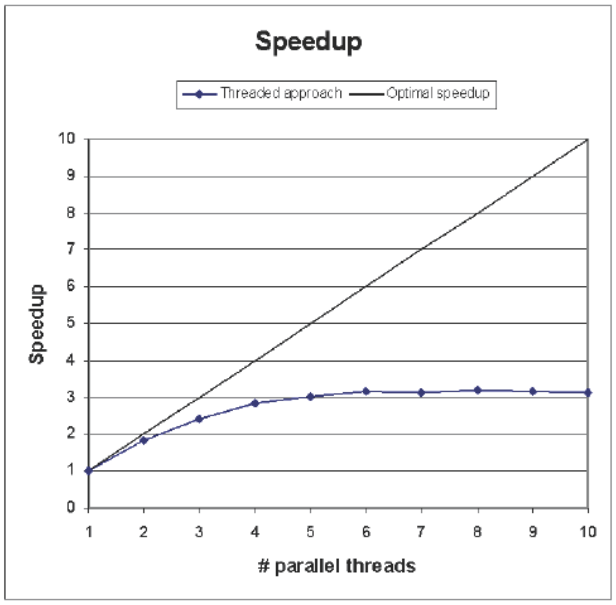


Figure 7. Runtime and speedup of the threaded node scenario

Unfortunately, taking more than 5 parallel threads does not reduce runtime. At first glance this might be considered surprising, as the other experiments described below show a much better speedup, however there are two main reasons for this effect. First, the used workflow analyzes image data. During its execution more than 5GB of data are read and written to disk. Despite the very fast disk array of the used system these I/O pauses cause degradation in speedup.

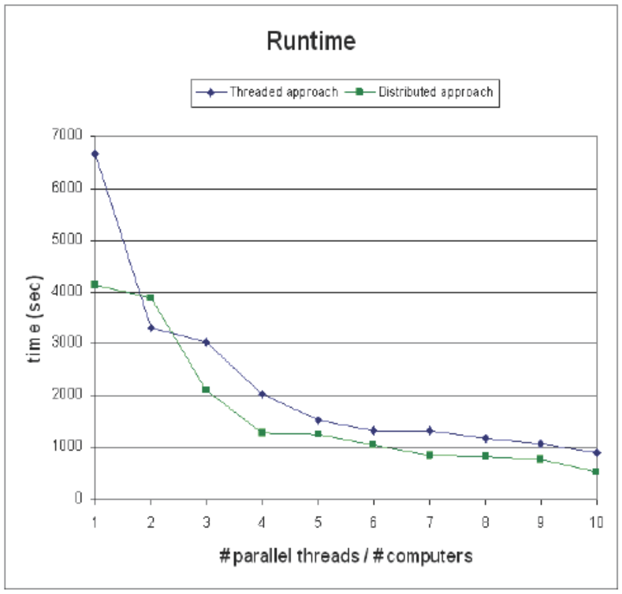
Table 1. System time and CPU usage for the "threaded node" workflow

Workflow threads	System time	Total Runtime	Total CPU usage
1	420.30s	3689.00s	109%
2	466.05s	2024.65s	202%
3	501.08s	1530.71s	284%
4	543.10s	1298.00s	366%
5	594.61s	1219.42s	449%
6	645.51s	1168.89s	513%
7	715.38s	1180.89s	573%
8	794.10s	1155.83s	648%
9	860.21s	1169.61s	703%
10	933.29s	1180.71s	758%

The values in table 1 support this conjecture: the number of threads used for executing nodes, the time the whole KNIME process spent inside the kernel (system time - mostly because of I/O), the total runtime and the total CPU usage are shown. As can be seen, the system time takes about 10% of the runtime with 1 thread and increases the more threads are used (the CPU usage is above 100% for 1 workflow thread, because there are other threads like the UI thread or the garbage collector that now and then occupy an additional free CPU). At the same time, the CPU usage does not rise at the same rate as the number of used threads, because the time a process/thread spends waiting for I/O is not directly attributed to the process. Another reason is the architecture of the Java runtime system. In Java there is one global heap to which all objects are allocated. Unfortunately, write access to the heap is internally synchronized by the virtual machine. Therefore, if a parallel algorithm allocates a lot of memory, its threads will very likely block each other while trying to access the heap. This effect has already been described in more detail in [14].

The second test describes the performance measurement of parallel executing metanodes. This test is executed with parallel threads and distributed computers. We use two flows with different characteristics for our tests. The first flow is the same as shown in Fig. 3. The outer flow reads in the data, samples a subset of data rows and then performs a 10-fold cross validation. The cross validation sub-workflow tests a Support Vector Machine (SVM) with a quadratic kernel. For this first test we used the shuttle data set from the UCI repository [16]. The shuttle data has 58,000 rows and 10 columns; its size is about 1.8 MByte and after sampling 40% of the data, we applied 23,200 rows with about 0.8 MByte to the cross validation node. Due to the quadratic kernel sub sampling is necessary to attain reasonable runtime for our experiments.

In the following we compare the threaded approach to the distributed approach. The distributed environment comprises 10 ordinary desktop PCs. Each machine has a 3.4 GHz Intel Pentium 4 (32 Bit architecture) processor with 1 GB of RAM. The PCs are connected via an ordinary LAN with a 100 MBit transfer rate.



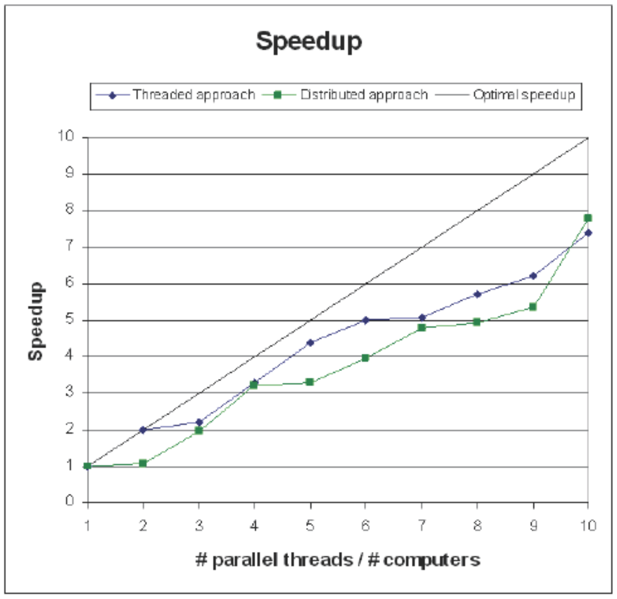
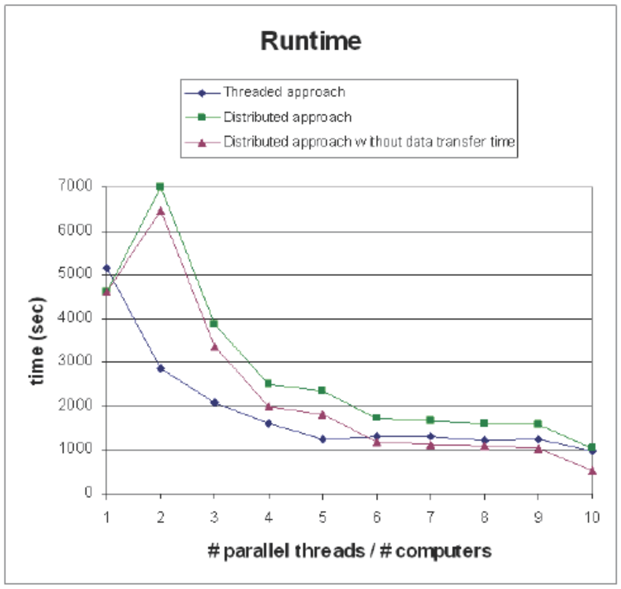


Figure 8. Runtime and speedup of the cross validation scenario (SVM) applied to the shuttle data

Fig. 8 shows the runtime and speedup analysis for both the threaded and the distributed approach. As shown in the speedup graph, both approaches perform well compared to the optimal linear speedup. Both graphs show a strong step characteristic. This is due to the coarse job granularity (one iteration represents one job). This is especially true if all computers perform equally. In this case the master simultaneously assigns one job to each machine and receives the results at almost exactly the same time followed by the next assignment round. In the event that the jobs cannot be equally distributed among the processors, some computers may remain idle in the last assignment round. If *p* is the number of processors and *j* is the number of jobs, (*p-j* mod *p*) processors will remain idle. In the distributed case the number of processors and the number of jobs must be reduced by 1, as the master performs just one job. The higher the number of idle computers for *p* processors the lower the speedup value compared to the speedup value for *p-1* processors.

In the distributed approach it is also notable that the runtime improvement from 1 to 2 computers is not significant. Also the step characteristic is shifted compared to the threaded approach. The reason is that the central master node only performs one iteration of the metanode (see Section 4.4); i.e. almost all work is performed by the second computer. This impact is reduced the more computers are involved in the execution. Ultimately, for 10 computers, each one performs exactly one run. This yields the strong speedup improvement from 9 to 10 machines.



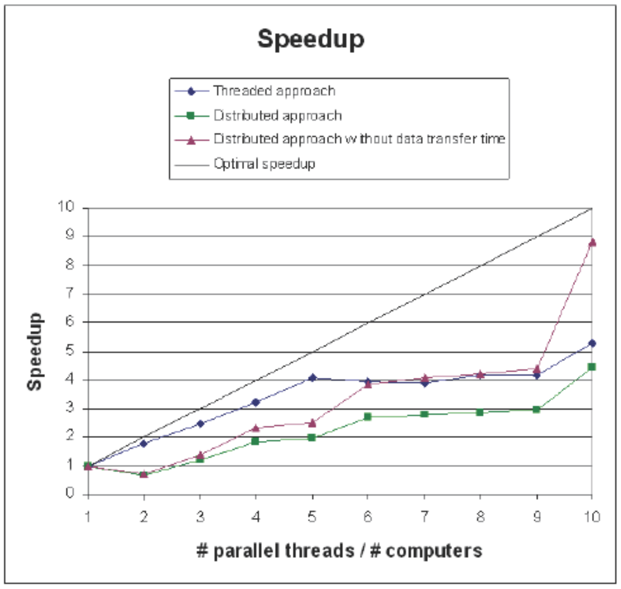


Figure 9. Runtime and speedup of the cross validation scenario (kNN) applied to the Océ data

The scenario described here employs a relatively small data set and the computationally expensive SVM (quadratic kernel). The second batch of tests changes this setup. Instead of the SVM a much faster k-Nearest-Neighbor (kNN) classifier is used inside the cross validation node (again 10-fold). As data input we applied a real-world data set [7] which consists of 177,655 feature vectors extracted from handwritten numerical characters. Each vector consists of 116 features (columns). The data set is about 106 MByte in size and is used to test character recognition engines. To reduce the data set to a reasonable size, we created a subsample of 30%, i.e. 53,297 feature vectors (about 32 MByte). Fig. 9 shows the runtime and speedup behavior of the described experiment.

Compared to the first test speedup has decreased. The distributed approach suffers particularly from the considerably higher communication costs due to the required transfer of large datasets. The overall communication time for a complete run of all 10 cross validation iterations is about 522 seconds. The assignment of one job (master to worker), which includes the input data, takes about 51 seconds; return of results (worker to master) takes about 7 seconds (about 4 MBytes).

This impact is illustrated by the graph and does not include the transfer time.

Similar to the scenario with the small data set, the speedup graphs show a slight step characteristic and the strong speedup jump from 9 to 10 threads/computers.

The longer runtime from 1 to 2 computers in the distributed case is a result of the bigger parallel overhead of the second scenario. Even after removing the data transfer time the runtime is greater than the runtime for simply one machine. This is due to the effort required to create an explicit description of the sub-workflow at the master node (saving it into a single file) and to restore it at the worker node. The same also holds for the result data that is sent back.

6. RELATED APPROACHES IN OTHER TOOLS

Looking at how other data analysis tools deal with parallel and distributed processing is not easy. Besides D2K all other popular programs are commercial and their owners do not provide much information about internal functionality. D2K has the ability to create sub-workflows, which can be executed on other computers by using a client-server concept similar to the one described here. There are no remarks on using several threads on one computer or executing iterations of sub-workflows in a parallel or distributed manner. Scitegic's Pipeline Pilot offers the possibility to execute nodes remotely but in the publicly available documentation there are no references to parallel or distributed execution that really speed up the workflow for the end user. Insightful Miner exploits pipeline parallelism whereby several nodes work on the data stream at the same time. This however comes to an end as soon as one of the nodes requires all the data in order to compute – which is especially important for data mining algorithms that need all the data for the learning process. KNIME exploits this parallelism, which requires independent data rows with the concept of threaded nodes as presented in section 4.2. The threaded node model also keeps the available processors busy but ensures that a predecessor node finishes its execution before forwarding the results. Pervasive's DataRush also exploits pipeline parallelism, however the visual tools are only for visualization of the flow, not for construction purposes; the flow must be assembled and configured via an XML file. InforSense KDE is able to execute independent branches in parallel on the same computer. As far as can be judged from the released information about the commercial products none offers the entire variety of alternatives to speed up a workflow by using parallel or distributed processing as discussed here.

7. CONCLUSIONS

We have presented several ways of speeding up data analysis in pipelining tools by using the power of multiple CPUs (or cores) in one system on the one hand, and distributed computers on the other. Such approaches are necessary because in future the speed of a single processor will not increase as fast as it has done in the last decades, instead we will see a rise in multi-core environments. However, the amount of data requiring analysis will continue to grow at the same speed or even faster.

Unfortunately, most of today's programmers are not specially trained in writing parallelized code, making some kind of framework necessary. The data analysis platform KNIME already offers a simple API to develop new nodes for a workflow. With the extensions we described above, it is not substantially more difficult to add parallel processing to a node than programming a sequential one. Of course, additional overheads must normally be taken into account with such ease of use. In our experiments we showed that it is still possible to achieve good speedups by applying the provided framework. When using multiple threads, however, the I/O and Java runtime system become bottlenecks. Nevertheless, speedups of up to 7.5 on 10 CPUs are possible.

When executing sub-workflows on other computers this effect is not a problem. In this case, however, sending data between the computers introduces considerable overhead which impairs performance. On one hand, this impacts the speedup of sub-workflows that contain a lot of data but are not very computing-intensive; on the other hand, with the right proportion it is possible to achieve speedups of almost 8 with 10 computers.

There are still some areas of the framework that can be further optimized. For instance, transfer of sub-workflows to other computers can be made more efficient. A saved workflow may contain more data than is actually necessary for executing remaining nodes. If this data is left out, the communication overhead will decrease. After these issues have been solved, the described approaches will be evaluated with much bigger datasets and on many more processing units. Another area for further improvement involves extending the "threaded nodes" so that those chunks are executed on other computers.

REFERENCES

[1] R. Agrawal, T. Imielinski, A. N. Swami, "Mining association rules between sets of items in large databases", Proceedings of the 1993 ACM SIGMOD Intl. Conf. on Management of Data, Washington, D.C., USA, 1993. ACM Press, pp. 207-216.

[2] Michael R. Berthold, Nicolas Cebron, Fabian Dill, Giuseppe Di Fatta, Thomas R. Gabriel, Florian Georg, Thorsten Meinl, Peter Ohl, Christoph Sieb, Bernd Wiswedel, "KNIME: the Konstanz Information Miner", Proceedings 4th Annual Industrial Simulation Conference, Workshop on Multi-Agent Systems and Simulation (ISC 2006), 2006.

[3] Christian Borgelt, Michael R. Berthold, "Mining molecular fragments: Finding relevant substructures of molecules", Proceedings of the IEEE Intl. Conf. on Data Mining ICDM, Piscataway, NJ, USA, 2002, IEEE Press, pp. 51-58.

[4] Nicolas Cebron, Michael R. Berthold, "Adaptive active classification of cell assay images", in Knowledge Discovery

in Databases: PKDD 2006 (PKDD/ECML), Vol. 4213, Springer Berlin / Heidelberg, 2006, pp. 79-90.

[5] The Eclipse Foundation, “The Eclipse Project”, available at [http://www.eclipse.org/], last accessed date: 3/2/2007.

[6] ALTANA Chair for Bioinformatics & Information Mining at the University of Konstanz, “KNIME - Konstanz Information Miner”, available at [http://www knime.org/], last accessed date: 3/2/2007.

[7] Océ Document Technologies GmbH. Dataset of 177,655 feature vectors extracted from handwritten numerical characters, 2006.

[8] Michael I. Gordon, William Thies, Saman Amarasinghe, “Exploiting coarse-grained task, data, and pipeline parallelism in stream programs”, in ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems, New York, NY, USA, 2006. ACM Press, pp. 151-162.

[9] Jiawei Han, Micheline Kamber, “Data Mining - Concepts and Techniques”, 2nd edition, Morgan Kaufmann, 2006, ISBN:1-55860-901-6.

[10] David Hand, Heikki Mannila, Padhraic Smyth, “Principles of Data Mining”, The MIT Press, 2001, ISBN-13: 978-0262082907.

[11] InforSense, “InforSense KDE”, available at [http://www.inforsense.com/kde.html], last accessed date: 3/2/2007.

[12] Insightful, “Insightful Miner”, available at [http://www.insightful.com/products/iminer/default.asp], last accessed date: 3/2/2007.

[13] Claudia Leopold. Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches. Wiley, 2000, ISBN-13: 978-0471358312.

[14] Thorsten Meinl, Marc Wörlein, Ingrid Fischer, Michael Philippsen, “Mining molecular datasets on symmetric multiprocessor systems”, Proceedings of the 2006 IEEE International Conference on Systems, Man and Cybernetics, 2006, IEEE Press, pp. 1269-1274.

[15] MPI Forum, “MPI specifications”, available at [http://www mpi-forum.org/docs], last accessed date: 3/2/2007.

[16] David J. Newman, Seth Hettich, C. L. Blake, C. J. Merz, “UCI repository of machine learning databases”, http://www.ics.uci.edu/~mlearn/MLRepository.html, 1998.

[17] Scitegic, “Pipeline Pilot”, available at [http://www.scitegic.com/products/overview/], last accessed date: 3/2/2007.

[18] Christoph Sieb, Giuseppe Di Fatta, Michael R. Berthold, “A hierarchical distributed approach for mining molecular fragments”, Proceedings of the International Workshop on Parallel Data Mining (PKDD/ECML 2006), 2006, pp. 25-37.

[19] Pervasive Software, “Pervasive DataRush”, available at [http://www.datarush.org], last accessed date: 3/2/2007.

[20] James J. Thomas and Kristin A. Cook, editors. Illuminating The Path: Research and Development Agenda for Visual Analytics. IEEE Press, 2005, ISBN 0-7695-2323-4.

[21] Automated Learning Group University of Illinois, “D2K”, available at [http://alg ncsa.uiuc.edu/do/tools/d2k], last accessed date: 3/2/2007.

[22] Jack van Wijk, “The value of visualization”, Proc. IEEE Visualization 2005, 2005, pp. 79-86.

[23] Ian H. Witten and Eibe Frank. Data Mining - Practical Machine Learning Tools and Techniques. Elsevier, 2nd edition, 2005, ISBN-13: 978-0120884070.

[24] Cheng-Zhong Xu, Francis C.M. Lau, “Iterative dynamic load balancing in multicomputers”, Journal of the Operational Research Society, Vol. 43, No.7, July 1994, pp. 786-796.

[25] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Mitsunori Ogihara, Wei Li, “New algorithms for fast discovery of association rules”, in Proceedings of 3rd Intl. Conf. on Knowledge Discovery and Data Mining, AAAI Press, 1997, pp. 283-296.

Biographies

Christoph Sieb received his first diploma in Business Informatics from the University of Cooperative Education in Stuttgart, Germany in 2001. While writing his Diploma Thesis at the IBM Research and Development Labs Boeblingen, Germany he worked on parallel clustering algorithms. Subsequently, he joined IBM for 2 years as a software engineer in the field of commercial information systems. He received his M.Sc. in Computer Science from the University of Konstanz in 2005. In November 2005 he joined the ALTANA-Chair for Bioinformatics and Information Mining at Konstanz University as a PhD student doing research in the field of parallel and distributed Data Mining and Machine Learning.

Thorsten Meinl received his diploma in Computer Science from the University Erlangen-Nuremberg in July 2004. He then spent two years with the Programming Systems group in Erlangen where he worked on parallel fragment search in molecular databases. Since March 2006 he has been at the ALTANA Chair in Konstanz as a PhD student and is doing research in the field of virtual high throughput screening.

Michael R. Berthold received his PhD from Karlsruhe University. He then spent over seven years in the US, among others at Carnegie Mellon University, Intel Corporation, the University of California at Berkeley and - most recently - as director of an industrial think tank in South San Francisco. Since August 2003 he holds the ALTANA-Chair for Bioinformatics and Information Mining at Konstanz University, Germany where his research focuses on using machine learning methods for the interactive analysis of large information repositories in the Life Sciences. M. Berthold is Past President of the North American Fuzzy Information Processing Society, Associate Editor of several journals and a Vice President of the IEEE System, Man, and Cybernetics Society. He has been involved in the organization of various conferences, most notably the IDA-series of symposia on Intelligent Data Analysis and the conference series on Computational Life Science. Together with David Hand he co-edited the successful textbook, ”Intelligent Data Analysis: An Introduction”, which has recently appeared in a completely revised second edition.