A Parallel Iterative Linear System Solver with Dynamic Load Balancing

Peter Christen*
Institut für Informatik, Universität Basel
4056 Basel, Switzerland
christen@ifi.unibas.ch

May 31, 2000

Abstract

This paper describes the design and implementation of a parallel iterative linear system solver for distributed memory multicomputers and workstation clusters. It is capable of applying *heterogeneous data distribution* and *dynamic load balancing* within an iterative solver routine at matrix level. Matrices as well as vectors are distributed heterogeneously according to the available performances of the processors, and redistributions are carried out at run time if the load of the processors changes. We present the concepts behind the chosen matrix data structures and load measurements, and discuss our dynamic load balancing algorithm. The results show the suitability of our approach.

1 Introduction and Related Work

Solving large and sparse linear systems is the core of many applications in scientific computing and engineering. On parallel machines, iterative solvers are specially attractive. Only a small number of computational kernels – which can be parallelized efficiently – are needed. The matrix remains unchanged (no fill-in is produced), so data structures and algorithms are much simpler than they are for direct parallel sparse linear system solvers.

*Current address: Computer Science Laboratory, RSISE, Australian National University, Canberra ACT 0200, Australia, pchristen@csl.anu.edu.au

Several packages for solving sparse linear systems by iterative methods on distributed memory multicomputers are available today (e.g. Aztec, PETSc or P-Sparslib). These packages are primarily designed to run efficiently on homogeneous and static platforms with a regular network topology. They all provide – more or less – the same functionality, namely routines to create and preprocess distributed matrices and vectors, various iterative methods and several preconditioners. Heterogeneous data distribution and dynamic load balancing (DLB) are not addressed by these packages, which makes them fairly unattractive for heterogeneous or dynamically changing platforms.

Modern distributed memory multicomputers – e.g. the *IBM SP-2* – often allow several users to run their programs simultaneously. Serial as well as parallel programs in interactive or batch mode can be started, using only one processor, a subset or all available processors. The load on each processor may thus change during the run time of a program as other programs may run on the same processor. For many parallel programs, the most loaded processor becomes a bottleneck if no load balancing is applied at run time.

An attractive alternative to expensive parallel machines are workstation cluster (WSC) [1]. They provide computational performances comparable to dedicated parallel machines, but have some special properties. They may be heterogeneous (different machines) and usually many (background) processes are running on a processor. The underlying LAN is often a shared communication channel with an irregular topology and slower than the high-speed networks of dedicated parallel machines. Additionally, many users share a cluster and workstations may fail or even be shut down.

What is needed for WSC as well as for modern distributed memory multicomputers are parallel programs that can distribute their data heterogeneously at starting time according to the currently available performances of the processors, and which can apply a dynamic adjustment of the data distribution according to the changing load at run time.

DLB for parallel programs is a current field of research. Several projects (e.g. Hector or MPVM) aim at providing transparent DLB for message passing programs within the message passing environment by migrating tasks of the parallel program from overloaded to underloaded processors. Other systems involve the application programmer, i.e. some calls to load balancing functions have to be inserted into a parallel program. Object oriented approaches try to hide the load balancing mechanism within their distributed objects.

The aim behind our work is to provide a parallel iterative linear system solver with DLB. We developed a library of routines which can adapt the data distribution (matrix and vectors) dynamically according to the changing loads of the processors.

2 Concept and Design

In our parallel iterative solver [2] work is proportional to the locally stored data (matrix rows and vector elements). This data is distributed proportional to the computational performances of the used processors. If processors are loaded unequally, the most loaded processor becomes a bottleneck of the parallel program. Removing data from an overloaded processor and redistributing it to other processors will balance the load and thus eliminate the bottleneck.

Redistributing data always introduces an overhead, as additional work has to be done and some nodes get more data and thus more work to do. It is therefore better to tolerate a certain amount of load imbalance than redistribute data too often. A *threshold* value has to be chosen, and redistribution is only applied if the load imbalance is above this threshold. A second parameter that affects the overhead and thus the efficiency of DLB in our approach is the *interval* – i.e. the number of iterations – between two possible redistribution steps. This two parameters (called *dlb_threshold* and *dlb_iteration*) have to be set by the user. Choosing optimal values for both parameters will result in a minimal run time, but finding such values is difficult, as they depend upon hardware characteristics, the problem size and structure as well as the load situation (how imbalanced the system is and how fast the load of an individual node changes).

The main ideas behind our algorithm for DLB within an iterative solver are (1) to design data structures for matrices and vectors that allow a redistribution of matrix rows and vector elements from one processor to another, (2) to measure the current load on all processors in every iteration and (3) to carry out the redistribution (as transparently as possible) if needed. We describe these three points in more detail in the following sections.

2.1 Data Distribution and Matrix Data Structures

Distributed data in our parallel solver consists of the coefficient matrix and vectors. Before data is distributed, the floating-point performance of each processor is measured with a micro-benchmark. Data is then distributed according to these performances. The more powerful a processor at this moment is (i.e. the faster it is with floating-point computations), the more matrix rows it gets. If more processors are available than are needed, the most powerful ones are chosen.

Matrices are distributed *block row-wise*, so each processor gets a contiguous set of matrix rows. Our *dynamic* sparse matrix data structure consists of an array of length N on each processor (with N the matrix dimension),

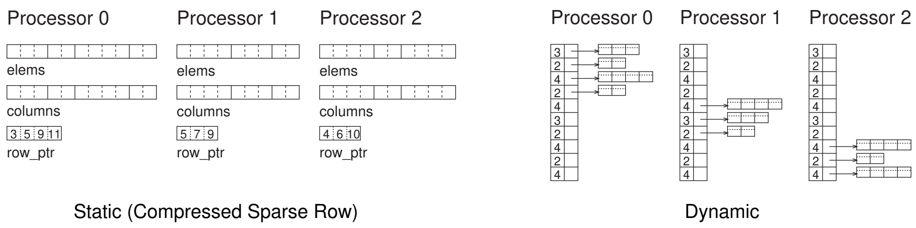

Figure 1: Sparse Matrix Data Structures

which contains pointers to the stored rows, as well as the number of non-zero elements (NZE) in the rows. Each sparse row consists of two arrays, one with the integer column indices and the other with the corresponding NZE stored as floating-point values. To compare the overhead of our dynamic data structure, we additionally provided our solver with the commonly used *Compressed Sparse Row* (CSR) data structure (which does not allow a data redistribution). An example of both data structures with a matrix of dimension 10 and 30 NZE can be seen in Figure 1.

With our distributed dynamic data structure it is possible to allocate and deallocate rows at run time and perform DLB by moving rows from one processor to another. Because a processor can only store a contiguous block of rows, redistribution can only be done by moving rows between *neighbouring* processors. The advantage of this method is that when a matrix-vector multiplication (MVM) is performed, each processor computes a contiguous part of the result vector, and no permutations of the matrix and vectors have to be managed. But there is the drawback that rows can not directly be moved from the most overloaded to the most underloaded processor.

2.2 Load Measurement

The aim of DLB is to achieve a minimal run time of a parallel program by equally distributing the work onto the processors at any time. The *load* of a processor is characterized as a measurement of the utilization of its various resources. There are several ways to measure the load of a Unix workstation. First, read the load by a system call. In Unix, however, one needs to have root privileges to access system (kernel) variables, so this method can not be applied by a normal user program. Secondly, use pipes and Unix commands like `w` or `iostat` to get the load. Unfortunately, this takes several hundred milliseconds. The third approach – the one we have chosen in our approach – is to measure the processing speed of the parallel application program itself on all processors. This method is almost system independent and only a small overhead is introduced.

Increment *dlb_counter*
IF ($dlb\_counter < dlb\_iteration$) THEN RETURN
$dlb\_counter = 0$
Compute current system imbalance *sys_imbalance*
IF ($sys\_imbalance < dlb\_threshold$) THEN RETURN
Compute new row distribution based on current loads
Find upper and lower neighbour processors P_{upper} and P_{lower}
Compute number of upper and lower exchange rows r_{upper} and r_{lower}

Redistribute matrix rows

Redistribute vectors according to new matrix row distribution
Update data structures
RETURN

Figure 2: Outline of the Dynamic Load Balancing Algorithm

In our parallel solver, we measure the run time (wall clock time) of the local computations (i.e. MVM, dot products and vector additions) within each iteration to get the current load. DLB then tries to balance the local computation times on all processors. If matrix rows are removed from a processor, its computation time is shortened and the load is reduced. Once the load is measured, it has to be broadcasted to all processors. Fortunately, this communication can be combined with the communication of the distributed dot products, so only a small overhead is introduced.

2.3 Dynamic Load Balancing

The general outline of our DLB algorithm can be seen in Figure 2. It works as follows. First the internal *dlb_counter* is increased. DLB is only performed if the value of this counter is equal to *dlb_iteration* (a parameter set by the user). Next, the system imbalance is computed using the current loads. If it is less than a chosen threshold, no redistribution is needed and the algorithm is left. It follows the computation of the new optimal matrix row distribution depending on the current loads. Each processor determines its neighbours and computes the number of matrix rows r_{upper} and r_{lower} it has to send to or receive from them. Matrix rows are exchanged next. As this communication can become quite time consuming, we have developed several methods to reduce this overhead (see below). The matrix row redistribution is done in three steps. First, new rows are allocated on a destination processor. Secondly, the rows are communicated as needed; and finally, unused rows are deallocated on the source processors. Several sparse matrix rows are communicated in one message to reduce startup times. After redistributing a

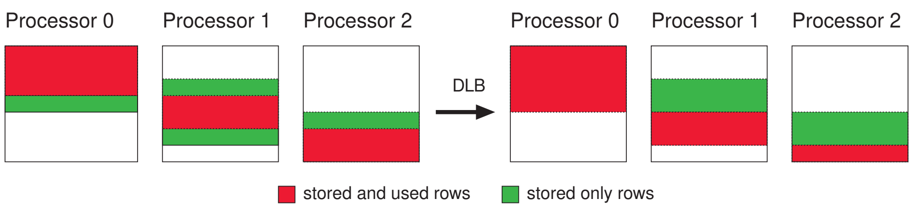

Figure 3: DLB with Overlapping Matrix Regions

matrix, vectors are redistributed as needed according to the new matrix distribution. Finally, all data structures are updated. The whole redistribution is completely transparent to the calling solver routine, i.e. before and after the DLB algorithm all data structures are consistent with the current data distribution.

When DLB within our parallel iterative solver is performed by moving matrix rows and vector elements from one processor to another, the amount of data to be communicated – i.e. matrix rows and vector elements – can become quite large. Because communication is usually much slower than computation, the DLB communication introduces a large overhead.

We have developed several methods to reduce the overhead of the DLB communication. The first one tries to overlap communication with computation (e.g. a MVM). The second method allows only the most overloaded processor to send rows to its neighbours. Another idea is to store matrix rows on several nodes redundantly and create *overlapping regions* (see Figure 3). In this method, no rows are moved physically between processors, so the only communication that takes place is the redistribution of vectors.

3 Implementation and Results

We have implemented our DLB algorithm in our solver *PAISS* [2] (**P**arallel **A**daptive **I**terative Linear **S**ystem **S**olver). This is a library of routines programmed in SPMD programming style in ANSI C which uses MPI [4] for communication. Currently the Conjugate Gradient (CG) method with polynomial and diagonal preconditioning is implemented. The prototype runs on *Sun* workstations, a Swiss *SCS Gigabooster* [5] and on an *IBM SP-2*.

Table 1 presents results achieved on a WSC with 6 *SUN Sparc-5* machines connected by a 10 MBit/s switched Ethernet. A linear system with the matrix *bcsstk17* (dimension 10974, 428650 NZE) from the *Harwell-Boeing* [3] collection has been solved. The DLB parameters have been set to *dlb_iteration* = 50 and *dlb_threshold* = 40%. The second workstations has been loaded artificially with another process performing floating-point computations. With

Processors	1	2	3	4	5	6
All Processors idle	178	105	85	72	71	75
One Processor busy, no DLB	342	231	163	112	97	84
Non-Overlap DLB	–	162	102	83	81	78
Overlap DLB	–	142	103	85	77	84
One-Sender DLB	–	141	100	84	95	85
Matrix-Overlap DLB	–	140	96	84	79	83

Table 1: CG Run Time for *bcsstk17* (Milliseconds per Iteration)

a small number of processors, our DLB algorithm can achieve good improvements. More timing results on several platforms can be found in [2].

Figure 4 shows a detailed view of a single test run with the matrix *bcsstk16* (dimension 4884, 209378 NZE) on three processors of a Swiss *SCS Gigabooster* [5]. As the second workstation gets loaded after 130 iterations, a redistribution step has been triggered in iteration 200. The DLB algorithm is called and matrix rows are removed from processor 1, so the computational times get balanced afterwards.

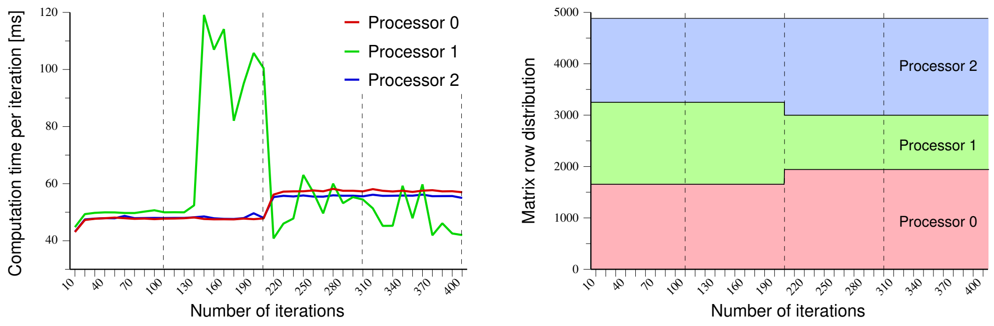

Figure 4: Computational Times and Matrix Redistribution for *bcsstk16*

4 Conclusions

We presented a parallel iterative linear system solver which is capable of applying dynamic load balancing at matrix level by redistributing rows and corresponding vector elements as needed by a changing load situation to get a balanced load. Although the overhead introduced is large, our approach can achieve improvements in the solver run time. A careful implementation – overlapping communication with computation or introducing overlapping regions – and choosing suitable values for the load balancing threshold and interval are required to achieve run time improvements. It is harder to achieve good improvements if the platform is very dynamic, because a new data distribution can already be outdated after a small number of iterations.

5 Acknowledgment

The author's stay in Australia has been made possible by grants from the *Swiss National Science Foundation* (SNF) and the *Novartis Stiftung, vormals Ciba-Geigy Jubiläums-Stiftung.*

References

[1] T.E. Anderson, D.E. Culler and D.A. Patterson. A case for NOW (Networks of Workstations). *IEEE Micro*, 15(1):54–64, February 1995.

[2] P. Christen. *A Parallel Iterative Linear System Solver with Dynamic Load Balancing.* Ph.D. thesis, University of Basel, February 1999.

[3] I.S. Duff, R.G. Grimes, and J.G. Lewis. User's Guide for the Harwell-Boeing sparse matrix test problems collection. Technical Report RAL-92-086, Computing and Information Systems Department, Rutherford Appleton Laboratory, Didcot, UK, 1992.

[4] Ohio Supercomputing Center, The Ohio State University. *MPI Primer/ Developing With LAM*, December 1995.

[5] B. Tiemann, H. Vonder Mühll, I. Hasler, E. Hiltebrand, A. Gunzinger and G. Tröster. Architecture and implementation of a single-board Desktop Supercomputer. In B. Hertzberger and G. Serazzi, Editors, *Lecture Notes in Computer Science 919: HPCN Europe 1995, Proceedings, Milan, Italy, 1995. Springer, 1995.*