Parallel Data Mining on a Beowulf Cluster

Peter Christen*, Ole M. Nielsen, Markus Hegland, and Peter Strazdins

Australian National University,
Canberra, ACT 0200, Australia

URL: `http://csl.anu.edu.au/ml/dm/`

Abstract. This paper presents a parallel data mining application for predictive modelling running on a Beowulf style Linux cluster. Data mining or Knowledge Discovery in Databases (KDD) is the process of analysing large and complex data sets with the purpose of extracting useful and previously unknown knowledge. The task of predictive modelling is the prediction of an attribute according to a model built with one or more other attributes given in a data collection. We describe two methods for predictive modelling of high-dimensional data sets, namely ADDFIT which implements additive models, and HISURF which uses wavelets for high-dimensional surface smoothing, and present a parallel implementation on a distributed memory cluster architecture which uses the scripting language Python as a flexible front-end to facilitate user-interaction, control the parallel application, and generate graphical outputs.

Keywords: Predictive modelling, Additive models, Python, MPI

1 Introduction

The computerisation of business transactions and the use of bar codes in commercial outlets are providing businesses with increasingly large amounts of data, and the growth of stored data has been explosive in recent years. Terabyte databases are common, with Gigabytes added every day. In science, for example, projects like the Human Genome Project deal with Terabytes of data.

With *Data Mining* one seeks techniques to automatically process and detect patterns in these very large data sets [15]. Revealing patterns and relationships in a data set can improve the missions and objectives of many organisations and research projects. For example, sales records can reveal highly profitable retail sales patterns. Data mining can be used for spotting trends in data that may not be easily detectable by traditional database query tools that rely on simple queries (like SQL statements), or on simple statistical data analysis. It is therefore important to develop automatic techniques that are able to reveal hidden relationships and patterns as well as uncover correlations [12] in very large data sets.

* Corresponding author, E-Mail: Peter.Christen@anu.edu.au

Algorithms applied in data mining have to deal with two major challenges: Large data sets and high dimensions (many attributes). In recent years, data sets had the size of Gigabytes, but Terabyte data collections are now being used in business and the first Petabyte collections are appearing in science [10]. It has also been suggested that the size of databases in an average company doubles every 18 months [4] which is similar to the growth of hardware performance according to Moore's law. Consequently, data mining algorithms have to be able to scale from smaller to larger data sizes when more data becomes available. The complexity of data is also growing as more attributes tend to be logged in each record. Data mining algorithms must, therefore, be able to handle high dimensions in order to process such data sets.

This combination of large data sizes with high data complexity poses a tough challenge for all data mining algorithms. Moreover, algorithms which do not scale linearly with the data size are not feasible. Parallel processing can help to tackle larger problems and to get reasonable response times. In this paper, we present scalable parallel algorithms for predictive modelling and high dimensional surface fitting that successfully deal with these issues.

An important technique applied in data mining is multivariate regression, which is used to determine functional relationships in high dimensional data sets. A major difficulty which one faces when applying non-parametric methods is that the complexity grows exponentially with the dimension of the data set. This has been called the *curse of dimensionality*. In Section 2 we present two algorithms for predictive modelling and explain how they deal with this curse. The implementation of this methods on a Beowulf style Linux cluster is discussed in Section 3 and in Section 4 we present results using a public available census database. Section 5 finalises this paper with conclusions and gives an outlook to future work.

1.1 Parallel Data Mining

Parallel processing can help both to tackle larger problems and to get reasonable response times. It is not only that more computing power becomes available, but equally important is the increased I/O bandwidth and larger memory provided by most parallel machines.

Parallel data mining is a hot research topic (see [31] for recent research papers), as the need for parallel processing is clearly given by the huge and increasing data collections available. The requirements for parallel data mining systems [23] include not only parallel scalable hardware platforms, parallel I/O and databases, and parallel data mining algorithms, but also frameworks for rapid algorithm development and evaluation. Issues like security, fault tolerance, heterogeneous data access and representation, quality of service, pricing and portability have to be addressed as well. Large-scale parallel data mining systems should support the entire data mining process, including pre- and post-processing.

For the development of our algorithms we are using a Beowulf [26] style Linux cluster with 96 processing nodes at the Australian National University [1]. This

type of parallel architecture is a cost efficient alternative to expensive supercomputers, and therefore specially useful for algorithm development. A more detailed description of the used cluster is given in Section 4.

2 Scalable Parallel Predictive Modelling

Predictive modelling is widely used in data mining to discover average relationships or trends between numerical and categorical attributes which can then be applied to estimate properties of future objects or transactions. In particular, the discovered models can be used to find unusual data records. Such records could, in some case, be indicative to fraud. In general, they point to exceptional circumstances which suggest further investigation and possibly action.

A predictive model is described by a function $y = f(\boldsymbol{x})$, where $\boldsymbol{x} = (x_1, \ldots, x_d)$ belongs to the set T of predictors and y belongs to the set S of responses. If the set S is finite (often $S = \{0, 1\}$), the determination of f is a *classification problem* and if $S = \mathbb{R}$ (the set of real numbers) one speaks of *regression*. In the following we will mainly consider the case of regression. Note, however, that the techniques discussed here can be modified to include *logistic regression* which is used for classification.

In many applications the response variable y is known to depend smoothly on the predictor variables x_i. This allows the application of approximation results which have successively been applied in engineering and science. There, the finite element methods provide a very efficient computational foundation for the approximation of effects as diverse as mechanical stress, electrical fields and fluid flow vorticity. It has been shown earlier that such finite element approximations are also efficient for the approximation of smooth regression functions [18]. In principle, these approximations all use linear representations by generating or basis functions of the form

$$f(\boldsymbol{x}) = \sum_{j=1}^{m} c_j \beta_j(\boldsymbol{x}). \tag{1}$$

As the β_j are not necessarily linearly dependent, linear constraints are used to provide a one-to-one correspondence between the coefficients c_j and the functions f.

The data is modelled as a sequence of pairs $(\boldsymbol{x}^i, y^i) \in T \times \mathbb{R}$ where $i = 1, \ldots, n$ are n data records (in data mining applications n is typically in the range of 10^6 or higher). The function f should fit the data in the sense that $f(\boldsymbol{x}^{(i)}) \approx y^{(i)}$ but it should also be smooth. A good compromise between smoothness and data fit is obtained with a penalised least squares fit where f minimises a functional of the form

$$J_\alpha(f) = \sum_{i=1}^{n} \left(f(\boldsymbol{x}^{(i)}) - y^{(i)} \right)^2 + \alpha \int_T |\mathcal{L}(\boldsymbol{x})|^2 \, dx \tag{2}$$

where the smoothing parameter α controls the trade-off between smoothness and data fit. If $\alpha = 0$ f in (2) will be an interpolant. The symbol $\mathcal{L}$ denotes a

differential operator, examples include the gradient, Hessian and the Laplacian, and $|\cdot|$ denotes the Euclidean norm for vector operators.

While the actual application can only be developed when a given set of generating or basis functions β_j is chosen (in the following subsections two particular cases will be presented), several aspects are independent of the particular choices, however, and they will be discussed here. First, the coefficients c_i are determined by a quadratic optimisation problem, in particular, the minimiser

$$Q_\alpha(\boldsymbol{c}) = \boldsymbol{c}^T\boldsymbol{A}\boldsymbol{c} + 2\boldsymbol{b}^T\boldsymbol{c} + \alpha\boldsymbol{c}^T\boldsymbol{C}\boldsymbol{c}$$

possibly with additional constraints. The assembly of the finite element matrix $\boldsymbol{C}$ is standard for finite element calculations and the matrix elements are

$$C_{j,k} = \int_T (\mathcal{L}\beta_j(\boldsymbol{x}))^T \mathcal{L}\beta_k(\boldsymbol{x})dx.$$

The matrix $\boldsymbol{A}$ and vector $\boldsymbol{b}$ are assembled from the data. Their components are

$$A_{j,k} = \sum_{i=1}^{n} \beta_j(\boldsymbol{x}^{(i)})\beta_k(\boldsymbol{x}^{(i)}) \tag{3}$$

and

$$b_j = \sum_{i=1}^{n} \beta_j(\boldsymbol{x}^{(i)})y^{(i)} \tag{4}$$

respectively. Thus the determination of the functions f consists of two steps:

1. **Assembly**: The computation of the matrices $\boldsymbol{A}$ and $\boldsymbol{C}$ and the right-hand side $\boldsymbol{b}$. This computation is expensive due to the large number of data records n. However, it is scalable in the data size, i.e. it has linear time complexity $O(n)$. It is, however, $O(m^2)$ and a large set of generating functions m can make this computation infeasible. The number of generating functions required is strongly influenced by the dimension of the problem. However, in data mining applications, $m \ll n$ so that this step corresponds to a data reduction.
   At the end of this step the penalty term $\alpha\boldsymbol{C}$ is added to the matrix $\boldsymbol{A}$.
2. **Solution**: The linear system of equations $(\boldsymbol{A} + \alpha\boldsymbol{C})\boldsymbol{c} = \boldsymbol{b}$ is solved using a direct solver. The solution does not require reading the data and thus has a time complexity independent of the data size. Typically, for a size m system we require $O(m^3)$.

Note that for large n step 1 will dominate whereas for large m step 2 will dominate. As the number of data records n is usually very large for data mining applications, the overall complexity is mainly determined by n.

In this paper we discus algorithms for two particular choices of the basis vectors β_j. We call these algorithms ADDFIT and HISURF respectively, and they are presented in more details in the following two subsections. A third method using a thin plate splines approach and called TPSFEM is presented elsewhere [7].

A dense linear system is assembled for both ADDFIT and HISURF, so the same solver can be used to solve theses systems. TPSFEM on the other hand results in a sparse linear system, which requires different solving techniques.

The process of assembling the linear systems has the same structure both for ADDFIT and HISURF. For each data record, some nonzero elements are added into the matrix and vector. The number of nonzero elements per data record is $O(d)$ (assuming a d-dimensional data set), forming the normal equations matrix $\boldsymbol{A}$ is thus of order $O(d^2)$ for each data record. The total complexity of assembling n data records sequentially is therefore

$$T_{assem}(1) = O(d^2 n).$$

The assembly of data records into the linear system is additive and so each record can be assembled independently from all others. Having p processing nodes available, each of them can read a fraction n/p of all data records, and a local linear system is assembled on each node without communication. The parallel complexity of the assembly process on p processing nodes therefore becomes

$$T_{assem}(p) = O\left(\frac{d^2 n}{p}\right).$$

The complete matrix data structure has to be stored on each node, as every data record can contribute nonzero elements anywhere in the matrix. The linear system is therefore distributed, but not replicated, on the processing nodes and the final linear system is the sum of all local linear systems.

The assembly process without communication is limited by the available amount of memory on each processing node. For matrices that are too large, a more complex assembly has to be applied, where the matrix data has to be distributed in a memory-scalable way. A blocking structure of reading and re-distributing data can be used for this.

2.1 High-dimensional Surface Smoothing using Wavelets

This section describes the High Dimensional Surface Smoothing (HISURF) method. It uses a hierarchical interpolatory wavelet basis and tensor products thereof to approximate f as given in (2). Wavelets provide a multi-level decomposition of f in each dimension which can lead to very compact approximations of reasonable well behaved functions. See [8, 24, 27] for introductions to wavelet theory.

Let $I_{j,l} = \left[\frac{l}{2^j}; \frac{l+1}{2^j}\right]$ be an interval. The one-dimensional hierarchical basis functions are defined as

$$\beta_{j,l} = \begin{cases} \varphi_{0,l}, & j = 0 \\ \psi_{j-1,l}, & j > 0 \end{cases} \tag{5}$$

where

$$\varphi_{j,l}(x) = \begin{cases} 1 + 2^j x - l, & x \in I_{j,l-1} \\ 1 - 2^j x + l, & x \in I_{j,l} \\ 0, & \text{otherwise} \end{cases}$$

and $\psi_{j-1,l} = \varphi_{j,2l+1}, l = 0, \ldots, 2^j$. The high dimensional basis functions are formed as the tensorial multi dimensional wavelet functions $\boldsymbol{\beta}_{\boldsymbol{j},\boldsymbol{l}} = \bigotimes_{s=1}^{d} \beta_{j_s,l_s}$ where $\boldsymbol{j} = (j_1, \ldots, j_d)^T$ is a vector of scales in each dimension and $\boldsymbol{l} = (l_1, \ldots, l_d)^T$ is a vector of positions in each dimension.

Let $j \in Z$ be a fixed maximal resolution, let $\gamma = 2^j + 1$ be the number of grid points in each dimension, and let U_j^d be the space generated by piecewise linear d-dimensional functions interpolated from the γ^d total grid points. These *hat* functions comprise the generic finite element basis widely used in the literature and in finite element methods. However, it can be shown [20] that the functions $\boldsymbol{\beta}_{\boldsymbol{j},\boldsymbol{l}}$ also form a basis for U_j^d. We call this basis the *rectangular wavelet basis*.

A function $f(x_1, \ldots, x_d) \in U_j^d$ then has an expansion in terms of the rectangular basis as follows:

$$u = \sum_{j_1,\ldots,j_d=0}^{j} \sum_{l_1=0}^{\theta(j_1)} \cdots \sum_{l_d=0}^{\theta(j_d)} d_{\boldsymbol{j},\boldsymbol{l}} \boldsymbol{\beta}_{\boldsymbol{j},\boldsymbol{l}} \tag{6}$$

where the function $\theta(i)$ denotes the last local index of coefficients at scale i, defined as $\theta(i) = 2^{|i-1|} - 1, \quad i \geq 0$.

It can be shown [20] that terms of this expansion where $j_1 + \cdots + j_d > j$ can be deactivated without sacrificing the essential approximation power.

The compression error is bounded by the expression const$\binom{j+d-1}{j}2^{-2j}$ where the constant depends only on the smoothness of f. In return, the dimension m of the compressed system is bounded by $j^{d-1}(2^{j+1} - 1)$, which is a significant reduction compared to computing the full surface [20], especially for large j and d. The approximated smoothing surface is then computed in terms the active coefficients only.

This is data *independent* or a-priori compression as opposed to the more common (in the wavelet literature) data dependent compression where wavelet coefficients are discarded based on their magnitude. The data dependent compression is efficient for a function with isolated singularities. On the other hand, for fitting a high dimensional smooth surface, singularities are unlikely to occur so good approximations can be achieved by using a-priori compression. Since this compression scheme is data-independent the algorithm can be very fast.

2.2 Additive Models

In this section we describe our method for Additive Model Fitting called ADDFIT. Functions of d variables can be represented as sums of the form

$$f(x_1, \ldots, x_d) = f_0 + \sum_{i=1}^{d} f_i(x_i) + \sum_{i<j} f_{i,j}(x_i, x_j) + \cdots .$$

Such decompositions originate from the Analysis of Variance and have thus been called *ANOVA-decompositions* [11]. They can be viewed as generalisations of Taylor and Fourier-series. However, the terms are only uniquely determined if

additional constraints are imposed. If this is not done, the component $f_1(x_1)$, for example, is a special case of $f_{1,2}(x_1, x_2)$ and thus cannot be determined.

Including so-called *interaction terms* $f_{i_1,\ldots,i_k}(x_{i_1}, \ldots, x_{i_k})$ up to order $k = d$ allows the exact representation of f. This, however, is computationally infeasible in general due to the *curse of dimensionality* which also poses a major challenge to HISURF. Luckily, for high-dimensional data only the inclusion of lower-order terms is required as they give approximations which converge with the dimension d for smooth functions [19]. Practical algorithms [3, 13, 16, 30] typically give good approximations for $k = 1$ or $k = 2$ and it is common folklore that interactions with higher order than $k = 5$ are highly unusual. (Of course, one also requires enough data in order to identify such high-dimensional interactions.) The terms in the ANOVA decomposition are represented using the same basis functions which have been used for HISURF. The more general cases will be discussed elsewhere, here we only discuss functions of the form

$$f(\boldsymbol{x}) = f_0 + \sum_{s=1}^{d} f_s(x_s).$$

These *additive models* are discussed extensively in [16] from a statistical viewpoint. The predictor variables (or attributes) x_s can be real numbers, categories or even more complex objects like sets, graphs and vectors. Vectors allow the inclusion of higher order interactions. In the following, however, only simple data types (real and categorical) will be discussed.

Additive models have many advantages. They are easy to interpret as the overall effect is given as a sum of effects of single variables. When interpreting additive models, however, one has to take into account that the variables x_s might be correlated. Our implementation of additive models uses a basis representation of the component functions f_s as

$$f_s(x_s) = \sum_{i=1}^{\gamma_s} c_{s,i}\beta_{s,i}(x_s)$$

with the basis functions $\beta_{s,i}$, the coefficients $c_{s,i}$ and where γ_s is the number of basis functions characterising f_s. The basis functions are such that for any x_s only a small number of basis functions have nonzero values. For categorical variables the basis functions are just the category indicator functions and for real variables we use piecewise linear functions. A difficulty is that these functions may be linearly dependent over the data set and this has to be addressed with constraints.

While the components of the sum defining $\boldsymbol{A}$ in Equation (3) are typically sparse, the final matrix $\boldsymbol{A}$ is more or less dense. In the parallel implementation each processing node needs storage of the size of $\boldsymbol{A}$ to store the partial sum. This means that we will not be able to increase the accuracy of the model in a scalable way with the numbers of nodes, but this was not essential for our project. The accuracy of the estimate is given by the accuracy of the model (bias) and the

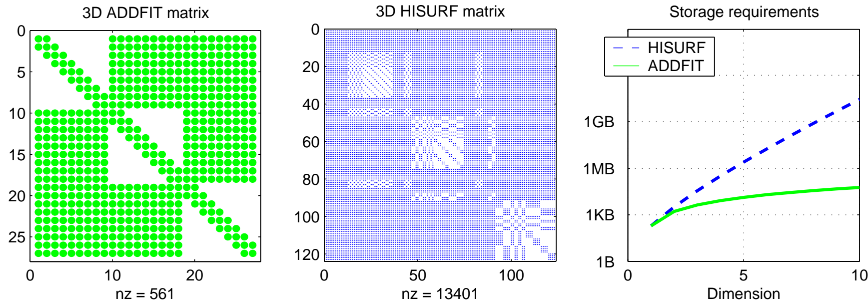

Fig. 1. Matrix structures and storage requirements.

variance of the estimate. The number of data records controls the variance of the estimate.

The total number of basis functions used is $m = 1 + \sum_{s=1}^{d} \gamma_s$. Thus, for constant $\gamma_s = \gamma$ the number of unknowns – and thus the size of the assembled linear system – scales linearly with the dimension d,

$$m = 1 + d\gamma.$$

This scalability with dimension d is an import advantage of additive models.

Figure 1 shows the structure of matrices assembled for a 3-dimensional data set for both ADDFIT and HISURF. It also shows the increasing size of the assembled linear systems with increasing dimensionality. However, computing a full surface will grow exponentially with dimensions.

3 Cluster Implementation

First prototypes of the presented predictive modelling algorithms have been developed by our group over the last couple of years [6, 7, 18–20] using Matlab and the scripting language Python. The ADDFIT method has been used in an administrative health data mining research consultancy [17] to predict the behaviour of pathology laboratories.

To increase the performance and being able to use larger data sets we gradually implemented parts of the algorithm in C. In a first step a C version of the assembly routine for ADDFIT has been embedded into the Python code, allowing both faster execution and the assembly of linear systems using larger data sets. Next we developed a parallel version of the ADDFIT assembly routine [6] using MPI [21] for communication, while a parallel dense solver for indefinite linear systems [28] has been modified to fit into our framework. By now, we have a parallel implementation for the ADDFIT and HISURF methods, including both the assembly and solving stages. We are using a Python wrapper code to facilitate the user interface and present graphical output. We describe both the Python wrapper and the parallel implementation in the following subsections.

```
num_proc = 10                                   # Specify number of processes to use
alpha = 0.001                                           # Select smoothing parameter
solver_args = [1,1,64,64]                              # Parameters for dense solver

stat_dict = load_stat('CENSUS')                       # Load CENSUS statistical file

stat_dict.update({'data_dir':'/tmp/predmod_data/'})      # Local directories
stat_dict.update({'pred_model':'ADDFIT'})                # Select model type

stat_dict.update({'response_attr':'MARSUPWT'})   # Select response attribute
pred_attr = ['YEAR', 'CAPGAIN','INCOME','AAGE','ASEX']
stat_dict.update({'predictor_attr':pred_attr}) # Select predictor attributes
stat_dict.update({'grid_resolution':10})            # Select grid resolution

write_conf('CENSUS', stat_dict)                    # Create configuration file

                                                   # Start (parallel) MPI code
result = run_predmod('CENSUS', stat_dict, num_proc, solver_args, alpha)

                                                    # Create model description
model = plot_predmod(result, 'CENSUS', pred_attr, stat_dict, alpha)

model.plot()                                       # Plot model (see Figure 3)
```

Fig. 2. Python code for calling ADDFIT with CENSUS database.

3.1 Python Wrapper

We are using the scripting language Python [5], both for the development of prototypes, as well as controlling the parallel implementations, facilitating user interface and to display results graphically. Python is an excellent tool for rapid code development. It handles large amounts of data efficiently, it is very easy to write scripts as well as general functions, it can be run interactively (interpretable) and it is flexible with regards to data types because it is based on general lists and dictionaries (associative arrays), of which the latter are implemented as very efficient hash-tables. A numerical extension [9] provides functionalities similar to Matlab, and many more modules are available in all kinds of areas, including interfaces to Tkinter and Gnuplot.

Our Python wrapper code reads the data structures which describe a data set and a predictive model and all its parameters, and creates a temporary configuration file which is then read by the parallel application. This data structures use dictionaries and are in an easy readable form as can be seen in Figure 2. The Python code also dynamically creates an MPI call, which starts the parallel application. Results are written to files by the parallel program, loaded by a Python function and displayed using a graphical interface as shown in Figure 3.

We also use Python for testing purposes, as it is easy to run parallel codes with different arguments in batch mode, to compare results and create reports. The results from our Python prototype codes are compared to the results from the parallel C/MPI code (running on various number of processing nodes), thus giving us another tool for testing.

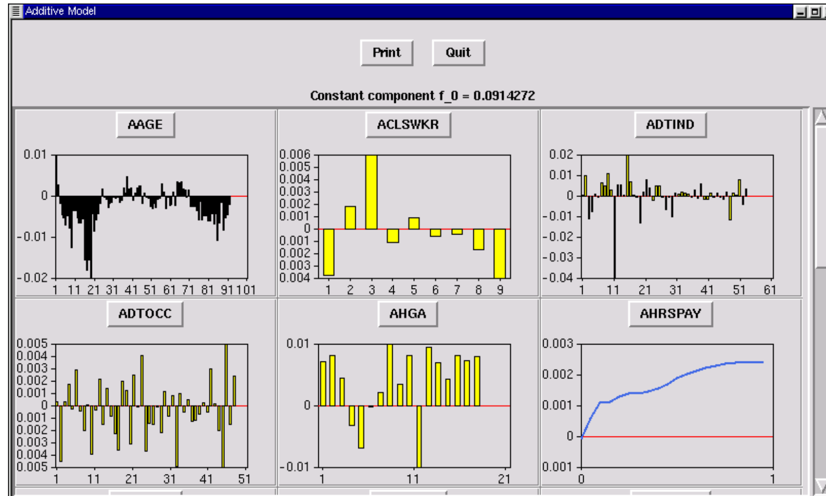

Fig. 3. Graphical output for ADDFIT.

3.2 Parallel Assembly

The parallel codes have been implemented in C using MPI [21] for communications. The basic structure of the assembly routine is the same for both ADDFIT and HISURF, and both can be solved with the dense linear system solver described in Section 3.3. The matrix data structures and the assembly of data records into the linear system are the only parts that differ in ADDFIT and HISURF.

A Single-Program-Multiple-Data (SPMD) style is used for both the assembly and solver steps. A configuration file (dynamically created by the Python wrapper code) is read by all processes from a commonly accessible directory (assuming a NFS for parts of the directory tree). In the assembly step, the outermost loop iterates over the available data files. For each given data file, the number of records n is divided by the number of processes p and each process reads its part, with p_0 reading from the beginning of the files and all other processes skipping to their corresponding positions.

In the innermost loop each process reads and assembles its n/p data records in a blocking fashion, i.e. a number of records is loaded, assembled into the local linear system, and the next block is loaded. Thus, it is possible to trade memory usage and I/O access (larger blocks are usually more efficient to load). In the assembly step a local linear system is assembled on each process as described in Section 2, and the sum of all theses local systems is then the final linear system to be solved.

If the assembled linear system is small, a sequential solver can be used to solve it. In this case, the local systems have to be reduced into the final linear system on one processing node before it can be solved sequentially. Alterna-

tively, if the assembled linear system is too large to be solved efficiently on one processing node (i.e. if a parallel solving is faster than the sequential), a block-cyclic redistribution is performed and the parallel solver discussed in the next subsection is used to solve the system.

3.3 Parallel Dense Solver

Solving the assembled linear system can be done with either a sequential or parallel solver, depending on the size of the system and the available parallel architecture.

The systems currently generated by HISURF and ADDFIT are dense and symmetric, positive definite in the former case, and semi-definite in the latter case. However, in future refinements of these models, the definiteness property may be lost, for example because of the addition of extra constraints or – in the case of additive models – extending it to a second-order model.

For HISURF and ADDFIT a solver is thus required that will be accurate for any symmetric dense system, and also has good parallel and sequential performance. The former requirement argues for a direct solver with good stability properties; the latter argues for one that exploits symmetry to require only $\frac{m^3}{3} + O(m^2)$ floating point operations, and that has been shown to have an efficient parallelisation. A direct solver for general symmetric (indefinite) systems based on the diagonal pivoting method [2, 14] meets these requirements.

In the diagonal pivoting method, the decomposition $\boldsymbol{A} = \boldsymbol{LDL}^T$ is performed, where $\boldsymbol{L}$ is an $m \times m$ lower triangular matrix with a unit diagonal, and $\boldsymbol{D}$ is a block diagonal matrix with either 1×1 or 2×2 sub-blocks [14]. The factorisation of $\boldsymbol{A}$ proceeds column by column; in the elimination of column j, three cases arise:

1. Eliminate using a 1×1 pivot from $A_{j,j}$. This corresponds to the definite case, and will be used when $A_{j,j}$ is sufficiently large (compared with $\max(A_{j+1:m,j})$).
2. Eliminate using a 1×1 pivot from $A_{i,i}$, where $i > j$. This corresponds to the semi-definite case; a symmetric interchange with row/columns i and j must be performed.
3. Eliminate using a 2×2 pivot using columns i' and i ($i', i \geq j, i' \neq i$). This case produces a 2×2 sub-block at column j of $\boldsymbol{D}$. This corresponds to the indefinite case; a symmetric interchange with rows/columns i', i and $j, j+1$ must be performed. However, columns j and $j+1$ are eliminated in this case.

The tests used to decide between these cases, and the searches used to select column i (and i'), yield several algorithms based on the method, the most well-known being the variants of the *Bunch-Kaufman* algorithm (see [14] and the references cited within).

It has been recently shown for the *Bunch-Kaufman* algorithm that there is no guarantee that the growth of $\boldsymbol{L}$ is bounded [2]. Variants such as the *bounded Bunch-Kaufman* and *fast Bunch-Parlett* algorithms have been devised which

overcomes this problem. The extra accuracy of these methods results from more extensive searching for stable pivot columns i (and i') for cases 2 and 3, with a correspondingly more frequent use of these cases.

For linear systems that are close to definite, such as are likely to be generated by our models, the diagonal pivoting methods permit most columns to be eliminated by case 1, requiring no symmetric interchanges. For a parallel implementation, this is a highly useful property, as even for large matrices the communication startup and volume overheads of symmetric interchange, when the rows and columns come from different nodes, is considerable [28].

Instead of suppressing interchanges, which even if done judiciously may result in the loss of some accuracy [28], high parallel performance can also be achieved with a *block-search* algorithm that searches for suitable pivot columns i and i' from the current storage block [22]. If this search was successful, the symmetric interchanges would require no communication, resulting in no parallel overhead. Such a strategy could be based on the *Duff-Reid* algorithm used for sparse matrices [2, 22], which also has strong guarantees of accuracy.

However, if the search was not successful, an equally stable means of eliminating column j must then be used. We chose the *bounded Bunch-Kaufman* algorithm over the *fast Parlett-Reid* algorithm, as the latter requires sorting of the columns by the size of the diagonal, which would give it higher parallel overheads. Further details can be found in [29].

The implementation of this parallel solver assumes the processing nodes being arranged logically in a rectangular grid. The parallelisation of the solver step is independent from the assembly step. We are using a partial reduction operation which reduces the assembled local linear systems from p processing nodes onto a logical $q_1 \times q_2$ grid. While it is often appropriate to use all available processing nodes to assemble the linear system (because a large amount of data has to be loaded from disk), solving the system on only a small number of nodes might be more efficient due to the increasing communication overhead. A special case is $q_1 = q_2 = 1$, i.e. a sequential solving of the system, in which case the local linear systems are reduced onto one processing node.

4 Cluster Architecture and Performance Results

For test and timing purposes we downloaded and installed the publicly available *Census-Income* database from the *UCI KDD Archive* at the *University of California, Irvine*[1]. This database contains weighted census data extracted from US population surveys conduced by the *US Bureau of Census* in 1994 and 1995. It contains 42 demographic and employment related attributes, six of them continuous and the others categorical with up to 92 categories. The data set contains a total of 299, 285 records.

In a preprocessing step statistical information was collected and stored in a configuration file. For continuous attributes the minimal and maximal values

[1] See: `http://kdd.ics.uci.edu/`

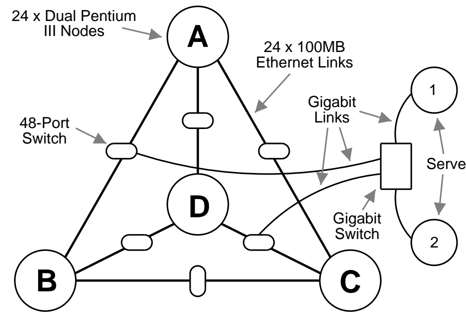

Fig. 4. Architecture of the ANU *Bunyip* Beowulf cluster.

are stored, while for categorical attributes the number of categories are stored and the category names are saved in separate files. A further normalisation step then converted the original text files into binary files, one per attribute. Continuous attributes were normalised into the interval $[0, 1]$ and for categorical attributes only the category numbers were stored. Using only these binary files for the predictive modelling process allows very efficient file reading using the C function `fread()` instead of the much slower `fscanf()`.

The implementations discussed and the results presented in this paper were all developed and measured on the Beowulf style Linux cluster called *Bunyip*[2] [1] at the *Australian National University, Canberra*. This cluster is built with 98 dual 550 MHz Pentium III nodes, each equipped with 384 Megabytes of RAM (total about 36 Gigabytes), 13 Gigabytes of disk space (total 1.3 Terabytes) and 3×100 MBit/s fast Ethernet cards. Logically 96 nodes are connected in four groups of 24 nodes arranged as a tetrahedron with a group of nodes at each vertex, as depicted in Figure 4. Two nodes are dedicated as servers connected to a network-switch by a Gigabit link

4.1 ADDFIT Performance Results

In this section we present timing results for ADDFIT using the *Census* database described above. For these tests we used up to 48 processing nodes (groups B and C) on the *Bunyip* cluster. The database (total size around 54 Megabytes) has been copied onto local discs, so file access was handled locally.

To simulate a larger data set, we also run tests where we loaded the *Census* database ten times, with a total of $2,992,850$ records. We denote the normal one-time loading of *Census* with $Census_1$ and the tenfold loading with $Census_{10}$. Note that loading the data ten times does not affect the size of the linear system.

In a first series of tests we choose a subset of 8 attributes (7 categorical and one continuous), and a small matrix of dimension 100×100 was assembled and solved. The amount of data which was loaded in this test was around 7

[2] See: `http://tux.anu.edu.au/Projects/Bunyip/`

Megabytes in total. The grid resolution for the continuous attribute was set to 33 grid points. Each test run was performed 10 times and we report average times. Test were run both with loading *Census* once and ten times ($Census_1$ and $Census_{10}$, respectively).

The second series of timings was run with all attributes used for the predictive modelling (i.e. one continuous attribute as response attribute and all others as predictor attributes). In this case, a total of around 54 Megabytes were loaded in the $Census_1$ case and around 540 Megabytes with the $Census_{10}$ case. The resulting linear system had a dimension of 1000×1000.

In Table 1 we show the timings achieved with the Python prototype code of ADDFIT on one processing node of Bunyip, and Tables 2 and 3 show timings for the C/MPI implementation. All presented times are in seconds.

	8 attributes		41 attributes	
	Census_1	Census_{10}	Census_1	Census_{10}
Loading	61	617	322	3209
Assembly	4.142	4.160	93.243	93.294
Solving	0.029	0.037	18.132	18.221

Table 1. ADDFIT Python prototype timings (in seconds).

For the Python prototype, the total run-time is dominated by loading the data from files. Once the data is loaded into memory, assembling the linear system is quite fast and solving the system is even faster (even for the 1000×1000 system).

For the parallel C/MPI code the time for the assembly step (which includes loading the data from files) is reduced linearly with increasing number of processing nodes used, while the time to reduce the final linear system increases with number of nodes. So there is a trade-off between reduced local I/O and communication. As expected the times for loading and assembling the $Census_{10}$ data takes ten times longer than for the $Census_1$ data.

Processing nodes	1	2	4	8	16	24
Assembly $Census_1$	3.921	1.975	0.984	0.493	0.243	0.160
Assembly $Census_{10}$	39.139	19.558	9.714	4.823	2.390	1.573
Reduce:	–	0.007	0.057	0.038	0.054	0.064
Solving:	0.009	0.009	0.010	0.010	0.010	0.010

Table 2. ADDFIT C/MPI timings with 8 attributes and a 100×100 system.

Processing nodes	1	2	4	8	16	24	32	40	48
Assembly $Census_1$	50.37	25.32	12.69	6.35	3.19	2.13	1.80	1.45	1.22
Assembly $Census_{10}$	498.00	250.06	124.94	62.42	31.01	20.60	17.33	13.92	11.54
Reduce:	–	0.76	1.51	2.18	2.82	3.50	3.45	4.12	4.38
Solving:	2.17	2.20	2.18	2.18	2.18	2.18	2.37	2.37	2.37

Table 3. ADDFIT C/MPI timings with 41 attributes and a 1000×1000 system.

Note that the time to solve the linear system is the same for different number of nodes used. This is because we solved the linear system sequentially in all presented tests. Even for the 1000×1000 system a parallel solving resulted in a speed-down due to the high communication overhead. But as the overall run time is dominated by the assembly step – specially if we look at data mining applications with millions of records – the solving step is only a fraction and can almost be neglected. However, it should be noted that systems of this size are small on even two processing nodes, and the solver performance scales well for larger matrices [29].

As the complete triangular matrix is stored on each processing node, the reduce step becomes a large portion of the total run-time if the number of processing nodes is increased. In the case of the 1000×1000 matrix, almost 4 Megabytes are communicated from one node to another. Even with a binary-tree like reduction algorithm, $log_2(p)$ messages are needed to reduce the linear system to one processing node.

5 Outlook

In this paper we presented two methods for parallel predictive modelling that can be used for data mining of large and complex data sets. Based on additive models and wavelets, ADDFIT and HISURF both require the assembly of a symmetric dense linear system, which can be be done in parallel. The solution step has parallelisation properties which are independent of the assembly.

At the time of this writing, the ADDFIT method is fully implemented in C/MPI, and we are currently working on the integration of HISURF into the same framework. A third method, called TPSFEM [7], will be added later.

The presented implementations were developed on a Beowulf cluster, however, we are planning to port our applications to the APAC [3] National Facilities as soon as they will be available.

The Python wrapper code currently used helps the user to set up and control a predictive modelling process, by facilitating the choice of attributes and model parameters, and by providing a graphical output. We plan to include our parallel predictive modelling applications into the framework of the *DMtools* [25], a data mining toolbox – written in the scripting language Python – that allows efficient

[3] Australian Partnership for Advanced Computing, `http://www.apac.edu.au`

and flexible access to relational databases and helps with handling of common tasks in data mining.

References

1. D. Aberdeen, J. Baxter and R. Edwards, *92 cents/MFlops/s, Ultra-Large-Scale Neural-Network Training on a PIII Cluster*, Proceedings of the High Performance Networking and Computing Conference (SC2000), ACM Press and IEEE Computer Society Press, November 2000.
2. C. Ashcraft, R.G. Grimes, and J.G. Lewis, *Accurate Symmetric Indefinite Linear Equation Solvers*, Simax, 20(2), 1998.
3. S. Bakin, M. Hegland and M. Osborne, *Can MARS be Improved with B-Splines?*, Computational Techniques and Applications: CTAC-97, World Scientific Publishing Co., 1998.
4. G. Bell and J.N. Gray, *The revolution yet to happen*, Beyond Calculation (P.J. Denning and R.M. Metcalfe, eds.), Springer Verlag, 1997.
5. D.M. Beazly, *Python Essential Reference*, New Riders, October 1999.
6. P. Christen, M. Hegland, O.M. Nielsen, S. Roberts, P.E. Strazdins and I. Altas, *Scalable Parallel Algorithms for Surface Fitting and Data Mining*, accepted by the Elsevier Journal of Parallel Computing, special issue on Aspects of Parallel Computing for Linear Systems and Associated Problems, September 2000.
7. P. Christen, M. Hegland, S. Roberts and I. Altas, *A Scalable Parallel FEM Surface Fitting Algorithm for Data Mining*, Technical Report TR-CS-01-01, ANU Joint Computer Science Technical Report Series, January 2001.
8. I. Daubechies, *Orthonormal bases of compactly supported wavelets*, Comm. Pure and Appl. Math., 1998.
9. P.F. Dubois, K. Hinsen and J. Hugunin, *Numerical Python*, Computers in Physics, Vol.10, No. 3, May 1996.
10. D. Düllmann, *Petabyte databases*. Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD-99), ACM Press, July 1999.
11. B. Efron and C. Stein, *The jackknife estimate of variance*, The Annals of Statistics, vol. 9, num. 3, pages 586-596, 1981.
12. A.A. Freitas and S.H. Lavington, *Mining Very Large Databases with Parallel Processing*, Kluwer Academic Publishers, 1998.
13. J.H. Friedman, *Multivariate adaptive regression splines*, The Annals of Statistics, vol. 19, 1991.
14. G. Golub and C. Van Loan, *Matrix Computations*, John Hopkins University Press, Baltimore, Second edition, 1989.
15. J. Han and M. Kamber, *Data Mining: Concepts and Techniques*, Morgan Kaufmann Publisher, August 2000.
16. T.J. Hastie and R.J. Tibshirani, *Generalized Additive Models*, Monographs Chapman and Hall, 1990.
17. S. Hawkins, G. Williams, R. Baxter, P. Christen, M. Fett, M. Hegland, F. Huang, O.M. Nielsen, T. Semenova and A. Smith, *Data Mining of Administrative Claims Data for Pathology Services*, Proceedings of the 34th Annual Hawaii International Conference on System Sciences (HICSS-34), January 2001.
18. M. Hegland, S. Roberts and I. Altas, *Finite Element Thin Plate Splines for Data Mining Applications*, in Mathematical Methods for Curves and Surfaces II, M. Daehlen, T. Lyche and L.L. Schumaker, Eds., Vanderbilt University Press, 1998.

19. M. Hegland and V. Pestov, *Additive Models in High Dimensions*, School of Mathematical and Computing Sciences, Victoria University of Wellington, 1999.
20. M. Hegland, O.M. Nielsen and Z. Shen, *High Dimensional Smoothing Based on Multilevel Analysis*, submitted to SIAM J. Scientific Computing, November 2000.
21. W. Gropp, E. Lusk and A. Skjellum,*Using MPI – Portable Parallel Programming with the Message-Passing Interface*. MIT Press, Cambridge, Massachusetts, 1994.
22. J.G. Lewis. Private communications, 1999.
23. W.A. Maniatty and M.J. Zaki, *A Requirements Analysis for Parallel KDD Systems*, IPDPS'2000 Data Mining Workshop, Cancun, Maxico, May 2000.
24. O.M. Nielsen, *Wavelets in Scientific Computing*, PhD thesis, Technical University of Denmark, 1998.
    Available at `www.bigfoot.com/~Ole.Nielsen/thesis.html`
25. O.M. Nielsen, P. Christen, M. Hegland and T. Semenova, *A Toolbox Approach to Flexible and Efficient Data Mining*, Accepted for the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'2001), Hong Kong, April 2001.
26. T. Sterling, D. Savarese, D.J. Becker, J.E. Dorband, U.A. Ranawake and C.V. Packer, *BEOWULF: A Parallel Workstation for Scientific Computation*, International Conference on Parallel Processing (ICPP-95), Vol.1: Architecture, CRC Press, August 1995.
27. G. Strang and T. Nguyen, *Wavelets and Filter Banks*, Wellesley-Cambridge Press, 1996.
28. P.E. Strazdins, *Accelerated methods for performing the LDLT decomposition*, Proceedings of the 1999 International Conference on Computational Techniques and Applications (CTAC-99), Australian National University, Canberra, September 1999, Appears in ANZIAM J. 42 (C), Adelaide, December 2000.
29. P.E. Strazdins and J.G. Lewis, *An Efficient and Stable Method for Parallel Factorization of Dense Symmetric Indefinite Matrices*, submitted to the HPC Asia Conference, Gold Coast, Australia, 24-28 September, 2001
30. G. Wahba, *Spline models for observational data*, CBMS-NSF Regional Conference Series in Applied Mathematics, vol. 59, SIAM, 1990.
31. M.J. Zaki and C-T. Ho, *Large-Scale Parallel Data Mining*, Lecture Notes in Computer Science and Lecture Notes in Artificial Intelligence, Vol. 1759, Springer-Verlag, 2000.