Parallel Computing 40 (2014) 34–46

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

A study of shared-memory parallelism in a multifrontal solver

Jean-Yves L'Excellent [1], Wissam M. Sid-Lakhdar [*]

University of Lyon and LIP laboratory (UMR 5668 CNRS, ENS Lyon, Inria, UCBL), ENS Lyon, 46 allée d'Italie, 69364 Lyon Cedex 7, France

ARTICLE INFO

Article history:
Received 11 February 2013
Received in revised form 7 February 2014
Accepted 8 February 2014
Available online 17 February 2014

Keywords:
Shared-memory
Multi-core
NUMA
LU factorization
Sparse matrix
Multifrontal method

ABSTRACT

We introduce shared-memory parallelism in a parallel distributed-memory solver, targeting multi-core architectures. Our concern in this paper is pure shared-memory parallelism, although the work will also impact distributed-memory parallelism. Our approach avoids a deep redesign and fully benefits from the numerical kernels and features of the original code. We use performance models to exploit coarse-grain parallelism in an `OpenMP` environment while, at the same time, also relying on third-party optimized multithreaded libraries. In this context, we propose simple approaches to take advantage of NUMA architectures, and original optimizations to limit thread synchronization costs. The performance gains are analyzed in detail on test problems from various application areas. Although the studied code is a direct solver for sparse systems of linear equations, the contributions of this paper are more general and could be useful in a wider range of situations.

1. Introduction

Since the advent of multi-core systems, many efforts had to be taken to make existing software take advantage of these new architectures. The costs of synchronizations, the increasing gap between processor and memory speeds, the NUMA (Non-Uniform Memory Accesses) effects and the increasing complexity of modern cache hierarchies are among the main difficulties encountered. In this study, we are interested in a parallel direct approach for the solution of sparse systems of linear equations of the form

$$Ax = b, \tag{1}$$

where A is a sparse matrix, b is the right-hand side vector (or matrix) and x is the unknown. We consider that matrix A is unsymmetric but has a symmetric pattern, at the possible cost of adding explicit zeros when treating matrices with an unsymmetric pattern. We rely on the multifrontal method [1] to decompose A under the form $A = LU$. This method uses a task graph which is a tree called *assembly tree*, that must be processed from the leaves to the root following a topological order (i.e., children must be processed before their parent). At each node of the tree, a partial factorization of a small dense matrix is performed, possibly in parallel, leading to *node parallelism*; furthermore, nodes on distinct subtrees can be processed independently, resulting in so called *tree parallelism*. Because sparse direct solvers, in particular those using message-passing [2–4], often have a long development cycle, sometimes with many features added over the years, it is not always feasible to redesign them from scratch, even when computer architectures evolve a lot.

[*] Corresponding author. École Normale Supérieure de Lyon. Tel.: +33 472728352.
E-mail addresses: Jean-Yves.L.Excellent@ens-lyon.fr (J.-Y. L'Excellent), mohamed.sid_lakhdar@ens-lyon.fr (W.M. Sid-Lakhdar).
[1] Inria.

http://dx.doi.org/10.1016/j.parco.2014.02.003
0167-8191/© 2014 Elsevier B.V. All rights reserved.

In this paper, we consider a sparse direct solver in a pure shared memory environment using the example of the MUMPS code [5,6], although the methodology and contributions are more general. We study and combine the use of multithreaded libraries (in particular BLAS – Basic Linear Algebra Subprograms [7]), of loop-based fine-grain parallelism based on OpenMP directives; and of coarse-grain parallelism between independent tasks. For that, we propose an algorithm based on a performance model to decide when to use coarse-grain parallelism (tree parallelism) and when to use multithreaded libraries (node parallelism). On NUMA architectures, we show how the memory allocation policy and resulting memory affinity strongly impacts performance and that it should be chosen with care, depending on the threads working on each task. Furthermore, when treating independent tasks in a multithreaded environment, if no ready task is available, a thread that has finished its share of the work will wait for the other threads and become idle. In an OpenMP environment, we show how, technically, it is in that case possible to re-use idle cores to help other busy threads, dynamically increasing the amount of parallelism exploited.

In relation to the present work, we note that multithreaded sparse direct solvers aiming at addressing multi-core environments have been the object of a lot of work [2,8–17]. Our approach and contribution in this paper are different in several respects, as explained below.

We start from an existing code, originally designed for distributed-memory architectures, with a wide range of options and numerical features. One of our objectives is to show how such a code can be modified without too much redesign.

Most solvers manage all the parallelism themselves and use serial BLAS libraries on dense matrices whose size is sometimes arbitrarily limited. On the contrary, at least on large dense matrices, we aim at taking as much advantage as possible of existing multithreaded BLAS libraries that have been the object of a lot of tuning by specialists of dense linear algebra.

In the so called DAG-based approaches [10,15] (and in the codes described much earlier in references [8,12]), a much finer grain parallelism allows tree parallelism and node parallelism not to be separated: each individual task can be either a node in the tree or a subtask inside the node of the tree, which can start as soon as the dependencies on input to the task are met. This is also the case of the distributed-memory approach we start from, where a processor can, for example, start working on a parent node even when some work remains to be done at the child level [18]. In the current paper and in order to use simple mechanisms to manage the threads, we study and push as far as possible the approach consisting in using tree parallelism up to a certain level, and then switching to node parallelism, at the cost of a synchronization.

Many approaches rely on local/global tasks pools managed by a task dispatch engine [10,15,19]. In the case of NUMA architectures, the task dispatch engine aims at maintaining memory affinity, by placing tasks as close as possible to the cores that will work on them. For example, in HSL_MA87 [15], threads sharing the same cache also share a common local task pool. Moreover, workstealing strategies also tend to be NUMA-aware (see, for example, [19]), by preferably stealing tasks from the pools associated to close cores. Whereas a local memory allocation is generally preferred, a round-robin memory allocation may also provide interesting performance in some cases [10]. In this paper, we further show how different memory allocation policies may be exploited to efficiently perform local/global computations on NUMA architectures.

Some recent evolutions in linear algebra tend to use runtime systems [20–22]. Such approaches have also been experimented in the context of sparse direct solvers on SMP machines [23] and on modern architectures [24,25], sometimes with the possibility to also address GPU accelerators. However, numerical pivoting leading to dynamic task graphs, specific numerical features, or application-specific approaches to scheduling, still make it hard to use generic runtime systems in all cases. Had we used such runtime systems instead of OpenMP, most observations and contributions of this paper would still apply.

This paper is organized as follows. In Section 2, we present the multifrontal approach used and the software we rely on, together with our experimental setup. In Section 3, we briefly describe the fork-join approach to multithreaded parallelism, based on multithreaded BLAS libraries and OpenMP directives. In Section 4, we propose and study an algorithm to go further in the parallelization thanks to a better granularity of parallelism in the part of our task graph (the assembly tree) where this is necessary. In Section 5, the case of NUMA architectures is studied, and we show how memory accesses can dramatically impact performance. In Section 6, because the approach retained involves a synchronization when switching from one type of parallelism to the other, we study how idle cores can be dynamically re-assigned to help busy threads. Finally, we conclude by summarizing the lessons learned and we give some perspectives to this work.

2. Context of the study

2.1. Multifrontal method and solver

We refer the reader to [1,26] for an overview of the multifrontal method. In this section, we only provide the algorithmic details of our multifrontal solver that will be necessary in the next sections.

As said in the introduction, the task graph in the multifrontal method is a tree called *assembly tree*. At each node of this tree, a dense matrix called *front* or *frontal matrix* is first assembled, using contributions from its children and entries of the original matrix. Some of its variables are then factorized (partial factorization), and the resulting Schur complement (block updated but not yet factorized) is copied for future use. The Schur complement is also called *contribution block*, because it will contribute to the assembly of the parent's frontal matrix. At the root node, a full factorization is performed.

The three steps above (namely: assembly, factorization and stacking) correspond to three computational kernels of our multifrontal solver, which we describe in Algorithms 1, 2, and 3. When assembling contributions in the frontal matrix of