Non-Blocking Load Balancing for Branch-and-Bound-Type Algorithms

Peer Ueberholz[1], Paul Willems[2], Mark Bull[3], and Bruno Lang[2]

[1] Niederrhein University of Applied Science, Depa rtment of Electrical Engineering and Computer Science, Reinarzstr. 49, D-47805 Krefeld, Germany
[2] University of Wuppertal, Institute for Applied Computer Science and Scientific Computing, D-42097 Wuppertal, Germany
[3] EPCC, University of Edinburgh, James Clerk Maxwell Building, Mayfield Road Edinburgh, EH9 3JZ, U.K.

Abstract. Branch-and-bound-type algorithms are used in a variety of areas, e.g., in the result-verifying solution of nonlinear systems. Since such algorithms are very computationally intensive, an efficient parallelisation is essential for many practical problems. When parallelising such algorithms one is faced with two problems. Since one does not know in advance how much computer time and memory each subproblem uses, a simple parallelisation quickly leads to poor load balancing, or to running out of memory on individual nodes. To address these issues, a sophisticated MPI implementation using a diffusion-like algorithm is presented, which shows linear speedup and good balance in memory up to a large number of processors. Although our load balancing approach is presented only in the context of a result-verifying nonlinear solver, it is generalisable to other branch-and-bound-type algorithms.

Key words: branch-and-bound algorithms, nonlinear systems, result-verifying algorithms, interval arithmetic, parallelisation, dynamic load balancing algorithms, diffusion algorithm, dimension exchange algorithm.

1 Introduction

Branch-and-bound-type algorithms are used in diverse areas, e.g., in the result-verifying solution of nonlinear systems or in global optimisation. The load balancing approach described in this paper applies to all branch-and-bound algorithms, although only the first application is discussed here.

Nonlinear systems $F(z) = 0$ arise as subproblems in diverse applications, one example being the search for singular states of a chemical process. Here, the variables z contain the state of the system, parameters for influencing the process (such as heating), and additional quantities describing the singularity. Since the parameters can be adjusted only within certain tolerances, a parametrically robust state requires that no singularity occurs for any parameter within some "box" $[z] = [z_1] \times \ldots \times [z_n]$ around the prospective point of operation. For security reasons one uses result-verifying algorithms to be *sure* that the system $F(z) = 0$ has no solution in $[z]$.

The basic branch-and-bound algorithm for the result-verifying solution of nonlinear systems proceeds as follows. Given a box $[z]$, evaluating the components F_i of the function F with appropriate interval-based methods yields intervals $[F_i]$ that are guaranteed to enclose the ranges of the F_i over the box $[z]$. If $0 \notin [F_i]$ for some i then the system cannot have a zero in $[z]$, and the box can be excluded ("bound"). If $0 \in [F_i]$ for all i then the system may have a zero in $[z]$. In this case the box is split into two or more subboxes, and the above test is applied recursively to these ("branch"). To make this procedure feasible in practice, it must be complemented with acceleration techniques such as an interval variant of Newton's method [1, 2]. Several such acceleration devices are included in the SONIC framework (**S**olver and **O**ptimizer for **N**onlinear Problems based on **I**nterval **C**omputation), which has been developed at the University of Wuppertal and RWTH Aachen University [3].

Hard problems lead to deep recursion trees with huge numbers of boxes to be considered and utilising parallelism is essential for solving these problems. An efficient parallelisation with OpenMP based on a task scheduling algorithm is described in [4]. For really hard problems, however, a larger number of processors, and thus an efficient parallelisation with MPI, is needed. Here we are faced with two problems: First, the required time to completely analyse a box may vary widely, depending on the effectiveness of the accelerators, and the size of the associated subtree. Second, it is not known in advance how many subboxes will result from the subdivision of a given box. This can lead to a memory problem on individual nodes.

Nearest neighbour algorithms can solve the above-mentioned problems. These distributed algorithms can follow a diffusion scheme, where each process balances its workload within a local domain of processes. In the following we will present an efficient version of a diffusion algorithm, which shows a good scaling behaviour up to large numbers of processors. We will compare this algorithm to a simple centralised master–worker algorithm, which shows a good scaling behaviour up to moderate numbers of processors.

2 Implementation of the diffusion algorithm

Our algorithm belongs to the nearest-neighbours class and operates in the diffusion scheme where load balancing is done iteratively based on communications between neighbouring processes, similar to the way it is done in the dimension exchange method. It is sender-initiated and asynchronous. The basic strategy is similar to the strategies proposed by Willebeek-LeMair et al. in [5]. Nearest neighbour algorithms do not need a "global view" of the work load, but each process tries to balance the work load locally by using only information about its own load and that of its neighbours. Here the neighbouring processes are defined by superimposing a virtual topology upon the processes, e.g., a d-dimensional torus. We have implemented the algorithm, in which all processes exchange information about their current workload as well as work with neighbour processes in a non-blocking way. Since there are no synchronisation points until the computations are finished or nearly finished we do not need to balance the work at any given point in time, we only must prevent processes running idle while others still have much work left over. This amounts to balancing the *overall* work done by the processes. Gau and Stadtherr [6] discuss three different versions of dynamic load balancing algorithms for a result-verifying solution of nonlinear systems and tested their scaling properties up to 32 processors. The best of these algorithms, named the asynchronous diffusive load balancing (ADLB), is similar to ours with a couple of differences. Most importantly, their load transmissions are receiver-initiated whereas ours are sender-initiated and we use a different migration strategy for work, which does not depend on a fixed threshold and sends work at most to one neighbour in one step. Furthermore our global termination detection is done in a centralised way and not by Dijkstra's token algorithm. With the implemented modifications our algorithm shows a high efficiency and scales up to 1024 processors. The main steps of the algorithm can be divided into three parts;

1. **Initialisation.** In the initialisation phase the processes are arranged in a d-dimensional torus, and the start boxes are distributed over the processes. Each process also receives the number ℓ_i of boxes in the worklists of its neighbours i. In addition a master process receives the total number of boxes stored on all processes (the total worklist length, ℓ_{global}).
2. **Work loop**. Throughout the branch-and-bound algorithm, each process manages its own list of boxes, stored in a worklist. It analyses the boxes in an iteration loop, until a stop signal is received from the master process. This can result in 0 to n new boxes, which are inserted in the worklist. There are three communications to be done in each iteration step:
   – *Process load evaluation.* Each process exchanges the number of boxes in its worklist with its neighbours.
   – *Task migration.* With the current information of the workload of the neighbouring processes, each process checks if one of its neighbours has substantially fewer boxes than itself. In this case some boxes are transferred to that neighbour.
   – *Update of the global worklist length.* The overall number of boxes in the local worklists changes if a box gets deleted or more than one new box is created. This change is sent to the master node.
3. **Stopping condition.** All processes have to be stopped if
   – a solution of the problem was found,
   – all boxes were analysed, or
   – a checkpoint signal was reached.

   In these cases, the master node sends a shutdown signal to all other nodes.

3 Performance

The algorithm was developed and tested on an IBM BlueGene/L system at the University of Edinburgh. Figure 1 shows the aggregate computing time vs. the

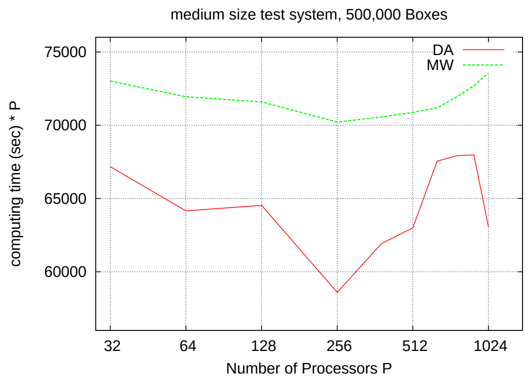

Fig. 1. Aggregate computing time on BlueGene (DA: diffusion algorithm, MW: master–worker algorithm).

number of processors for a test problem. The diffusion algorithm is about 10% faster than the master–worker algorithm. The fluctuations in computing time are due to different dimensions in the virtual process torus used. In summary, the diffusion algorithms and, to a somewhat lesser degree, also the master–worker algorithm both scale very well up to a large number of processors on different platforms. But one has to keep in mind that for the large problem, the memory of the master node was nearly exhausted for the master–worker algorithm on the BlueGene/L, while the memory usage of the diffusion algorithms is very well balanced. A further advantage of the non-blocking communication is the possibility to use this algorithm also for loosely coupled machines or even in the computational grid environment, because communication and computation can perfectly overlap as long as there is work left on a processor.

References

1. R.B. Kearfott: Rigorous Global Search: Continuous Problems, Kluwer Academic Publishers, Dordrecht, The Netherlands (1996).
2. A. Neumaier: Interval Method for Systems of Equations, Cambridge University Press, Cambridge, UK (2000).
3. T. Beelitz, A. Frommer, B. Lang, and P. Willems: Symbolic–numeric techniques for solving nonlinear systems. Proc. Appl. Math. Mech. 5(1), 705–708 (2005).
4. T. Beelitz, C.H. Bischof, and B. Lang: Efficient task scheduling in the parallel result-verifying solution of nonlinear systems, Reliab. Comput. 12(2), 141–151 (2006).
5. M. Willebeek-LeMair and A.P. Reeves: Strategies for dynamic load balancing on highly parallel computers, IEEE Trans. Parallel and Distributed Systems 4(9), 979–993 (1993).
6. C.-Y. Gau and M.A. Stadtherr: Dynamic load balancing for parallel interval-Newton using message passing. Comput. Chem. Eng. 26, 811–825 (2002).