

# ALGORITHMIC SKELETONS FOR GENERAL SPARSE MATRICES ON MULTI-CORE PROCESSORS

Philipp Ciechanowicz  
Department of Information Systems  
Westfälische Wilhelms-Universität Münster  
Leonardo-Campus 3  
48149 Münster, Germany  
email: ciechanowicz@ercis.de

## ABSTRACT

We develop an extension of the Muenster Skeleton Library Muesli by a distributed data structure for general sparse matrices. The data structure supports data parallel algorithmic skeletons such as *fold*, *map*, and *zip*. Our implementation is highly flexible, object-oriented and makes use of the C++ template mechanism. As a result, the storable data type as well as the compression and distribution scheme can easily be changed and even be substituted by a user-defined one. As a unique feature, our implementation not only supports multi-processor architectures, but also efficiently makes use of current multi-core processors.

## KEY WORDS

algorithmic skeletons, multi-core processing, sparse matrix

## 1 Introduction

Sparse matrices play an important role in numerical analysis: The discretization of partial differential equations with the finite element method or the description of graphs by means of an adjacency matrix often results in a matrix primarily populated with zeros. Especially for very large matrices it is beneficial to use special data structures that take advantage of its sparse structure. Thus, operations can be performed faster while simultaneously consuming less memory compared to standard data structures. To speed up computations even more, multi-processor and/or multi-core systems are used frequently. In such an environment the development of parallel applications can be tedious and error-prone due to synchronization and communication problems. However, *algorithmic skeletons* offer an easy to use and convenient way to hide these problems [1, 2, 3]. They can best be described as typical parallel computation patterns and are used to manipulate the data structure in parallel. The main contribution of this paper is to develop a distributed data structure for general sparse matrices which is both easy to use and highly flexible. This is achieved by providing the following core features:

- Support for various data parallel skeletons such as *fold*, *map*, and *zip* in terms of member functions in order to manipulate the data structure in parallel.

- Support for arbitrary compression schemes. In fact, the user has the option to extend our predefined compression schemes and implement her own.
- Support for arbitrary distribution schemes. The user can define how the sparse matrix is distributed across the processors. This load balancing mechanism is highly flexible, since again our predefined schemes can be extended.
- Besides supporting multi-processor architectures with a distributed memory such as clusters, our data structure also makes use of multi-core processors with a shared memory architecture such that some skeletons and auxiliary functions can be executed even faster.

The remainder of this paper is structured as follows: First of all, Section 2 recapitulates two basic storage schemes for sparse matrices. Section 3 then introduces the key concepts and design goals on which our implementation is based. After that, Section 4 covers some implementation details. Section 5 shows the results of a couple of tests which we have conducted on a multi-core workstation cluster. Related work is discussed in Section 6. Finally, Section 7 summarizes the main findings and gives an outlook to future work.

## 2 Storage Schemes

This section introduces two important storage schemes for sparse matrices which we have implemented for our data structure: Compressed Row Storage (CRS) and Block Sparse Row (BSR). Readers already familiar with them may skip this section and proceed to Section 3. All examples are based on a  $n \times m$  sparse matrix  $A$  with  $n, m \in \mathbb{N}$ .

### 2.1 Compressed Row Storage

CRS, sometimes also referred to as Compressed Sparse Row, only stores the non-zero values of a sparse matrix by using three arrays [4]:

- The array  $a$  stores all non-zero values of  $A$  row-wise (cf. Figure 1). Therefore,  $a$  is of length  $nnz$  with  $nnz$  being the number of non-zero elements.