ELSEVIER Computational Statistics & Data Analysis 31 (1999) 457–474

COMPUTATIONAL STATISTICS & DATA ANALYSIS

www.elsevier.com/locate/csda

Parallel implementation of a multilevel modelling package

J.M. Bull[a,*], G.D. Riley[a], J. Rasbash[b], H. Goldstein[b]

[a] *Centre for Novel Computing, University of Manchester, Manchester, M13 9PL, UK*
[b] *Institute of Education, University of London, London, WC1H 0AL, UK*

Received 15 August 1998; received in revised form 13 March 1999; accepted 1 April 1999

Abstract

A portable parallel implementation of *MLn*, a multilevel modelling package, for shared memory parallel machines is described. Particular attention is paid to cross-classified and multiple membership models, which are more computationally demanding than those with simple hierarchical structure. Performance results are presented for a range of shared-memory parallel architectures, demonstrating a significant increase in the size of models which can be handled interactively.

Keywords: Multilevel models; Cross classification; Multiple membership; Parallel computing; Shared memory; Threads

1. Introduction

Multilevel models are also known as random effects models, variance components models, random coefficient models and hierarchical linear models. This last term is something of a misnomer since these techniques have expanded to include non-linear models and non-hierarchical population structures. There is currently an explosion of interest in these techniques amongst quantitative researchers in the social and biological sciences. Multilevel models extend regression and generalised linear models to allow multiple random effects to be included in the model. This allows these techniques to directly model the patterns of heterogeneity that are imposed by the

* Corresponding author.
E-mail addresses: mbull@cs.man.ac.uk (J.M. Bull), griley@cs.man.ac.uk (G.D. Riley), j.rasbash@ioe.ac.uk (J. Rasbash), h.goldstein@ioe.ac.uk (H. Goldstein)

0167-9473/99/$ - see front matter
PII: S 0167-9473(99)00042-0

complex population structures from which social and biological scientists draw data for modelling. There is a growing literature on these techniques: three of the more widely read texts are Goldstein (1995), Bryk and Raudenbush (1993) and Longford (1993).

MLn is a widely used software package that was designed specifically for the estimation and exploration of multilevel models. The purpose of this paper is to describe a parallel implementation of *MLn* for shared memory parallel computers. Section 2 provides some background on multilevel models, Section 3 describes the algorithms used in *MLn* and Section 4 describes how parallelism can be exploited in them. Section 5 covers the issues involved in the implementation, paying attention to choice of parallel programming paradigm, portability and parallelisation strategy. In Section 6 we present performance results on some benchmark data sets across a variety of shared memory parallel machines. These results are analysed in Section 7, and in Section 8 we draw conclusions and give pointers to future work.

2. Multilevel models and *MLn*

2.1. Hierarchical models

Initially, multilevel modelling focused on strictly hierarchical population structures. Some simple examples of these are: pupils within schools, individuals within households within areas, and repeated measurements within cows within herds within farms.

The simplest multilevel model corresponds to a one-way random effects analysis of variance,

$$y_{ij} = (\beta + u_j + e_{ij})x,$$

where $u_j \sim \mathrm{N}(0, \sigma_u^2)$ and $e_{ij} \sim \mathrm{N}(0, \sigma_e^2)$. This basic model has the following important extensions:

- Inclusion of any number of hierarchical levels.
- Inclusion of explanatory variables defined at any level.
- The coefficient of any explanatory variable may be random at any level.
- Handling of non-Normally distributed responses.

2.2. Cross-classified models

The assumption of purely hierarchical populations often does not hold true for many of the data sets collected by social and biological scientists. For example, pupils attending any school are drawn from many neighbourhoods and pupils from one neighbourhood attend more than one school. No pure hierarchy can be found and pupils are said to be contained within a cross-classification of schools by neighbourhoods. Other examples are: pupils within (primary $\times$ secondary school), patients

within (primary health care units × secondary health care units), and repeated measures within (individuals×raters).

The basic model can be written as

$$y_{i(j_1j_2)} = X\beta + u_{j_1} + u_{j_2} + e_{i(j_1j_2)},$$

where $u_{j_1} \sim \mathrm{N}(0, \sigma^2_{u_1})$, $u_{j_2} \sim \mathrm{N}(0, \sigma^2_{u_2})$ and $e_{i(j_1j_2)} \sim \mathrm{N}(0, \sigma^2_e)$. If we take an educational system example, $y_{i(j_1j_2)}$ is the response measurement for the ith pupil from the (secondary school j_1, primary school j_2) cell in the cross-classification, X is a matrix of explanatory variables with fixed coefficients β, u_{j_1}, u_{j_2} and $e_{i(j_1j_2)}$ are random effects at the secondary school, primary school and pupil levels, $\sigma^2_{u_1}$, $\sigma^2_{u_2}$, and σ^2_e are the between secondary school, between primary school and between pupil variances. The same extensions that apply to hierarchical models apply to cross-classified models. However, there is the additional extension to multi-way cross-classifications.

2.3. Multiple membership models

Where lower level units are members of more than one higher level unit we have a multiple membership model. The differences between multiple membership and cross-classification models are sometimes subtle. For example, suppose we measure pupils every three months for two years. At each occasion we have a response measurement for each pupil. Some children will move schools over the period. Now we have a population structure which is cross-classified: repeated measurements within (individuals × schools).

Each lowest level unit, measurement occasion, is a member of only one school and one individual. However, suppose we have a single response on each pupil, taken say at the end of the study, but we know the identities of the schools each pupil attended over the two years. Now the lowest level units are pupils and some pupils are members of multiple schools. Hence we have the multiple membership model.

The multiple membership model provides a framework for representing structures where individuals are simultaneously members of multiple social groups (such as peer friendship groups or extended families). These groupings are dynamic, they form, break down and reform in different configurations over time. The multiple membership model provides a powerful quantitative framework for modelling these social processes in studies which track such groupings over time. The multiple membership model is set out and an example analysis described in Hill and Goldstein (1998). This paper uses a notation for the multiple membership model which is not entirely consistent, although its intent is clear. A general notation for a two-level multiple membership model is

$$y_{i\{j\}} = (X\beta)_{i\{j\}} + \sum_{h\in\{j\}} u_h \pi_{ih} + e_{i\{j\}},$$

with $u_h \sim \mathrm{N}(0, \sigma^2_u)$ and $e_{i\{j\}} \sim \mathrm{N}(0, \sigma^2_e)$, where j is the full set of the m level 2 units. The level 1 units, for example pupils, are indexed uniquely by i and may be a member of more than one school. The weight π_{ih} represents the proportion of membership of pupil i in school h. Thus if pupil i spent a quarter of their time

in school h a weight of 0.25 would be reasonable. For the particular cases of a simple hierarchy, where every level 1 unit belongs to a single level 2 unit, or a two-way cross-classification where every level 1 unit belongs to a single level 2 unit from each classification then $y_{i\{j\}}$ can be replaced respectively by y_{ij}, $y_{i(j_1j_2)}$ without ambiguity.

In a single model we can have mixtures of hierarchical, cross-classified and multiple membership structures. For example, we may have pupils within classes within schools. We may also have measured peer-friendship group activity at the within class-room level: pupils will therefore be multiple members of peer-friendship groups. We may also have family information: families will be cross-classified with schools and individuals may be members of multiple families. We can estimate patterns of variability attributable to pupils, peer-groups, classes, schools and families. Furthermore, we can introduce explanatory variables associated with each of these units in an attempt to explain the estimated variability.

One major drawback with cross-classified and multiple membership models is that they are extremely computationally intensive both in terms of processor time and storage. This paper explores and explains the use of parallelism to reduce the elapsed run-time to bring a wider range of cross-classified and multiple membership models within a time scale where the software can be used interactively. Experience has shown that users find it acceptable to wait up to about 10–15 min to estimate models interactively. When waits go beyond this, users tend to move to a different pattern of work where using the software is mixed with other activities.

3. Statistical and computational algorithms

3.1. The statistical algorithm

In this section we describe in more detail the underlying statistical and computational structures and algorithms and describe why multiple membership and cross-classified models are more computationally intensive. This section also sets the context for subsequent sections on parallelisation strategy.

The general linear model with multiple sets of random effects can be written as

$$Y = XB + \sum_{r=1}^{R} Z_r u_r, \tag{1}$$

where X is a matrix of explanatory variables with coefficients B (the fixed parameters), R is the number of sets of random effects in the model and Z_r is a design matrix for u_r, the rth set of random effects. Z_r is of size $n \times n_r$, where n is the number rows in the entire data set, n_r is the size of the rth set of random effects and u_r is a vector of length n_r. We wish to estimate B and Ω, the $R \times R$ variance covariance matrix of the sets of random effects.

To clarify this, consider a simple two-level variance components model, of pupils within schools. Suppose we have 3000 pupils nested within 50 schools. Let Z_1 be the design matrix for the pupil-level set of random effects and Z_2 be the design matrix

for the school-level set of random effects. Z_1 is an identity matrix of order 3000, while Z_2 is dimensioned 3000×50, with Z_{2ij} having the value one if the ith pupil is in school j, and zero otherwise. In this example we considered nested random effects, though (1) allows for the more general case of non-nested random effects.

Where $\mathrm{cov}(Y \mid XB) = V$ is known, the usual generalised least-squares estimators, for the fixed effects, can be used:

$$\hat{B} = (X^{\mathrm{T}}V^{-1}X)^{-1}(X^{\mathrm{T}}V^{-1}Y) \quad \text{and} \quad \mathrm{cov}(\hat{B}) = (X^{\mathrm{T}}V^{-1}X)^{-1}. \tag{2}$$

Given B we can obtain estimates of the parameters of V, which are $B^{\star}$ (the vector of the upper triangular elements of Ω), from the estimator

$$\hat{B}^{\star} = (Z^{\star\mathrm{T}}(V^{\star})^{-1}Z^{\star})^{-1}Z^{\star\mathrm{T}}(V^{\star})^{-1}Y^{\star} \quad \text{and} \quad \mathrm{cov}(\hat{B}^{\star}) = (Z^{\star\mathrm{T}}(V^{\star})^{-1}Z^{\star}). \tag{3}$$

Here $Y^{\star}$ is the vector of the elements of $(Y-XB)(Y-XB)^{\mathrm{T}}$, and therefore has length n^2, $V^{\star}$ is the covariance matrix of $Y^{\star}$ and $Z^{\star}$ is the design matrix linking $Y^{\star}$ to V in the regression of $Y^{\star}$ on $Z^{\star}$. $Z^{\star}$ is of size $n^2 \times P$, where P is the number of random parameters to be estimated. In the case where all elements of Ω are to be estimated, $P = R(R+1)/2$. However, in almost all models it is only sensible to estimate a subset of all possible covariances, since there are strong substantive reasons to assume that many of the pairwise combinations of sets of random effects are independent. For example, it is sensible to assume that pupil-level and school-level sets of random effects are independent. The form of $(V^{\star})^{-1}$ is $V^{-1} \otimes V^{-1}$, and thus it is of size $n^2 \times n^2$. Columns of $Z^{\star}$ relate to the variance of a set of random effects, or the covariance between two sets of random effects. In the case of a variance the column has the form $\mathrm{vec}(Z_r Z_r^{\mathrm{T}})$ and in the case of a covariance, $\mathrm{vec}(Z_{r'} Z_r^{\mathrm{T}} + Z_r Z_{r'}^{\mathrm{T}})$.

When V and B are both unknown, which is the usual case, then the iterative generalised least-squares estimates are those which simultaneously satisfy both (2) and (3). In the case where the sets of random effects follow a multivariate Normal distribution these estimators are equivalent to maximum likelihood. For a discussion of multilevel models with non-normal random effects, see Goldstein (1991) and Goldstein and Rasbash (1996).

The estimation procedure commences from an initial estimate of V, typically $V = I$, which is then used to obtain estimates of $\hat{B}$. Improved estimates of $\hat{B}^{\star}$ (the parameters of V) are then obtained, and so on, alternating between (2) and (3) until convergence is achieved, see Goldstein (1986).

3.2. Computational issues

The evaluation of (3) dominates computation, though a simplification is available. Consider a typical element of the first part of (3), for the variance of the rth set of random effects

$$\mathrm{vec}(Z_r Z_r^{\mathrm{T}})^{\mathrm{T}}(V^{-1} \otimes V^{-1})\mathrm{vec}(Z_r Z_r^{\mathrm{T}}).$$

The number of operations required to evaluate this term is proportional to n^4. By a well-known result (see Searle, 1982), this is equal to

$$\text{trace}(Z_r Z_r^{\mathrm{T}} V^{-1} Z_r Z_r^{\mathrm{T}} V^{-1}). \tag{4}$$

The number of operations required to evaluate the rearranged form is proportional to n^3. If the random effects fall into an hierarchical structure, and random effects operating at different levels of this structure are assumed independent, then V and its inverse have a nested block diagonal structure. Suppose we have a three-level model (pupils:classes:schools, say). It immediately follows that we can rewrite (4) as

$$\sum_{k=1}^{K} \text{trace}(Z_{r(k)} Z_{r(k)}^{\mathrm{T}} V_{(k)}^{-1} Z_{r(k)} Z_{r(k)}^{\mathrm{T}} V_{(k)}^{-1}), \tag{5}$$

where k indexes schools. The number of operations is now proportional to $\sum_{k=1}^{K} n_{(k)}^3$. The difference between $\sum_{k=1}^{K} n_{(k)}^3$ and n^3 for most datasets will be several orders of magnitude. Furthermore, with hierarchical random effects we can partition $V_{(k)}$ and its inverse into a series of components, one for each level of the hierarchy. The component for each level can be represented as a list of matrices with total storage requirements for each list of $2n_{(k)}R_l$, where R_l is the number of sets of random effects operating at level l of the hierarchy. Similarly, the sparse block diagonal structures such as $Z_{r(k)} Z_{r(k)}^{\mathrm{T}}$ can be represented by a list of dense vectors with total storage requirement $n_{(k)}$ for variance terms or $2n_{(k)}$ for covariance terms. These techniques, described in detail in Goldstein and Rasbash (1993), reduce the number of operations to be proportional to nR^2.

Some models contain a sub-group of sets of random effects which fit into an hierarchical structure and other sets of random effects which have a cross-classified or multiple membership relationship. In this case the hierarchical sub-group can be handled using the techniques just described. For the non-hierarchical sets of effects some partitioning of V and its inverse are possible. This reduces computation to be proportional to nm^2, where m is the number of categories in the crossed or multiple membership classification. For example, if we have 10,000 students drawn from 500 primary schools attending 100 secondary schools, then we can sort the data into student within primary school and fit secondary schools as a non-nested crossed classification. Computation is proportional to $10,000 \times 100^2 = 10^8$, as opposed to $10,000^3 = 10^{12}$, which would result from ignoring the structure in V and naively using (4). The techniques for efficiently estimating multilevel models with non-hierarchical random components are described in detail in Rasbash and Goldstein (1994).

For illustration, we consider the above example (10,000 students, 500 primary schools and 100 secondary schools) and compare the cases where primary school is nested within secondary school and where primary and secondary school are crossed. We also assume that we have three sets of random effects, one for each level, and that the maximum number of pupils in a secondary school, $\max(n_{(k)}) = 800$. Storage requirements are proportional to $\max(n_{(k)})R = 800 \times 3 = 2400$ for the nested random effects model and to $nm = 10,000 \times 100 = 10^6$ for the non-nested random effects model.

The number of operations required is proportional to $nR^2 = 10,000 \times 3^2 = 90,000$ for the nested random effects model and to $nm^2 = 10,000 \times 100^2 = 10^8$ for the non-nested random effects model.

4. Parallelism in *MLn*

The multilevel algorithm comprises four layers. The first three layers are implemented in C++ classes. Each class represents an archetypal matrix structure that occurs in the algorithm, defines an efficient storage representation and matrix algebra and manipulation methods for the structure. The lowest layer, layer 1, handles both symmetric and rectangular dense matrix objects. Layer 2 defines lists of these dense matrix objects that correspond to matrix structures such as the components of V, described in Section 3.2. The third layer handles matrix structures such as V itself which can be represented as lists of layer 2 objects. The highest layer, layer 4, uses these three classes to implement the algorithm: with purely hierarchical random effects this involves iteration across each highest level unit, as in (5).

Parallelism is discernible at each layer and can be described as follows:

Layer 1: Matrix operations. Parts of result matrices can be computed independently.

Layer 2: Operations on lists of matrices. Entire matrices within the list can be processed independently.

Layer 3: Operations on lists of layer 2 objects. Each layer 2 object can be processed independently.

Layer 4: Highest level units in the hierarchy can be processed independently.

As we move from layer 1 up to layer 4 the parallelism involved becomes more coarse-grained, the storage overhead for implementing parallelism increases and the changes required to the source code become less well encapsulated. Since cross-classified models are dominated by the computation of terms which cannot exploit (5), virtually no parallelism exists at layer 4 for these models. At layers 2 and 3, there will be instances where the number of items in the list is small (sometimes there is only one), so exploiting parallelism in these layers can only be effective if implemented in combination with parallelism at lower layers.

If we wish to focus on hierarchical models, which intrinsically have a low storage requirement, it would be reasonable to attempt to utilise the parallelism in layers 3 and 4. However, with cross-classified models there is no parallelism in layer 4 and parallelism in layer 3 requires a large storage overhead. The sequential algorithm for estimating cross-classified models already has a large storage requirement (for example, in a 2-level cross-classified model with n level 1 data items and m cross-classified categories, the dominating storage requirement is for a number of temporary matrices of size $n \times m$). Parallelising layer 3 of the algorithm would require replication of this temporary storage on every processor, greatly increasing the likelihood of having to use virtual memory. Once this occurs the overhead of paging to and from disk is likely to result in performance deteriorating beyond any benefits gained from the parallel implementation.

The computation time for cross-classified models is dominated by a few types of operation on large dense matrices. For moderately sized cross-classified data sets ($n \sim 2000$, $m \sim 100$) more than 95% of the time is spent in just 7 types of matrix operation:

(1) $A := BC$,
(2) $A := A + B$,
(3) $t := \text{trace}(AB)$,
(4) $A := \text{diag}(v)B$ or $A := B\,\text{diag}(v)$,
(5) $A := SC$,
(6) $A := 0$,
(7) $A := B$,

where A, B and C are rectangular matrices, S is a symmetric matrix, v is vector and t is a scalar. As the problem size increases, the first of these operations (rectangular matrix product, which has complexity $O(nm^2)$) increasingly dominates the execution time.

Thus it is reasonable to expect that layer 1 parallelism can be applied effectively to these models. Exploiting the parallelism in this layer has the advantages that there is no additional storage overhead, and the changes required to the source code are well encapsulated within a C++ matrix class library. We also explore the use of layer 2 parallelism, which has a small additional storage overhead, but also requires relatively minor changes to existing code.

5. Implementation

5.1. Threads versus message passing

The first decision which has to be taken before embarking on the parallelisation of *MLn* is to choose which parallel programming paradigm to employ. Given that the existing package is written in C++, and that we wish the parallel version to be robust and portable, we are effectively restricted to choosing between a distributed memory, message passing paradigm (as supported by PVM (Geist et al., 1994) or MPI (Message Passing Interface Forum, 1994)) and a shared memory, multiple threads paradigm (as supported by, for example, POSIX threads (International Organization for Standardization (ISO), 1996)). There are a number of (possibly conflicting) objectives which the choice of paradigm should satisfy. As well as portability, we should consider ease of programming, efficiency of the resulting code, and compatibility with the hardware typically available to the user base. We now consider each of these in turn.

Considering portability, message passing has the advantage that both PVM and MPI are freely available for a wide variety of systems. In contrast, POSIX threads are not nearly so widely available. Some operating systems on shared memory parallel systems (for example IRIX and Windows NT) provide only a native threads interface. Threads libraries, however, are normally part of the standard operating system release, whereas message passing libraries require separate installation. It is worth noting

that the recently announced OpenMP specification includes support for C and C++, which if widely adopted would solve the portability issue for shared memory systems.

It is widely accepted that threads libraries provide a simpler programming paradigm than message passing, largely because they support a global address space across all processors. This is especially significant when porting well-established sequential software, as opposed to developing parallel code from scratch. Use of threads will typically result in fewer changes to the existing code (therefore offering better maintainability) and shorter development time than is the case for message passing.

Given that we intend to exploit parallelism in the lower layers described in Section 4, the parallelism will be fine grained, and require significant communication between processors. Furthermore, the data objects in *MLn* (dense matrices) have a short lifetime, compared, say, to the main data objects in scientific programs, and are embedded in complex structures representing the sparse, nested block diagonal matrices described in Section 3.2. There is therefore no obvious data decomposition strategy on which to base a message passing implementation. Furthermore, due to the fine-grained nature of the parallelism and the short lifetime of data objects, we might expect a message passing implementation to require large numbers of small messages to be sent between processors. Since each message sent incurs a startup overhead, this is unlikely to be an efficient solution. A threads implementation, running on shared memory hardware, would not suffer the same overhead, as memory access costs are independent of the number of data items accessed.

The majority of *MLn* users have PCs, with the bulk of the remainder using UNIX workstations. For a message passing implementation to have any chance of being efficient on a distributed memory system, a fast, dedicated network as found in large MPP systems would be required: a typical Ethernet-based network of workstations or PCs with message startup costs of the order of 100 μs (see, for example Warren et al., 1997) is not likely to prove adequate.

To quantify this issue, consider running matrix–matrix multiplication, the dominant operation in *MLn*, on a network of eight workstations each delivering 250 Mflop/s, with a message startup time of 100 μs. The number of messages required depends on the data distribution strategy of operand and result matrices, but with the reasonable estimate that each processor would need to send four messages, a simple calculation shows that the number of floating point operations must be at least 8×10^5 for parallelisation to be worthwhile. For the typical problem sizes we are interested in, however, *MLn* spends significant time in matrix products with fewer than this number of operations. As will be shown in Section 6, matrix multiplications with up to two orders of magnitude fewer operations can be successfully parallelised on shared memory architectures.

A significant number of users have access to multiprocessor PCs, multiprocessor workstations, or shared memory UNIX servers, whereas very few have access to large MPP machines. Although message passing libraries can be efficiently implemented on shared memory hardware, the overhead of message startup costs will still be present.

Apart from the portability issue, the above arguments favour the use of threads over message passing. The portability issue can be overcome by utilising the native

threads libraries for each operating system and tightly encapsulating the differences between them so that they are invisible to nearly all of the application.

5.2. A portable threads interface

To overcome the portability issues described in Section 5.1, all parallel functionality is encapsulated in a small C++ class library. The library is designed to contain only the basic features required for parallel programming, which is a small subset of the functionality available in the underlying threads libraries. The library contains just three classes:

(1) *Thread Manager class*: The main feature of this class is a `doParallel` method, which takes a function and a list of arguments and executes the function with those arguments on all the threads. The standard facility `varargs` is used to pass the arguments. The class also contains methods to create the threads, and to return the current number of threads and the identity number of the calling thread.

(2) *Barrier class*: This class contains a method for synchronising all threads. It is used by the Thread Manager class to synchronise the threads at the beginning and end of each function executed using the `doParallel` method. It is also available to be called directly by the user, though this is not required in *MLn*. Since barrier synchronisation is often a significant source of overheads in shared memory programs, we have implemented an efficient barrier algorithm (the static F-way tournament of Grunwald and Vajracharya (1993)) as well as a simple centralised counter algorithm. The static F-way tournament algorithm is lock-free, and scales as the logarithm of the number of threads. Even for small numbers of threads, we have found it to be faster than a simple centralised counter scheme.

(3) *Lock class*: This class permits mutual exclusion between threads, using `lock` and `unlock` methods. These can be used to synchronise the updating of shared variables.

We use conditional compilation to provide versions for POSIX, Solaris, IRIX and Windows NT threads. The class library is compact, consisting of less than 700 lines of code in total.

5.3. Parallelisation strategy

We chose to perform sequential optimisation on the existing code before parallelisation. This ordering is not always sensible, since sequential optimisation may prevent subsequent parallelisation. However, the types of optimisation applied (eliminating indexing variables, ensuring stride-one memory access, loop unrolling and loop blocking) have no such impact. Applying them before parallelisation prevented us from expending effort parallelising code which, once optimised, did not account for a significant proportion of the execution time. This was important in the case of *MLn*, which contains some 50,000 lines of code in 660 methods, with many little-used branches, but would be unnecessary in simpler codes.

While parallelisation of dense matrix algebra can be seen as straightforward, we must take into account the wide variety of shapes and sizes of the matrices which

occur in *MLn*. For sufficiently small matrices, parallelisation is not worthwhile, as any benefits are outweighed by the overheads incurred, which are principally due to synchronising the processors at the beginning and end of each operation. For each matrix operation we compute the number of basic operations (either floating point operations or assignments) required. Only if this exceeds a given threshold is parallel execution enabled for that operation. The threshold is left as a tunable parameter, as the cost of synchronisation versus computation may vary significantly between platforms, and should be selected whenever the code is ported to a new platform.

For those operations which are large enough to be parallelised, we partition over either the rows or columns of the result matrix. If there are more rows than columns, we partition over rows, and vice versa, thus minimising any load imbalance which might occur if we partition over a small number of rows or columns. The sole exception to this is the operation $t := \mathrm{trace}(AB)$, where we partition over either rows or columns of A whichever is the larger number, and each processor accumulates a partial sum of t. These are then reduced to a global sum.

Parallelism at layer 2 (over lists of matrices) is exploited by dividing iterations of the loop over the list elements between processors. The matrices in the list may be of different sizes, so a simple partitioning may result in load imbalance. To overcome this, we use a dynamic scheduling algorithm. Since minimising synchronisation overheads is important and load imbalance not usually severe, we use the trapezoid self-scheduling algorithm (Tzen and Ni, 1993), which is designed for loops with these properties. Each of the p processors is initially assigned one pth of the iterations of the loop. Each processor's set of iterations is divided into approximately four chunks of decreasing size. The chunks are executed largest first, and if a processor runs out of work, it steals a chunk from the processor with the most remaining chunks.

Again, if the number of matrices in the list is too small, say less than four times the number of processors, then we exploit parallelism at layer 1 instead. In layer 2 we only exploit parallelism in one operation, which consists of computing the product $A = BC$, where B is a block diagonal matrix, and the list of matrices consists of the (dense) blocks.

A minor change to the algorithm used by *MLn* was made to increase the granularity of one of the matrix algebra operations: a set of vector cross-products was subsumed into a single matrix-crossproduct operation. This actually had a small beneficial impact on the sequential execution time.

5.4. Numerical libraries

Since many vendors supply a library of highly tuned and, in some cases, parallelised matrix algebra routines in the form of the Basic Linear Algebra Subprograms (BLAS) (Lawson et al., 1979; Dongarra et al., 1988, 1990), we considered the possible use of this library in the parallel version of *MLn*. Of the seven most important matrix operations, only $A = BC$ can be implemented by a single BLAS call. All the others must be implemented as a loop over multiple BLAS calls, and to avoid unnecessary synchronisation, parallelism must be exploited over the multiple calls, rather than within each call.

Unfortunately, although the library's (Fortran) interface is standard, its behaviour with respect to parallelisation is not, and mixing parallelism in both user code and library code tends to have undesirable results. Typically, a parallel version of the library creates its own set of threads, and it is not possible for the same set of threads to execute both user code and library code. Since, for example, the threads created by our portable threads interface busy wait (for efficiency reasons) between calls to `doParallel`, this will waste parallel resources by having more threads than processors.

Experiments on a Silicon Graphics Challenge showed no performance gain in *MLn* from implementing $A = BC$ using the BLAS library. On a twin processor, 200 MHz PentiumPro PC, our optimised matrix–matrix multiply delivered 240 MFlop/s on large matrices, compared to 294 MFlop/s for the BLAS implementation by Intel Corporation for Windows NT. Thus we concluded that the potential benefits of using native BLAS library calls did not outweigh the potential disadvantages. We should emphasise, however, that this decision was specific to *MLn*, and certainly should not be generalised to other code where use of vendor-supplied BLAS libraries may well provide an effective solution.

5.5. Experiences and lessons learnt

The work described above was estimated to have required approximately four man-months of programming effort. This was roughly equally divided between (i) sequential optimisation, (ii) developing and testing the threads library and (iii) adding parallelism to *MLn*. The majority of code changes (about 2000 new or modified lines) occurred in phases (i) and (ii): in phase (iii) debugging and tuning took most of the time.

Profiling of the sequential code was found to be crucial in determining which parts of the code effort should be expended on. However, profiling by itself did not tell the whole story, since the total time spent in the linear algebra methods was composed of many calls with a wide range of shapes and sizes of matrices. We found histogramming execution time against numbers of floating point operations to be a useful tool, to discover whether the time was being spent mainly in large numbers of short calls, or in small numbers of long ones. This was achieved by adding calls to a high-resolution system timer in the appropriate parts of the code.

Profiling of the parallel code also gave us much useful information about sources of overhead, such as the time spent in barrier synchronisation, and in unparallelised sections of the code. It also meant that we could compute speedups for individual methods, helping us to locate potential performance problems. Timer calls were also required to detect overhead due to load imbalance.

The use of C++ caused us few difficulties. The performance comparison with BLAS libraries described above suggests that performance comparable to Fortran can be achieved on linear algebra kernels, provided that suitable optimisations are applied. When calling a method in parallel, it is necessary to trace all methods called from within that method, to ensure that it is safe to do so. We found that use of operator overloading and virtual functions does complicate this process somewhat.

We offer the following advice to others contemplating a similar exercise:

- Do not consider parallelisation until you are certain that sequential optimisation and/or use of existing libraries cannot solve your performance problems (we observed up to an order of magnitude increase in performance from such optimisation).
- If functionality is still being added to the code, be aware that parallelisation will complicate revision control procedures.
- When choosing the parallel paradigm, carefully weigh up the issues of availability, portability, performance and programming effort.
- Be familiar with tools such as profiling, timers and parallel debuggers that might assist in locating and identifying problems in the parallel code.
- Remember that package users are unlikely to be familiar with parallelism: its presence should be as unobtrusive as possible.
- Be aware of all the possible causes of overhead in parallel programs. For example, barrier synchronisation can be a very significant source of overhead on shared memory architectures.

6. Results

To test the efficiency of the optimised and parallelised code, we have used a family of two-level cross-classified data sets. These data sets were produced by simulation, but are representative of typical real-world problems. For clarity, we can describe these data sets as patients being cross-classified by the clinic they attended and neighbourhood they live in. The dimensions of the data sets are as follows:

xc100: 2000 patients, 200 clinics, 100 neighbourhoods.

xc200: 4000 patients, 200 clinics, 200 neighbourhoods.

xc400: 10,000 patients, 200 clinics, 400 neighbourhoods.

For the reasons described in Section 3.2, the data is always sorted as patient within clinic, so that the clinic classification is absorbed into a hierarchical structure. Where we have a cross-classification of 400 neighbourhoods, this is computationally equivalent to (i) a three-way cross-classification of clinics with two other classifications, each with 200 units, for example, (clinics $\times$ neighbourhoods $\times$ doctors), or (ii) a model where patients are multiple members of neighbourhoods, neighbourhood is cross-classified with clinic, and there are 400 neighbourhoods. In general, if we have r classifications, the lowest level units, in this case patients, can be single or multiple members of the higher level classifications. If m_i is the number of units in the ith classification, then the computational load is defined by the number of patients and $\sum_{i=1}^{r} m_i$. Thus, although the data sets simulated and tested have a relatively simple structure, they accurately calibrate the computational performance of the adapted algorithm for a wide range of more complex models.

In each case we ran the iterative generalised least-squares procedure to convergence. This requires four iterations for xc100 and three iterations for xc200 and xc400. Note that the first iteration is different from subsequent iterations because

Table 1
Specifications of test platforms

Platform	CPU	L1 cache	L2 cache	OS	Compiler
Dell Work station 400	300 MHz PentiumPro	16 kb	512 kb	NT 4.0	Microsoft VC++ 5.0
SGI Challenge	100 MHz MIPS R4400	16 kb	1 Mb	IRIX 5.3	IRIX CC 4.0
Cray CS6400	66 MHz Sparc	16 kb	2 Mb	Solaris 2.5	SunSoft C++ 4.1
Sun Enter-prise 10000	250 MHz Ultra Sparc	16 kb	4 Mb	Solaris 2.5.1	Sun WorkShop C++ 4.2
SGI Origin 2000	195 MHz MIPS R10000	32 kb	4 Mb	IRIX 6.4	MIPSpro CC 7.20

Table 2
Parameter values

Platform	Threshold	Block size
Dell Workstation 400	20,000	200
SGI Challenge	1000	32
Cray CS6400	1000	512
Sun Enterprise 10000	20,000	512
SGI Origin 2000	2000	512

an identity matrix is used as the initial estimate for the weights matrix required to calculate the random parameters. This significantly simplifies the computation in the first iteration, which therefore requires less time to execute than subsequent iterations. Table 1 shows the specification of the various platforms on which we have run our benchmarks, including the CPU architecture, L1 and L2 data cache sizes and operating system and compiler versions. On each machine the highest available compiler optimisation level was used, but no other special options were invoked. The benchmarks were run in an environment where we had exclusive use of the number of processors required, but only on the PC did we have exclusive use of the whole machine.

Table 2 shows the setting of two variable parameters, the parallelisation threshold and block size for matrix–matrix multiplication, which we used on each machine. These values were determined by experiment — the best block size is essentially independent of data set or number of processors. The best parallelisation threshold does depend weakly on the number of processors, but performance is not sensitive to small changes in its value.

We have run performance tests on four versions of the code: Version 0 is the original sequential code; Version 1 is the optimised code, using layer 1 parallelism with a centralised barrier algorithm; Version 2 is as Version 1 but additionally exploiting layer 2 parallelism and Version 3 is as Version 2 but with the static F-way tournament barrier algorithm. The reason for including Version 2 as well as Version 3 was to be able to quantify the benefits of fast barrier synchronisation. Due to time

Table 3
Execution times on dual processor PC (left) and SGI Challenge (right)

	Version 0	Version 1	
p	1	1	2
xc100	63	34	25
xc200	545	241	148

	Version 0	Version 1			Version 2		
p	1	1	2	4	1	2	4
xc100	522	138	88	60	134	81	48
xc200	2895	676	402	260	648	364	217

Table 4
Execution times on Cray CS6400 (left) and Sun Enterprise 10000 (right)

	Version 0	Version 1			
p	1	1	2	4	6
xc100	436	184	115	63	72
xc200	2158	807	447	277	261

	Version 0	Version 1			
p	1	1	2	4	8
xc100	161	36	26	19.5	17.0
xc200	1185	285	164	104	75

Table 5
Execution times on SGI Origin 2000

	Version 0	Version 1				Version 2				Version 3			
p	1	1	2	4	8	1	2	4	8	1	2	4	8
xc200	504	94	65	53	61	94	65	53	61	94	59	46	43
xc400	9836	801	421	259	176	756	450	263	183	755	407	240	158

and machine availability constraints we have not run all versions of the code on all test data sets on every platform. However, the results presented are sufficient to demonstrate the important performance characteristics of the different versions of the code. Tables 3–5 show the total execution time in seconds of various versions of the code on the different platforms.

7. Discussion

The first observation is that both sequential optimisation and parallelisation result in substantial performance benefits on all the hardware platforms. The performance gains from both sources are more significant for the larger data sets. For the xc100 data set, some or all of the temporary matrices may fit into the L2 cache, and therefore the benefits of blocking are not as significant as for larger data sets. The benefits of sequential optimisation vary quite widely between different machines. For example on the xc200 data set, we observe a factor of 2.3 decrease in execution time on the PC, and a factor of 5.4 on the Origin 2000. These differences are due to

the differences in CPU architecture and in the quality of the compiler optimisations across the different platforms. Apart from choice of block size, the sequential optimisations are generic and not targeted at any particular system. It is likely that further performance gains could be attained by tuning the code specifically for each platform.

The larger the data set, the more computation occurs in tasks which are sufficiently large to make parallelisation worthwhile. In particular, multiplication of large matrices increasingly dominates the execution time as data-set size increases. Thus the relative cost both of synchronisation and of unparallelised code diminishes as the data-set size increases. However, even for the largest data set these overheads are still significant, and we do not observe linear speedup on any hardware platform. On several of the platforms, we come close to exhausting the potential for performance gain from parallelism: for data set xc100 on the Cray CS6400, the code is actually slower on six processors than on four. To make the code scalable to larger numbers of processors, we would need to exploit more parallelism from layers 2 and 3.

Adding parallelism in layer 2 has the largest benefit for small data sets. For example, the execution time for data set xc100 on four processors of the SGI Challenge, is reduced from 60 to 48 s. The larger the data set is, however, the larger the individual tasks to be performed on each matrix in the list are. Thus for these tasks, parallelism at layer 1 can be exploited effectively: moving up to layer 2 has less impact.

Finally, we note that efficient barrier synchronisation has a substantial impact on performance. Comparing Versions 2 and 3 on the Origin 2000 shows that there is a benefit from two processors upwards, and for data set xc200 on eight processors, a 30% reduction in execution time is obtained. For the larger xc400 data set the improvements are smaller (because the parallel tasks are larger) but still significant.

Overall, we see that sequential and parallel optimisations combine to give performance gains of up to 60-fold. These gains can make a significant difference to the way in which users are able to work with this package, as the size of models which can be manipulated in an interactive mode has been greatly increased.

8. Conclusions and further work

A parallel implementation of *MLn* for shared memory architectures has been presented. The choice of the shared memory paradigm has been carefully justified in terms of the typical characteristics of the computation. Through the creation of a portable threads library the issue of portability between parallel machines has been addressed. Results for benchmark data sets across a selection of parallel architectures have been presented. Useful performance gains are observed across a range of data-set sizes.

There are a number of possibilities for future work in this area. Exploiting parallelism at layer 4 would be beneficial for modelling very large data sets with a simple

hierarchy, and also for multivariate response models. New simulation-based estimation techniques implemented in *MLwiN*, the Windows version of *MLn*, notably bootstrapping and Markov chain Monte Carlo methods, are very computationally intensive. These techniques exhibit a coarse-grained parallelism and, unlike the algorithms discussed in this paper, could be readily distributed across networks of workstations as well as across the processors of shared memory parallel machines.

Acknowledgements

The authors are grateful to the Joint Information Systems Committee for funding this work under their Technology Applications Programme (JTAP). They would also like to thank John Brooke of Manchester Computing, David Clayton of the Medical Research Council Biostatistics Unit and Alastair Leyland of the University of Glasgow for their assistance in running the benchmark tests.

References

Bryk, A.S., Raudenbush, S.W. 1993. Hierarchical Linear Models. Sage, Newbury Park, California.

J.J. Dongarra, J. Du Croz, S. Hammarling, I.S. Duff, A set of level 3 basic linear algebra subprograms., ACM Trans. Math. Software 16 (1) (1990) 1–17.

J.J. Dongarra, J. Du Croz, S. Hammarling, R. Hanson, An extended set of FORTRAN basic linear algebra subprograms., ACM Trans. Math. Software 14 (1) (1988) 1–17.

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R., Sunderam, V., 1994. PVM 3 Users Guide and Reference Manual. Oak Ridge National Laboratory, Oak Ridge, Tennessee.

H. Goldstein, Multilevel mixed linear model analysis using iterative generalised least squares., Biometrika 73 (1) (1986) 43–56.

H. Goldstein, Nonlinear multilevel models, with an application to discrete response data., Biometrika 78 (1) (1991) 45–51.

Goldstein, H., 1995. Multilevel Statistical Models. Edward Arnold, London and Wiley, New York.

H. Goldstein, J. Rasbash, Efficient computational procedures for estimating parameters in multilevel models based on iterative generalised least squares., Comput. Statist. Data Anal. 13 (1993) 63–71.

H. Goldstein, J. Rasbash, Improved approximations for multilevel models with binary response., J. Roy. Statist. Soc. A 163 (1996) 505–513.

Grunwald, D., Vajracharya, S., 1993. Efficient barriers for distributed shared memory computers. Tech. Rep. CU-CS-703-94-93, Department of Computer Science, University of Colorado, Boulder, CO.

P.W. Hill, H. Goldstein, Multilevel modelling of educational data with cross-classification and missing identification of units., J. Ed. Behav. Statist. 23 (2) (1998) 117–128.

International Organization for Standardization (ISO), 1996. Portable operating system interface (POSIX) — Part 1: system application program interface. ISO/IEC Standard 9945-1.

C.L. Lawson, R.J. Hanson, D.R. Kincaid, F.T. Krogh, Basic linear algebra subprograms for FORTRAN usage., ACM Trans. Math. Software 5 (3) (1979) 308–323.

Longford, N.T., 1993. Random Coefficient Models. Clarendon Press, Oxford.

Message Passing Interface Forum, 1994. MPI: a message-passing interface standard. Internat. J. Supercomput. Appl. High Performance Comput. 8 (3,4).

J. Rasbash, H. Goldstein, Efficient analysis of mixed hierarchical and cross-classified random structures using a multilevel model., J. Ed. Behav. Statist. 13 (4) (1994) 337–350.

Searle, S.R., 1982. Matrix Algebra Useful for Statistics. Wiley, New York, pp. 332–333.

T.H. Tzen, L.M. Ni, Trapezoid self-scheduling scheme for parallel computers., IEEE Trans. Parallel Distributed Systems 4 (1) (1993) 87–98.

Warren, M.S., Becker, D.J., Goda, M.P., Salmon, J.K., Sterling, T., 1997. Parallel supercomputing with commodity components. In: Arabnia, H.R. (Ed.), Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'97). CSREA Press, pp. 1372–1381.