Parallel benefits in test-day evaluations

I. Strandén
Agricultural Research Centre - MTT,
Animal Production Research, 31600 Jokioinen, Finland

Abstract

One of the goals in the HPBREEDING project was to reduce computing time using parallel computing. Parallelization focused on the following aspects: 1) each of the processors should have about an equal amount of work, 2) communication between the processors should be kept small, 3) memory requirements per processor should be as small as possible.

The parallel implementation was tested with a small and a large random regression test-day model having about 7.28 and 21.7 million unknowns in the mixed model equations, respectively. The iterative solver method was iterated 50 rounds on Cycle Ultra AXmp and SGI Origin 2000 computers. Testing with the smaller (larger) model on the Cycle computer yielded speedups 1.97 and 3.86 (1.98 and 3.60) with 2 and 4 processors, respectively. Speedups for the iterating the smaller system on the SGI Origin 2000 were 1.65 and 3.06 with 2 and 4 processors, respectively. Increasing number of processors increased speedup almost linearly up to 8 processors, after which speedup increased slower than linearly due to increased communication.

1. Introduction

Random regression test-day models have become an active research field in dairy cattle breeding. Full benefits of this model can be realized in continuous evaluation. Hence, fast computing time is of great interest. Continuous evaluation using random regression test-day model for the Finnish dairy cattle industry requires heavy computing. Number of unknowns in the mixed model equations (MME) will increase from about 3.5 million in the current evaluation model to 20-60 million depending on the model adopted. It was estimated that the DMUIOD software (Lidauer et al., 1998a) designed to do this task would have required almost a month to solve the three production traits of interest using test-day model.

Parallel computing has been advocated as a cost effective way to increase computing capacity. Ideally, the computation time is reduced by the same factor as the number of engaged processors. Unfortunately existing parallel solvers cannot be used because the coefficient matrix cannot be stored in the core memory and iteration on data technique needs to be used (Schaeffer and Kennedy, 1986). In addition, the structure of the coefficient matrix (e.g. relationship matrix) makes parallelization difficult.

Madsen and Larsen (1998, 1999), Lidauer and Strandén (1998b), Strandén and Lidauer (1999) were the first to report experiences with a parallel implementation of an iteration on data breeding value estimation program in animal breeding. The latest results of work by Madsen and Larsen are described during this workshop. Their implementation used Gauss-Seidel and second-order Jacobi in the solving algorithm. It was designed for a distributed memory architecture and used parallel virtual machine (PVM) library. Objective of this paper is to describe the latest developments in parallel computing strategies in the Finnish HPBREEDING project.

2. Material and methods

Parallel computing cannot be considered as a separate tool in decreasing computing time. It is well known that slow programs tend to show high benefits from parallel computing. So, first, we questioned the solver algorithm in DMUIOD which used Gauss-Seidel and second order Jacobi algorithms. We decided to go to preconditioned conjugate gradient (PCG) method because it would be easier to parallelize. We also had some expectation on it to be superior to the algorithms in DMUIOD especially when a good preconditioner is used. Martin Lidauer explained results from our choice during this workshop.

The second important decision was finding a suitable parallel computing library (or environment). We decided to use message passing interface (MPI) as our parallel library because it is a standard, so it is likely to run on several computers without changes. In addition, it is publicly available (http://www-unix.mcs.anl.gov/mpi/).

In the following, we will describe shortly basics on parallelization, the iterative algorithm, the parallelization approach, our data set, and measures of parallelization.

Basic parallel concepts

In parallel computing the work is divided to be done by many processors. In practice, it is not possible to make all of the work parallel and there is some penalty for making the program parallel. A simple formula for the execution time of the parallel code with N_p processors is

$$T_{N_p} = T_{serial} + T_{parallel} / N_p + C(N_p) \qquad [1]$$

where T_{serial} = time of the serial part of code, $T_{parallel}$= time of the parallel part of code, $C(N_p)$. = communiacation and other work due to paralellization. Performance of the parallel code with increased number of processors is often described by speedup which is ratio of execution time using one processors code divided by parallel code, i.e., $S_{N_p} = T_1 / T_{N_p}$. Ideal speedup is when $S_{N_p} = N_p$.

According to formula [1] making the one processors program faster, i.e., having $T_{serial} + T_{parallel}$ smaller, always benefits parallelization as well as long as the serial time proportion $T_{serial} / (T_{serial} + T_{parallel})$ is not increased too much. So, optimization of performance of a program is benefitial.

A side effect of optimization is that the quicker the one processor program the lower the speedup may be because optimization does not affect time due to communication or extra work due to parallel code. Hence, poorly optimized program may show better speedup. However, overall performance of the code is poorer.

If we compare speedup results of the same program in different parallel computers then different computers may give different results because of work on communication and other work due to parallel code may have a different weight.

Parallelization

The PCG method (e.g. Shewchuk, 1994) was used for two reasons. First, it is easily applied to parallel processing. Second, it is superior over previously used solving algorithm when applied to dairy cattle breeding for Finnish data (Martin Lidauer during this workshop).

The parallel program code was developed from the optimized single processor program. The PCG algorithm was parallelized using MPI libraries. In practice, there are several strategies for parallelization of the PCG algorithm. Our experience lead to the implementation in Figure 1. Each processor performed all calculations for its part of the data set. Hence, each processor had its own part of the data and preconditioner files. Effective parallelism can be achieved when the processors have equal amount of work and communication between the processors is minimized. Consequently, development work concentrated on the following three aspects:

```
INITIALIZE  $k \Leftarrow 0$                                            MPI operations:
            $\mathbf{r}^{(0)} \Leftarrow \mathbf{b} - \mathbf{C}\mathbf{x}^{(0)}$;                       global sum vector r
            $\mathbf{d}^{(0)} \Leftarrow \mathbf{M}^{-1}\mathbf{r}^{(0)}$;                       send d
            $e_{new} \Leftarrow \mathbf{r}'^{(0)}\mathbf{d}^{(0)}$;                         global sum scalar $e_{new}$
DO UNTIL CONVERGENCE
            $\mathbf{w} \Leftarrow \mathbf{C}\mathbf{d}^{(k)}$;                              global sum vector w
            $s \Leftarrow \mathbf{d}'^{(k)}\mathbf{w}$;                              global sum scalar $s$
            $a \Leftarrow \frac{e_{new}}{s}$;  $\mathbf{x}^{(k+1)} \Leftarrow \mathbf{x}^{(k)} + a\ \mathbf{d}^{(k)}$
            IF k is divisible by 100: $\mathbf{r}^{(k+1)} \Leftarrow \mathbf{b} - \mathbf{C}\,\mathbf{x}^{(k+1)}$; global sum vector r
            ELSE:                     $\mathbf{r}^{(k+1)} \Leftarrow \mathbf{r}^{(k)} - a\,\mathbf{w}$
            $\mathbf{w} \Leftarrow \mathbf{M}^{-1}\mathbf{r}^{(k+1)}$;                        send w
            $e_{old} \Leftarrow e_{new}$; $e_{new} \Leftarrow \mathbf{r}'^{(k+1)}\mathbf{w}$;           global sum scalar $e_{new}$
            $b \Leftarrow \frac{e_{new}}{e_{old}}$;  $\mathbf{d}^{(k+1)} \Leftarrow \mathbf{w} + b\ \mathbf{d}^{(k)}$
            $k \Leftarrow k + 1$
```

Figure 1. Implemented parallel preconditioned conjugate gradient algorithm.

Equal amount of work on processors

Preliminary analysis of the code in solving test-day models indicated that most work is done in multiplication of the vector by the coefficient matrix of the MME. Consequently, the data was partitioned to processors by making the least-squares part computations as even as possible. Natural criterion in partitioning of the data was number of records. Pedigree and the preconditioner information were partitioned according to the partitioning of the least-squares part.

Minimization of required random access memory per processor

In practice, only those values that are required by the processor need to be stored. However, a general sparse vector may be inefficient both in memory usage and computing time. A better sparse vector storage method exploits structure of the MME. The MME was ordered such that in each processor most of the non-zero values were in two continuous vector blocks. The rest of the values were stored in a sparse list.

The equations of the MME were renumbered by cow family blocks (see talk by Martin Lidauer). The rest of the equations (attributed to sires, bull dams, and some fixed effects) are equations that “link” herds. The cow family blocks were assigned to processors making clusters of cow family blocks. The clusters were almost independent from each other; they were linked only by cows having records in different clusters (observations or progeny in herds that were in different clusters), and by the above mentioned equations that link herds.

Sparse vector presentation of the large vectors (**d**, **r**, **x**, **w**) had three parts: *private area* having values associated with equations in the cow family block cluster; *common area* having equations that link herds and are accessed by all processors; *sparse list* having values that are used by at least two processors and are not in the common area. In our application, *common area* contained phantom parent groups, sires, dams of sires, and across herd fixed effects. *Private area* had rest of the effects.

Minimization of message passing

The memory minimization described above decomposed the vectors into private and non-private areas. This translates naturally into local and communicated data. Communication was done in order to have the sparse list and the common area to be equal in all processors.

Application

Breeding values were estimated in the Finnish dairy cattle population using two random regression test-day models. The smaller model had 6,732,765 first lactation test-day records on milk yield from 24,321 herds. The larger model included 8,381,093, 3,933,460, and 3,933,556 first lactation test-day records on milk, fat, and protein yields from 26,312 herds. There were about 7.28 and 21.7 million unknowns in the MME of the small and large model, respectively.

Two computers were used: Cycle Ultra AXmp workstation (Sun clone at Agricultural Computing Centre) with at most 4 processors, and SGI Origin 2000 super computer (at CSC) with at most 24 processors. The future genetic evaluations in Finland are going to be calculated on the Cycle computer. The program was tested with both data sets using the two and four available processors. SGI Origin 2000 was used to illustrate scalability of the program up to 16 processors. Tests on SGI were performed with the small model using 2, 4, 8, 12, and 16 processors.. Data was read from the disk in all of the program runs.

Two parallel programs were compared. Both make the same communication and are in general terms the same. However, the programs differ in having a different data structure for the sparse vectors. The difference from programming point of view is a minor one. The mix99p program has physically separated the common area into the sparse list and continuous vector common area. In the mixp program these two vector areas were combined into one. There is still a separate sparse list area and a continuous vector block area. However, they reside physically in the same vector. As a consequence, number of communication calls is less although amount of communication is the same.

Measures of parallelization

All performance measures in Cycle were based on the wall clock time because we are interested in real time benefits of the parallel code. On the SGI we had to rely on sum of system and user time because there were other users that made wall clock time an unreliable measure. Speedup was calculated from the execution times. The one processor program used in this calculation was MiX99 that was introduced in talk by Martin Lidauer. It does not require parallel environment. The parallel solvers mixp and mix99p were tested as single processor programs as well. Because they are parallel solvers, their compilation and execution was done as normal parallel computing would require.

3. Results and discussion

Speedup on Cycle was close to 2 with 2 processors in solving either of the models (Table 1). With mix99p program speedup was even above 2 when solving the smaller model. However, the parallel program executed in a single processor mode was faster than the original serial code. Reason for this result is unclear. It may be due to better data locality that allowed better use of the cache memory. On the other hand the code is somewhat structurally different so that the compiler may have had it easier to optimize. Results with 4 processors are not as good but very good as

Table 1. Speedup and time when using two and four processors on SGI Origin 2000, and Cycle Ultra AXmp computers. Time is system+user time for the SGI, and wall clock time for the Cycle computer. Size of data set is in parenthesis.

Number of processors:	SGI (small)		Cycle (small)		Cycle (large)	
program	Time (min)	Speedup	Time (min)	Speedup	Time (h)	Speedup
1: mix99s	22.27	1.00	54.68	1.00	2.30	1.00
mix99p	-	-	50.13	1.09	2.26	1.02
mixp	23.47	0.95	52.35	1.04	2.34	0.98
2: mix99p	13.48	1.65	27.02	2.02	1.18	1.95
mixp	14.03	1.59	27.79	1.97	1.16	1.98
4: mix99p	7.43	3.00	14.06	3.89	0.65	3.53
mixp	7.27	3.06	14.15	3.86	0.64	3.60

the computing time is less than 28% of the serial program for both the smaller and larger model. SGI Origin 2000 gave smaller speedup values. This may be due to disk I/O. However, there may be other reasons related to the compiler or the computing architecture.

Speedups on SGI Origin 2000 in solving the problem up to 16 processors showed an increasing trend (Figure 2). The mixp solver had a better speedup with larger number of processors than with smaller when compared to execution time using the mix99p program (Figure 1). In general, increase in speedup was not as good when the number of processors was above 8. This is likely be due to increased communication between processors. Communication increased linearly as number of processors was increased. Communication increased from about 3 Mb between two processors to about 72 Mb with 16 processors. Total amount of messages sent between the processors was about 1.3% and 31% of the amount needed to store all solutions in a vector in the 2 and 16 processor cases, respectively.

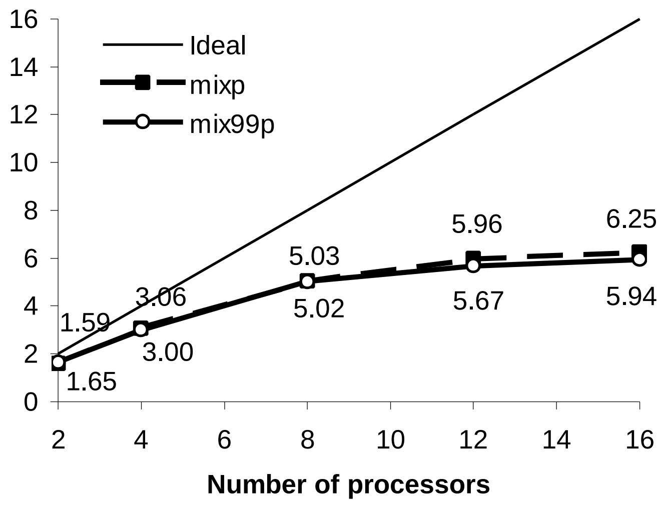

Figure 2. Plot of speedup versus number of processors on SGI Origin 2000 for the smaller data. The figures above (below) the curves correspond to mixp (mix99p).

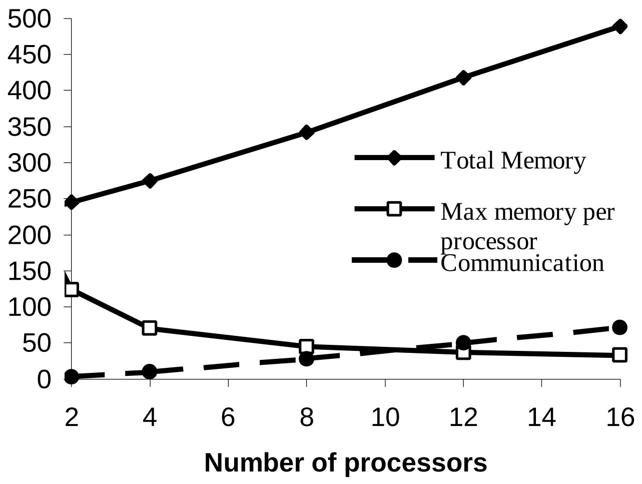

Figure 3. Total memory (in mega bytes) used over processors, maximum amount of memory per processor, and amount of communication per iteration on SGI Origin 2000 for the small data.

4. Conclusions

Parallel implementation of an iteration on data program using preconditioned conjugate gradient as solving algorithm yielded a good speedup. This was especially so for the Cycle computer having at most 4 processors. Parallelization of all steps of the algorithm and minimization of communication between the processors were mainly responsible for this result. Different computers showed different speedups. It seems that there are computer specific tricks that lead to better performance depending on the platform.

When the project was started it was estimated that it would take almost a month to calculate genetic values of three production traits for the Finnish dairy cattle population. Currently we expect this task to take about three days on Cycle using one processor. The results from this study suggest that the task can be made within 24 hours using 4 processors. So, in practice, this would allow doing the calculations during weekend while during week days the computer could be used for other purposes, e.g., accumulating new test day information.

Acknowledgement

This work has been supported by the European Commission’s Esprit programme on High-Performance Computing and Networking (Esprit project 23770).

References

Lidauer, M., E.A. Mäntysaari, I. Strandén, A. Kettunen, J. Pösö. 1998a. DMUIOD: A multitrait BLUP program suitable for random regression test day models. 6th WCGALP. Armidale, 27:463.

Lidauer, M., I. Strandén. 1998b. Experiences in using parallel computing to solve large test-day models. 49th Annual Meeting of EAAP, Warsaw, August 24-27, 1998.

Lidauer, M. et al. 1999. Improving convergence of iteration on data applied on large random regression test-day models by using preconditioned conjugate gradient. Submitted.

Madsen, P., M. Larsen. 1998. A parallel solver for multi-trait animal models. Interbull Open Meet., Rotorua, New Zealand, Bulletin 17: 96-99.

Madsen, P., M. Larsen. 1999. Monthly evaluation for better genes. CSC News 1/99: 9-11.

Schaeffer, L.R., B.W. Kennedy. 1986. Computing strategies for solving mixed model equations. J. Dairy Sci. 69:575-579.

Shewchuk, J.R. 1994. An introduction to the conjugate gradient methods without the agonizing pain. Tech. Rep. CMU-CS-94-125. Carnegie Mellon Univ., Pittsburgh.

Strandén, I., M. Lidauer. 1999. The good breed. CSC News 1/99:6-8.

Strandén, I., M. Lidauer. 1999. Solving Large Mixed Linear Models Using Preconditioned Conjugate Gradient Iteration. Submitted.