Parallel Computing 56 (2016) 1–17

Contents lists available at ScienceDirect

Parallel Computing

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

Fast Matlab compatible sparse assembly on multicore computers

Stefan Engblom*, Dimitar Lukarski

Division of Scientific Computing, Department of Information Technology, Uppsala University, SE-751 05 Uppsala, Sweden

ARTICLE INFO

Article history:
Received 21 June 2014
Revised 23 October 2015
Accepted 1 April 2016
Available online 6 April 2016

2010 MSC:
Primary: 68W10
Secondary: 65Y10

Keywords:
Sparse matrix
Column compressed format
Assemble
Matlab

ABSTRACT

We develop and implement in this paper a fast *sparse assembly* algorithm, the fundamental operation which creates a compressed matrix from raw index data. Since it is often a quite demanding and sometimes critical operation, it is of interest to design a highly efficient implementation. We show how to do this, and moreover, we show how our implementation can be parallelized to utilize the power of modern multicore computers. Our freely available code, fully Matlab compatible, achieves about a factor of 5 $\times$ in speedup on a typical 6-core machine and 10 $\times$ on a dual-socket 16-core machine compared to the built-in serial implementation.

© 2016 Elsevier B.V. All rights reserved.

1. Introduction

The popular Matlab programming environment was originally built around the insight that most computing applications in some way or the other rely on storage and manipulations of one fundamental object — the *matrix*. In the early 90s an important update was made with the support of a *sparse* storage format as presented in [7]. In that paper the way sparse matrices are managed in an otherwise dense storage matrix environment is described, including the initial creation of a sparse matrix, some basic manipulations and operations, and fundamental matrix factorizations in sparse format.

As a guiding principle the authors formulate the *“time is proportional to flops”*–rule [7, p. 334]:

> The time required for a sparse matrix operation should be proportional to the number of arithmetic operations on nonzero quantities.

The situation is somewhat different today since flops often can be considered to be “free” while memory transfers are, in most cases, the real bottlenecks of the program. With the multicore era here to stay programs need to be threaded in order to utilize all hardware resources efficiently. This is a non-trivial task and requires some careful design [1].

* Corresponding author. Tel.: +46 18 471 27 54; fax: +46 18 51 19 25.
E-mail addresses: stefane@it.uu.se (S. Engblom), dimitar.lukarski@it.uu.se (D. Lukarski).
URL: http://user.it.uu.se/~stefane (S. Engblom)

http://dx.doi.org/10.1016/j.parco.2016.04.001
0167-8191/© 2016 Elsevier B.V. All rights reserved.

In this paper we consider a sole sparse operation, namely the initial *assembly* operation as implemented by the Matlab function `sparse`;

```
>> S = sparse(i, j, s, m, n, nzmax);
```

After the call, S contains a sparse representation of the matrix defined by $S(i_k, j_k) = s_k$ for k a range of indices pointing into the vectors $\{i, j, s\}$, *and where repeated indices imply that the corresponding elements are to be summed together*. Many applications naturally lead to substantial repetitions of indices and the implied reduction must be detected and handled efficiently. For example, in the important case of assembly in linear finite element methods for partial differential equations, the resulting sparse matrix has a sparsity pattern which is identical to that of the matrix representation of the underlying triangular or tetrahedral mesh when viewed as a *graph*. The number of collisions during the assembly then corresponds exactly to the connectivity of the nodes in this graph.

Since the assembly must be performed before any other matrix operations are executed, the performance may become a bottleneck. The typical example is for dynamic nonlinear partial differential equations (PDEs), where re-assembly occurs many times as a numerical time integration proceeds, including during the iterations of the nonlinear solver. Thus, with the assembly process a quite time-consuming operation which is repeatedly performed, it cannot always be amortized over subsequent operations. Notably, in the truly large case presented in [[9], Section 5.1.2–5.1.3], the performance of the sparse assembly is found to be the reason behind the loss of strong scaling beyond a few thousands of cores.

Algorithms for sparse assembly have caught the attention also by others. General assembly via an intermediate hashed data format is considered in [2], where serial performance experiment in the PETSc library are also reported. As a follow-up on [7], in [15] the design of sparse matrices in Matlab∗P, a kind of parallel version of Matlab, is discussed. Unfortunately, little information about the current status of this language is available. More recently, a “graphBLAS” [10] has been suggested, where one of the operations, `BuildMatrix`, corresponds to the `sparse` function.

As mentioned, finite element methods naturally lead to the assembly of large sparse matrices. A stack-based representation specially designed for this application is suggested in [8], and is also implemented there using a hybrid parallel programming model on a Cray XE6. Another approach is reported in [4], where the assembly of finite element sparse matrices in both Matlab and Octave is considered using these high-level languages directly.

Using the “time is proportional to flops”–rule as a guiding principle – but paying close attention to memory accesses – we provide a fast re-implementation of the `sparse` function. The resulting function `fsparse` is Matlab compatible, memory efficient, and parallelizes well on modern multicore computers. Moreover, it is well tested and has been freely available in the public domain for quite some time.

In Section 2 we describe in some detail the algorithm proposed, which can be understood as an efficient index-based sorting rule. In Section 3 parallelization aspects are discussed and performance experiments are made in Section 4, where the memory bound character of the operation is also highlighted. In general, with most sparse algorithms, there are not enough non-trivial arithmetic operations to hide the format overhead and data transfer costs [3]. A summarizing discussion around these issues is found in Section 5.

Availability of software. The code discussed in the paper is publicly available and the performance experiments reported here can be repeated through the Matlab–scripts we distribute. Refer to Section 5.1 for details.

2. A fast general algorithm for sparse assembly

In this section we lay out a fast algorithm for assembling sparse matrices from the standard index triplet data. A description of the problem is first offered in Section 2.1, where some alternative approaches and extensions of the problem are also briefly mentioned. The formats of input and output are detailed in Section 2.2 after which the algorithm is presented stepwise in Section 2.3. A concluding complexity analysis in Section 2.4 demonstrates that the algorithm proposed has a favorable memory access pattern without requiring large amounts of auxiliary memory.

2.1. Description of the problem

The column compressed sparse (CCS)[1] is the sparse matrix storage format supported by Matlab but has also enjoyed a widespread use in several other packages. Given a 4–by–4 matrix S defined by

$$S = \begin{pmatrix} 10 & 0 & 0 & -2 \\ 3 & 9 & 0 & 0 \\ 0 & 7 & 8 & 7 \\ 3 & 0 & 8 & 5 \end{pmatrix}, \tag{2.1}$$

[1] Note that the abbreviation ‘CSC’ is also in widespread use.

Download English Version:

https://daneshyari.com/en/article/523791

Download Persian Version:

https://daneshyari.com/article/523791

Daneshyari.com