# An Educational Module Illustrating How Sparse Matrix-Vector Multiplication on Parallel Processors Connects to Graph Partitioning

M. Ali Rostami[1](✉) and H. Martin Bücker[1,2]

[1] Institute for Computer Science, Friedrich Schiller University, 07737 Jena, Germany
rostamiev@gmail.com
[2] Michael Stifel Center Jena for Data-Driven and Simulation Science, 07737 Jena, Germany

**Abstract.** The transition from a curriculum without parallelism topics to a re-designed curriculum that incorporates such issues can be a daunting and time-consuming process. Therefore, it is beneficial to complement this process by gradually integrating elements from parallel computing into existing courses that were previously designed without parallelism in mind. As an example, we propose the multiplication of a sparse matrix by a dense vector on parallel computers with distributed memory. A novel educational module is introduced that illustrates the intimate connection between distributing the data of the sparse matrix-vector multiplication to parallel processes and partitioning a suitably defined graph. This web-based module aims at better involving undergraduate students in the learning process by a high level of interactivity. It can be integrated into any course on data structures with minimal effort by the instructor.

## 1 Introduction

With the current and noticeable trend toward computer architectures in which multiple processing elements are solving the same problem simultaneously, the need for training future generations of scientists and engineers in parallel computing has become a critical issue. The ubiquity of parallelism is now a fact and is beyond dispute among the experts. However, it is currently not sufficiently recognized in the general academic world. In fact, today, the vast majority of undergraduate curricula in science and engineering, including computer science, does not involve parallelism at all.

For the time being, the absence of parallelism in undergraduate curricula can be accepted for general science and engineering. However, the situation is different for computer science, computer engineering, and computational science. In these disciplines, the importance and relevance of parallelism in all types of computational environments is so high that corresponding undergraduate programs should—if not have to—include topics from parallel and distributed computing. Despite the importance and omnipresence of parallelism in today's computing landscape, its integration into existing degree programs is typically difficult. One

S. Hunold et al. (Eds.): Euro-Par 2015 Workshops, LNCS 9523, pp. 135–146, 2015.
DOI: 10.1007/978-3-319-27308-2_12

of the authors of the present paper was involved in trying to embed a mandatory parallel computing course into the undergraduate programs in computer science at two German universities, RWTH Aachen University in the early 2000 s and Friedrich Schiller University Jena in 2013. Probably the most important lesson learned from these two unsuccessful attempts is that, while it is easy to find arguments to integrate parallel computing, it is extremely difficult to find a common consensus on those contents that will have to be replaced when transforming an existing into a new curriculum. According to the author's experiences at these two German universities, the situation is quite different when degree programs are being developed from scratch. This is witnessed by the successful integration of parallel computing courses into the following new degree programs: computational engineering science (bachelor and master) as well as simulation science (master) both at RWTH Aachen University and computational and data science (master) at Friedrich Schiller University Jena.

The focus of this paper is on the integration of elements from parallel computing into existing undergraduate programs whose mandatory courses do not involve parallelism. One option is to integrate parallel computing into elective courses. This strategy was successfully applied in the undergraduate program in computer science at RWTH Aachen University [2,4–6] but has the disadvantage that it reaches only a small subset of all enrolled students because the overall teaching load of the department is somehow balanced among competing elective courses in various areas including computer security, database systems, and human computer interaction, to name a few.

Another approach that we advocate in this paper is to integrate a narrow topic from parallel computing into an existing mandatory course. Here, we choose a course in data structures because it is among the core courses in any computing curriculum. We consider the multiplication of a sparse matrix by a dense vector on a parallel computer with distributed memory. We first sketch, in Sect. 2, a simple data structure for a sparse matrix and quote a serial algorithm for the matrix-vector multiplication. In Sect. 3 we briefly summarize the standard issues concerned with finding a suitable data distribution for that operation on a parallel computer and point out the relation to graph partitioning in Sect. 4. The new contribution of this paper is given in Sect. 5 where we introduce an interactive educational module illustrating the connection between finding a data distribution and partitioning a graph. Finally, we point the reader to related work in Sect. 6.

## 2 A Simple Sparse Matrix Data Structure

Undergraduate students typically think of a matrix as a simple aggregating mechanism that stores some entries in the form of a two-dimensional array. That is, they associate with this term a general dense matrix without any structure. However, in practice, matrices arising from a wide range of different application areas are typically "sparse." According to Wilkinson, a matrix is loosely defined to be sparse *whenever it is possible to take advantage of the number and location of its nonzero entries*. In a course on data structures, sparse matrices offer the opportunity to show students that the range of data structures and operations defined on matrices is much broader than the elementary two-dimensional aggregating mechanism.

Since the focus of this article is not on data structures for sparse matrices, we give only a simple example. Let $A$ denote a sparse $N \times N$ matrix and consider the matrix-vector multiplication

$$\mathbf{y} \leftarrow A\mathbf{x} \tag{1}$$

where the $N$-dimensional vector $\mathbf{y}$ is the result of applying $A$ to some given $N$-dimensional vector $\mathbf{x}$. Then, the $i$th entry of $\mathbf{y}$ is given by

$$y_i = \sum_{j \text{ with } a_{ij} \neq 0} a_{ij} \cdot x_j, \quad i = 1, 2, \ldots, N, \tag{2}$$

where an algorithm that exploits the sparsity of $A$ does not run over all elements of the $i$th row of $A$, but only over all elements that are nonzero.

In the compressed row storage (CRS) data structure, the nonzeros of a sparse matrix are stored in three one-dimensional arrays as follows:

```
CRS_matrix = record
   value : array[1 .. nnz] of REAL
   col_ind: array[1 .. nnz] of INTEGER
   row_ptr: array[1 .. N+1] of INTEGER
end_record
```

Here, the number of nonzero elements of $A$ is denoted by `nnz`. A nonzero element $a_{ij}$ is stored in `value`$(k)$ if and only if its column index $j$ is stored in `col_ind`$(k)$ and its row index $i$ satisfies `row_ptr`$(i) \leq k <$ `row_ptr`$(i+1)$. Then, assuming `row_ptr`$(N+1) :=$ `nnz` $+ 1$, it is well known that the matrix-vector multiplication (1) is computed by the pseudocode

```
for (i = 1; i <= N; ++i)
    y[i] = 0;
    for (j = row_ptr[i]; j < row_ptr[i + 1]; ++j)
        y[i] += value[j] * x[col_ind[j]];
```

The key observation for students is that there is some matrix data structure that stores and operates on the nonzeros only. Further storage schemes for and operations on sparse matrices are described in various books [13,15,19–21].

## 3 Sparse Matrix-Vector Multiplication Goes Parallel

The sparse matrix-vector multiplication is also an illustrating example to introduce parallelism. By inspecting (2) it is obvious that all entries $y_i$ can be computed independently from each other. That is, the matrix-vector multiplication is easily decomposed into tasks that can execute simultaneously. However, it is not obvious to undergraduates how to decompose data required by these tasks. The need for the decomposition of data to parallel processes is lucidly described by introducing a parallel computer as a network that connects multiple serial computers, each with a local memory. This way, an undergraduate course can easily introduce, in a combined way, data structures for sparse matrices that are more advanced than simple two-dimensional arrays as well as parallelism in the form of multiple processes that operate on data accessible via distributed memory.

The following questions then naturally arise: If data representing $A$, $\mathbf{x}$ and $\mathbf{y}$ are distributed to multiple processes, to what extent does this data distribution have an effect on the computation $\mathbf{y} \leftarrow A\mathbf{x}$? What are the advantages and disadvantages of a given data distribution? What are the criteria for evaluating the quality of a data distribution? How should data be distributed to the processes ideally?

To discuss such questions with undergraduates who are new to parallel computing we suggest to consider the following simple data distribution. The nonzero elements of $A$ are distributed to processes by rows. More precisely, all nonzeros of a row are distributed to the same process. The vectors $\mathbf{x}$ and $\mathbf{y}$ are distributed consistently. That is, if a process stores the nonzeros of row $i$ of $A$ then it also stores the vector entries $x_i$ and $y_i$. Given a fixed number of processes $p$, a data distribution may be formally expressed by a mapping called *partition*

$$P : I \rightarrow \{1, 2, \ldots, p\}$$

that decomposes the set of indices $I := \{1, 2, \ldots, N\}$ into $p$ subsets $I_1$, $I_2$, ..., $I_p$ such that

$$I = I_1 \cup I_2 \cup \cdots \cup I_p$$

with $I_i \cap I_j = \emptyset$ for $i \neq j$. That is, if $P(i) = k$ then the nonzeros of row $i$ as well as $x_i$ and $y_i$ are stored on process $k$.

Since the nonzero $a_{ij}$ is stored on process $P(i)$ and the vector entry $x_j$ is stored on process $P(j)$, one can sort the terms in the sum (2) according to those terms where both operands of the product $a_{ij} \cdot x_j$ are stored on the same process and those where these operands are stored on different processes:

$$y_i = \sum_{\substack{j \text{ with } a_{ij} \neq 0 \\ P(i) = P(j)}} a_{ij} \cdot x_j + \sum_{\substack{j \text{ with } a_{ij} \neq 0 \\ P(i) \neq P(j)}} a_{ij} \cdot x_j, \quad i = 1, 2, \ldots, N. \tag{3}$$

For the sake of simplicity, we assume that the product $a_{ij} \cdot x_j$ is computed by the process $P(i)$ that stores the result $y_i$ to which this product contributes. By (3), the data distribution $P$ has an effect on the amount of data that needs to be communicated between processes. It also determines which processes communicate with each other. Since, on today's computing systems, communication needs significantly more time than computation, it is important to find a data distribution using a goal-oriented approach. A data distribution is desirable that balances the computational load evenly among the processes while, at the same time, minimizes the communication among the processes.

## 4 An Undirected Graph Model for Data Partitioning

The problem of finding a data distribution is also interesting from another perspective of undergraduate teaching. It offers the opportunity to demonstrate that a theoretical model can serve as a successful abstraction of a practical problem. More precisely, a formal approach using concepts from graph theory is capable of tackling the data distribution problem systematically.

To this end, we now assume that the nonzero pattern of the nonsymmetric matrix $A$ is symmetric. Then, the matrix can be represented by an undirected graph $G = (V, E)$. The set of nodes $V = \{1, 2, \ldots, N\}$ is used to associate a node to every row (or corresponding column) of $A$. The set of edges

$$E = \{(i, j) \mid i, j \in V \text{ and } a_{ij} \neq 0 \text{ for } i > j\}$$

describes the nonzero entries. Here, the condition $i > j$ indicates that the edge $(i, j)$ is identical to the edge $(j, i)$ and that there is no self-loop in $G$. The data distribution to $p$ processes is then represented by the partition

$$P : V \to \{1, 2, \ldots, p\}$$

that decomposes the set of nodes $V$ of the graph into $p$ subsets $V_1$, $V_2$, …, $V_p$ such that

$$V = V_1 \cup V_2 \cup \cdots \cup V_p$$

with $V_i \cap V_j = \emptyset$ for $i \neq j$.

Then, (2) is reformulated in terms of graph terminology by

$$y_i = a_{ii} \cdot x_i + \sum_{\substack{(i,j)\in E \\ P(i)=P(j)}} a_{ij} \cdot x_j + \sum_{\substack{(i,j)\in E \\ P(i)\neq P(j)}} a_{ij} \cdot x_j .$$

Here, the first two terms of the right-hand side can be computed on process $P(i)$ without communication to any other process. The condition $P(i) \neq P(j)$ in the last term shows that the computation of $a_{ij} \cdot x_j$ requires communication between process $P(i)$ which stores $a_{ij}$ and process $P(j)$ which stores $x_j$. Minimizing interprocess communication then roughly corresponds to minimizing the number of edges connecting nodes in different subsets $V_i$ of the partition $P$. This number of edges is called the *cut size* and is formally defined by

$$\text{cutsize}(P) = \big|\{(i, j) \in E \mid P(i) \neq P(j)\}\big|. \tag{4}$$

In this graph model, the cut size does not exactly correspond to the number of words communicated between all processes in the computation of $\mathbf{y} \leftarrow A\mathbf{x}$ for a given partition $P$. However, it gives a reasonable approximation to this amount of communicated data called the *communication volume*; see the corresponding discussion in [17]. The communication volume is exactly described by the cut size if the underlying model is changed from an undirected graph to a hypergraph [11,12,22].

Assuming that the number of nonzeros is roughly the same for each row of $A$, the computation is evenly balanced among the $p$ processes if the partition $P$ is $\varepsilon$-balanced defined as

$$\max_{1\leq i\leq p} |V_i| \leq (1+\varepsilon)\frac{|V|}{p}, \tag{5}$$

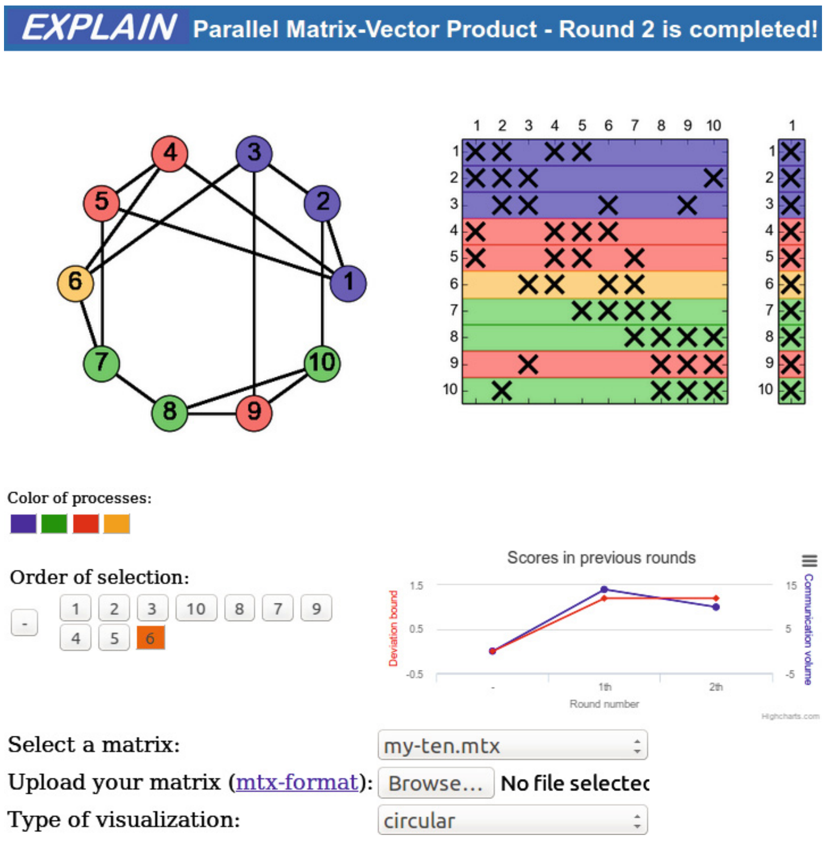
for some given $\varepsilon > 0$. The graph partitioning problem consists of minimizing the cut size of an $\varepsilon$-balanced partition. It is a hard combinatorial problem [14].

## 5 An Educational Module Illustrating the Connection

To illustrate the connection between computing a sparse matrix-vector multiplication in parallel and partitioning an undirected graph, we propose a novel educational module. This module is part of a growing set of educational modules called EXPLoring Algorithms INteractively (EXPLAIN). This collection of web-based modules is designed to assist in the effectiveness of teachers in the classroom and we plan to make it publicly available in the near future. Figure 1 shows the overall layout of this interactive module for sparse matrix-vector multiplication. The top of this figure visualizes—side by side—the representation of the problem in terms of the graph $G$ as well as in terms of the matrix $A$ and the vector $\mathbf{x}$. Below on the left, there is a panel of colors representing different processes and another panel displaying the order of selecting vertices of the graph. Next, on the right, there is a score diagram recording values characterizing communication and load balancing. At the bottom part, there are input controls used to select a matrix from a predefined set of matrices, to upload a small matrix, and to choose the layout of the graph vertices.

The first figure gives an overall impression of the status of the module after a data distribution is completed. Here, $p = 4$ processes represented by the colors blue, green, red, and yellow get data by interactive actions taken by the student. Figure 2 now shows the status of the module in a phase that is more related to the beginning of that interactive procedure. For a given matrix, the student can distribute the data to the processes by first clicking on a color and then clicking on an arbitrary number of vertices. That is, the distribution of vertices to a single process is determined by first clicking on a color $j$ and then clicking on a certain number of vertices, say $i_1, i_2, \ldots, i_s$ such that $P(i_1) = P(i_2) = \cdots = P(i_s) = j$. Then, by clicking on the next color, this procedure can be repeated until all vertices are interactively colored and, thus, the data distribution $P$ is finally determined.

Figure 2 illustrates the situation after the student distributed vertices 1, 2 and 3 to the blue process and the vertices 7, 8 and 10 to the green process. By interactively assigning a vertex to a process, not only the vertex is colored by the color representing this process, but also the row in the matrix as well as the corresponding vector entry of $\mathbf{x}$ are simultaneously colored with the same color. This way, the data distribution is visualized in the graph and in the matrix



Select a matrix: my-ten.mtx

Upload your matrix (mtx-format): No file selectec

Type of visualization: circular

**Fig. 1.** Overall structure of the sparse matrix-vector multiplication module (Color figure online).

simultaneously which emphasizes the connection between the matrix representation and the graph representation of that problem. The panel labeled "Order of selection" records the order of the vertices that are interactively chosen. By inspection from that panel in Fig. 1, we find out that the status depicted in Fig. 2 is an intermediate step of the interactive session that led to the data distribution in Fig. 1. Any box labeled with the number of the chosen vertex in that panel is also clickable allowing the student to return to any intermediate state and start a rearrangement of the data distribution form that state.

In EXPLAIN, the term "round" refers to the process of solving a single instance of a given problem. In this module, the problem consists of distributing all data needed to compute the matrix-vector product to the processes. Equivalently, the distribution of all vertices of the corresponding graph to the processes is a round. Suppose that round 2 is completed in Fig. 1. Then, the student can explore the data distribution in more detail by clicking on a color in the panel labeled "Color of processes." Suppose that the student chooses the red

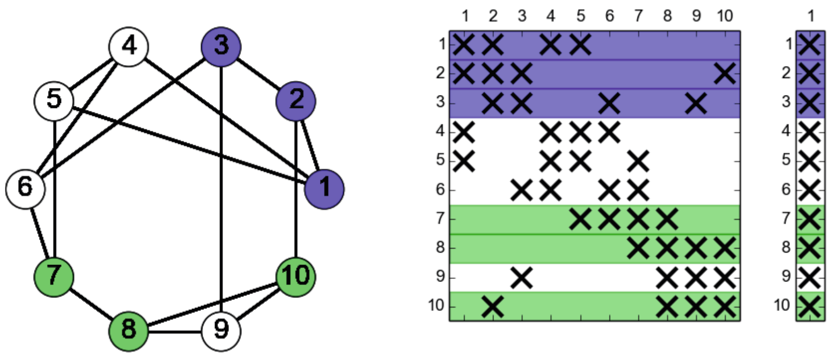


**Fig. 2.** The intermediate state after the student selected six vertices.

process, then this action will modify the appearance of the vector **x** in the matrix representation to the state given in Fig. 3. Here, all vector entries that need to be communicated to the red process are now also colored red. The background color still represents the process that stores that vector entry. This illustrates, for instance, that the vector entry $x_1$ is communicated from the blue process to the red process when computing $A\mathbf{x}$ using this particular data distribution. The matrix representation visualizes the reason for this communication. There is at least one row that is colored red and that has a nonzero element in column 1. In this example row 4 and row 5 satisfy this condition. Thus, $x_1$ is needed to compute $y_4$ and $y_5$. Again, EXPLAIN visually illustrates the connection between the linear algebra representation and the graph representation. In the graph representation, all vector entries that need to be communicated to the red process correspond to those non-red vertices that are connected to a red vertex. In this example, this condition is satisfied for vertices 1, 3, 6, 7, 8, 10 which correspond

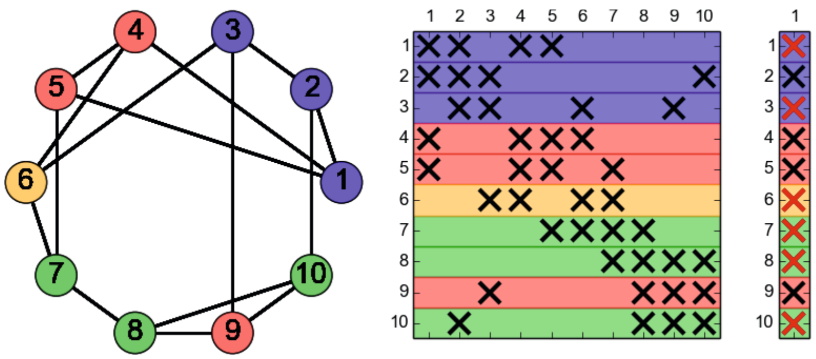


**Fig. 3.** All vector entries $x_i$ to be communicated to the red process are drawn in red (Color figure online).

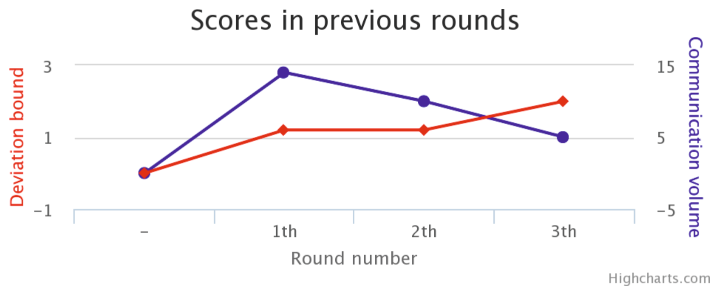


**Fig. 4.** The communication volume and the deviation bound versus various rounds (Color figure online).

to the vector entries $x_1$, $x_3$, $x_6$, $x_7$, $x_8$, $x_{10}$ in the matrix representation that are drawn in red.

When a round is completed it is also instructive to focus on the quality of the data distribution $P$. Recall that the graph partitioning problem aims at minimizing the cut size of $P$ while balancing the computational load evenly among the processes. To asses these two quantities, the module introduces the score diagram. An example of a score diagram is depicted in Fig. 4. For each round, this diagram shows the cut size defined by (4) using the label "communication volume." As mentioned in the previous section, the cut size in this undirected graph model is not an exact representation of the communication volume. However, it often captures the nature of the communication volume quite well. Therefore, this graph model uses the cut size as a measure of the communication volume. In that score diagram, the student can check his or her attempt to minimize the communication volume over a number of rounds.

The parameter $\varepsilon$ introduced in (5) is used to quantify the degree of imbalance allowed in a data distribution. If $\varepsilon = 0$ all processes are assigned exactly $|V|/p$ rows of $A$, meaning that no imbalance is allowed at all. When increasing $\varepsilon$ the load balancing condition (5) is relaxed. The larger $\varepsilon$ is chosen, the larger is the allowed imbalance. Thus, in some way, $\varepsilon$ quantifies the deviation from a perfect load balance. An equivalent form of (5) is given by

$$\frac{p}{|V|} \max_{1 \leq i \leq p} |V_i| - 1 \leq \varepsilon, \tag{6}$$

which can be interpreted as follows. Suppose that you are not looking for an $\varepsilon$-balanced partition $P$ for a given $\varepsilon$, but rather turn this procedure around and ask: "Given a partition $P$, how large need $\varepsilon$ at least be so that this partition is $\varepsilon$-balanced?" Then the left-hand side of the inequality (6) which we call *deviation bound* gives an answer to that question. The extreme cases for the deviation bound are given by 0 if the distribution is perfectly balanced and $p - 1$ if there is one process that gets all the data. The score diagram shows the value of the deviation bound for each round. A low deviation bound indicates a partition that balances the computational load evenly, whereas a large deviation bound represents a large imbalance of the load. The score diagram helps the student to evaluate the quality of a single data distribution and to compare it with distributions obtained in previous rounds. This feedback to the student is designed in the spirit of computer games, where a score has only a low immediate relevance to the current game. However, the idea is to achieve a "high score" and try to motivate the player to beat that score in subsequent rounds, thus offering an extra challenge. For this educational module, a "high score" would consist of a low communication value together with a low deviation bound.

Finally, we mention that EXPLAIN is designed to be extended for problems arising in the general area of combinatorial scientific computing, including but not limited to parallel computing. Previous modules are available on Cholesky factorization [18], nested dissection [7], column compression [9], and bidirectional compression [8].

## 6 Related Work

The idea to incrementally integrate topics from parallel computing into existing courses is not new. Brown and Schoop [3] introduce a set of flexible teaching modules designed to be used in many potential courses. Our approach is similar in that the syllabus of an existing course requires only minimal changes and that we try to reduce the effort needed by an instructor to deploy the module in a course. A collection of their modules is available at http://csinparallel.org. These modules cover various areas, but have a focus on programming. Like our module, one of the modules of this collection is related to data structures. Another approach is described in [10]. It integrates modules on parallel programming into undergraduate courses in programming languages at the two liberal arts colleges Knox College and Lewis & Clark College. Here, the functional language Haskel is chosen to introduce parallel programming via two class periods of about one hour each. Adams [1] employs the shared-memory parallel programming paradigm OpenMP to introduce parallel design patterns in a data structure course. Also in an existing course on data structures, Grossman and Anderson [16] take a comprehensive approach using fork/join parallelism.

## 7 Concluding Remarks

The overall idea behind this paper is to integrate elements of parallelism into existing courses that were previously designed with a serial computing paradigm in mind. A natural approach to implement this strategy is to focus on parallel programming. We strongly believe that parallel programming is an important element of any program in computer science, computer engineering, and computational science. However, we also advocate with this article that, in parallel computing, there is more than programming. To this end, we introduce an interactive educational module that can easily be integrated into an existing course on data structures. Though this web-based module can be augmented with parallel programming exercises, its focus is on a higher level of abstraction. It shows to undergraduate students that, when going from serial to parallel computing, it is necessary to consider additional topics of fundamental quality. More precisely, the data distribution needed to balance computational work while minimizing communication is connected to graph partitioning. Thus, a problem like sparse matrix-vector multiplication which is simple on a serial computer leads to a hard combinatorial problem when computed in parallel.

**Acknowledgments.** This work is partially supported by the German Federal Ministry for the Environment, Nature Conservation, Building and Nuclear Safety (BMUB) and the German Federal Ministry for Economic Affairs and Energy (BMWi) within the project MeProRisk II, contract number 0325389F.

## References

1. Adams, J.C.: Injecting parallel computing into CS2. In: Proceedings of the 45th ACM Technical Symposium on Computer Science Education, SIGCSE 2014, pp. 277–282. ACM, New York (2014). http://doi.acm.org/10.1145/2538862.2538883
2. Bischof, C.H., Bücker, H.M., Henrichs, J., Lang, B.: Hands-on training for undergraduates in high-performance computing using Java. In: Sørevik, T., Manne, F., Moe, R., Gebremedhin, A.H. (eds.) PARA 2000. LNCS, vol. 1947, pp. 306–315. Springer, Heidelberg (2001)
3. Brown, R., Shoop, E.: Modules in community: injecting more parallelism into computer science curricula. In: Proceedings of the 42nd ACM Technical Symposium on Computer Science Education, SIGCSE 2011, pp. 447–452. ACM, New York (2011). http://doi.acm.org/10.1145/1953163.1953293
4. Bücker, H.M., Lang, B., Bischof, C.H.: Teaching different parallel programming paradigms using Java. In: Proceedings of the 3rd Annual Workshop on Java for High Performance Computing, Sorrento, Italy, 17 June 2001, pp. 73–81 (2001)
5. Bücker, H.M., Lang, B., Bischof, C.H.: Parallel programming in computational science: an introductory practical training course for computer science undergraduates at Aachen University. Future Gener. Comput. Syst. **19**(8), 1309–1319 (2003)
6. Bücker, H.M., Lang, B., Pflug, H.-J., Vehreschild, A.: Threads in an undergraduate course: a Java example illuminating different multithreading approaches. In: Laganá, A., Gavrilova, M.L., Kumar, V., Mun, Y., Tan, C.J.K., Gervasi, O. (eds.) ICCSA 2004. LNCS, vol. 3044, pp. 882–891. Springer, Heidelberg (2004)
7. Bücker, H.M., Rostami, M.A.: Interactively exploring the connection between nested dissection orderings for parallel Cholesky factorization and vertex separators. In: IEEE 28th International Parallel and Distributed Processing Symposium. IPDPS 2014 Workshops, Phoenix, Arizona, USA, 19–23 May 2014, pp. 1122–1129. IEEE Computer Society, Los Alamitos (2014)
8. Bücker, H.M., Rostami, M.A.: Interactively exploring the connection between bidirectional compression and star bicoloring. In: Koziel, S., Leifsson, L., Lees, M., Krzhizhanovskaya, V.V., Dongarra, J., Sloot, P.M.A. (eds.) International Conference on Computational Science, ICCS 2015 – Computational Science at the Gates of Nature, Reykjavík, Iceland, 1–3 June 2015. Elsevier (2015). Procedia Comput. Sci. **51**, 1917–1926

9. Bücker, H.M., Rostami, M.A., Lülfesmann, M.: An interactive educational module illustrating sparse matrix compression via graph coloring. In: 2013 International Conference on Interactive Collaborative Learning (ICL), Proceedings of the 16th International Conference on Interactive Collaborative Learning, Kazan, Russia, 25–27 September 2013, pp. 330–335. IEEE, Piscataway (2013)
10. Bunde, D.P., Mache, J., Drake, P.: Adding parallel Haskell to the undergraduate programming language course. J. Comput. Sci. Coll. **30**(1), 181–189 (2014). http://dl.acm.org/citation.cfm?id=2667369.2667403
11. Çatalyürek, Ü.V., Aykanat, C.: Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication. IEEE Trans. Parallel Distrib. Syst. **10**(7), 673–693 (1999)
12. Çatalyürek, Ü.V., Uçar, B., Aykanat, C.: Hypergraph partitioning. In: Padua, D. (ed.) Encyclopedia of Parallel Computing, pp. 871–881. Springer, New York (2011)
13. Duff, I.S., Erisman, A.M., Reid, J.K.: Direct Methods for Sparse Matrices. Clarendon Press, Oxford (1986)
14. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman, San Francisco (1979)
15. George, A., Liu, J.W.H.: Computer Solution of Large Sparse Positive Definite Systems. Prentice-Hall, Englewood Cliffs (1981)
16. Grossman, D., Anderson, R.E.: Introducing parallelism and concurrency in the data structures course. In: Proceedings of the 43rd ACM Technical Symposium on Computer Science Education, SIGCSE 2012, pp. 505–510. ACM, New York (2012). http://doi.acm.org/10.1145/2157136.2157285
17. Hendrickson, B., Kolda, T.G.: Graph partitioning models for parallel computing. Parallel Comput. **26**(2), 1519–1534 (2000)
18. Lülfesmann, M., Leßenich, S.R., Bücker, H.M.: Interactively exploring elimination orderings in symbolic sparse Cholesky factorization. In: International Conference on Computational Science, ICCS 2010. Elsevier (2010). Procedia Comput. Sci. **1**(1), 867–874
19. Osterby, O., Zlatev, Z.: Direct Methods for Sparse Matrices. Springer, New York (1983)
20. Pissanetzky, S.: Sparse Matrix Technology. Academic Press, New York (1984)
21. Saad, Y.: Iterative Methods for Sparse Linear Systems, Second edn. SIAM, Philadelphia (2003)
22. Uçar, B., Aykanat, C.: Revisiting hypergraph models for sparse matrix partitioning. SIAM Rev. **49**(4), 595–603 (2007). http://dx.doi.org/10.1137/060662459