Term paper

Matrix-vector multiplication

in the context of the seminar “Parallel Programming and Parallel Algorithms”

Annika Biermann

Supervisor: Prof. Dr. Herbert Kuchen
Tutor: Dipl.-Wirt.-Inform. Philipp Ciechanowicz
Department of Information Systems
Practical Computer Science in Commerce

Table of contents

1 Motivation for parallelizing matrix-vector multiplication

Matrix-vector multiplication plays a very important role in most varied domains and application areas. It constitutes the kernel operation for computations not only in computer science, but also for example in natural science, engineering, economics, and everyday computer graphics like visualizations. Hence, besides a major importance it also features a high frequency. The fact, that the computer technology has been, and still goes, more and more towards multi core processing and even the private sector is completely captured by parallel computers by now, leads to the request for parallelizing the matrix-vector multiplication. A parallel matrix-vector algorithm takes advantage of the possibilities of modern computer architectures and increases its performance and efficiency compared to the serial algorithm.

Different approaches concerning data and work partitioning among the available processing units result from the aim of parallelizing the matrix-vector multiplication. This paper will discuss the three main approaches, their following algorithms, and for each a possible implementation including a performance comparison in the end.

At first, basic principles of the matrix-vector multiplication are introduced in Chapter 2 including an illustration of the serial matrix-vector multiplication and the different possible data decomposition approaches evolving from trying to parallelize it. Chapter 3 constitutes the main part of the paper and explains the algorithms and their analysis and implementation resulting from the three different data decomposition approaches. At the end of the chapter, the presented implementations of the algorithms are benchmarked and compared to each other. Chapter 4 constitutes the conclusion of this paper.

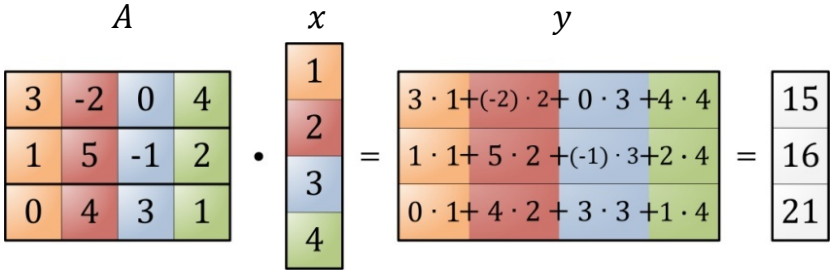
2 Basic principles of matrix-vector multiplication

2.1 Serial matrix-vector multiplication

At first, we take a look at the matrix-vector multiplication respecting its general constitution. Assuming we want to multiply a matrix A consisting of m rows and n columns with a vector x (with $m, n \in \mathbb{N}$, $A \in \mathbb{R}^{m \times n}$, $x \in \mathbb{R}^n$, holds for the following paper). In this case, vector x has to be composed of n elements. Thus, in a matrix-vector multiplication the number of matrix columns has to accord with the number of vector

elements. The result is formed by a column vector y with m elements ($y \in \mathbb{R}^m$). Therefore, the matrix-vector multiplication can formally be summarized as $Ax = y$ (nomenclature according to [Te05, pp. 67, 329]).

The procedure of a matrix-vector multiplication can easily be implemented by a sequence of scalar product calculations of vector x and the row vectors of matrix A [Qu03, pp. 179 et seq.]. Figure 1 illustrates the approach by means of an example with $m = 3$ and $n = 4$. Each entry y_i in row i of vector y thereby constitutes the result of the scalar product calculation of the i-th row vector of matrix A and vector x ($i \in \mathbb{N}, i \leq m$).



Source: [Qu03, p. 180]

Figure 1: Matrix-vector multiplication procedure example.

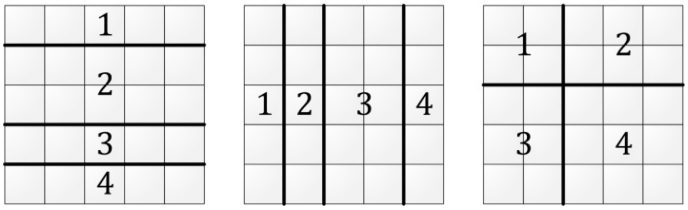
Concerning the complexity of the serial matrix-vector multiplication, we look at the complexity requirements of scalar product operations. The calculation of the scalar product of two vectors of length n implies n multiplication and $n - 1$ summation operations, which leads to a complexity of $\Theta(n)$. The matrix-vector multiplication of matrix A and column vector x comprises m scalar product computations of two vectors with n elements. Accordingly, its complexity is $\Theta(m \cdot n)$, which is confirmed by [Qu03, p. 180]. Therefore, if $m = n$, the serial algorithm has a complexity of $\Theta(n^2)$.

2.2 Data decomposition in parallel matrix-vector multiplication

When it comes to the parallelization of the matrix-vector multiplication, the single steps of calculation have to be distributed among the available parallel processing units [SK02, pp. 150 et seqq.] or processes [Qu03, pp. 180 et seqq.]. Depending on how the matrix and vector elements are partitioned over the processes, different parallel algorithms are the consequence [Qu03, p. 180], [Ll96, p. 102]. Thus, LEWIS and VAN DE GEIJN state in [LG93, p.487], that "the key decision is data distribution", because it determines the parallel algorithm and therewith "the number of necessary computation and communication operations", as SCHMOLLINGER and KAUFMANN explain [SK02,

p. 150]. The more balanced the elements are thereby distributed among the available processes, the better is the performance and efficiency of the resulting algorithm [SK02, p. 148], [HW94, p. 1203]. In other words, the matrix should be divided into element groups of equal size.

Trying to distribute an $m \times n$ matrix to p processes ($p \in \mathbb{N}$), three simple possibilities become obvious: *row-wise*, *column-wise* and *square block-wise* decomposition. Figure 2 shows the three different rudiments with the help of an example by a 5×5 matrix that is decomposed among four processes.



Source: [Qu03, p. 180]

Figure 2: Examples for different matrix decompositions.

In the row-wise case, considering the desired balance capacity, the matrix is separated into segments of $\left\lfloor \frac{m}{p} \right\rfloor$ respectively $\left\lceil \frac{m}{p} \right\rceil$ neighbouring rows. In the example in Figure 2, this means that each process gets $\left\lfloor \frac{m}{p} \right\rfloor = \left\lfloor \frac{5}{4} \right\rfloor = 1$ or $\left\lceil \frac{m}{p} \right\rceil = \left\lceil \frac{5}{4} \right\rceil = 2$ rows of the matrix.

Analogously to the row-wise case, in the column-wise decomposition, each process is in charge of $\left\lfloor \frac{n}{p} \right\rfloor$ or $\left\lceil \frac{n}{p} \right\rceil$ adjoining columns.

For the square block-wise decomposition, the matrix is partitioned into square blocks dimensioned at least $\left\lfloor \frac{m}{\lceil \sqrt{p} \rceil} \right\rfloor \times \left\lfloor \frac{n}{\lfloor \sqrt{p} \rfloor} \right\rfloor$ or $\left\lfloor \frac{m}{\lfloor \sqrt{p} \rfloor} \right\rfloor \times \left\lfloor \frac{n}{\lceil \sqrt{p} \rceil} \right\rfloor$ and at most $\left\lceil \frac{m}{\lceil \sqrt{p} \rceil} \right\rceil \times \left\lceil \frac{n}{\lfloor \sqrt{p} \rfloor} \right\rceil$ or $\left\lceil \frac{m}{\lfloor \sqrt{p} \rfloor} \right\rceil \times \left\lceil \frac{n}{\lceil \sqrt{p} \rceil} \right\rceil$. Therefore, in the above-mentioned example, the square blocks' sizes range from $\left\lfloor \frac{m}{\lceil \sqrt{p} \rceil} \right\rfloor \times \left\lfloor \frac{n}{\lfloor \sqrt{p} \rfloor} \right\rfloor = \left\lfloor \frac{5}{\lceil \sqrt{4} \rceil} \right\rfloor \times \left\lfloor \frac{5}{\lfloor \sqrt{4} \rfloor} \right\rfloor = 2 \times 2$ to $\left\lceil \frac{m}{\lceil \sqrt{p} \rceil} \right\rceil \times \left\lceil \frac{n}{\lfloor \sqrt{p} \rfloor} \right\rceil = \left\lceil \frac{5}{\lceil \sqrt{4} \rceil} \right\rceil \times \left\lceil \frac{5}{\lfloor \sqrt{4} \rfloor} \right\rceil = 3 \times 3$.

The concrete number of allocated rows, columns, or the concrete block size per process depends on the algorithm implementation. In Figure 2, the allocations are arbitrarily chosen.

There are two different ways to allocate the two vectors x and y to the processes. On the one hand, they could be completely copied to every process. On the other hand, they

could be partitioned among the processes. In the latter case, each process would handle $\lfloor \frac{n}{p} \rfloor$ or $\lceil \frac{n}{p} \rceil$ neighbouring vector elements. As QUINN shows in [Qu03, p. 181], "the storage requirements are in the same complexity class" irrespective of the way the vectors are distributed. But he does not say anything about the communication complexity.

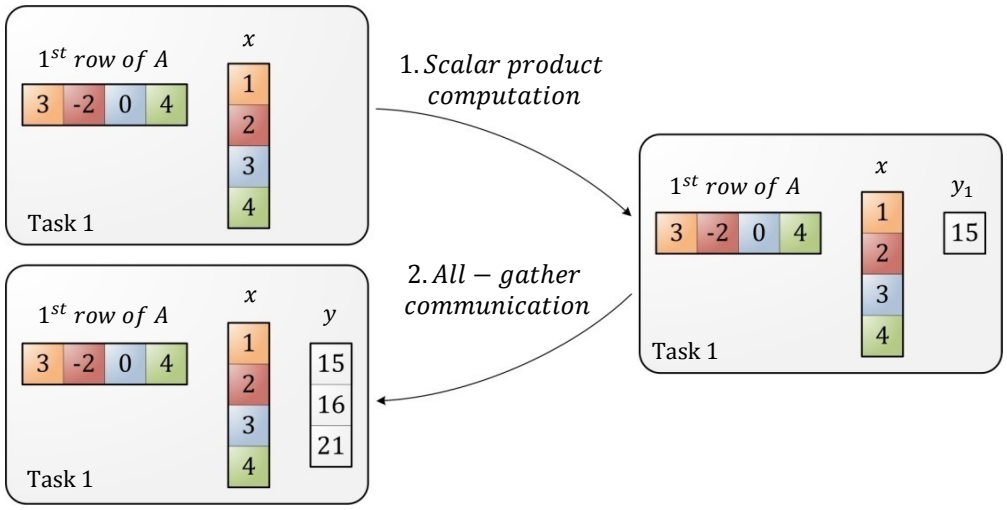
In the following, the three presented matrix decomposition options and the consequential parallel algorithms will be examined. The respective vector decomposition will be chosen depending on the minimal requirements of vector elements for each process to keep the communication between the processes minimal. Hence, for the row-wise matrix decomposition, the vectors will be entirely replicated and the column-wise and square block-wise matrix decomposition get by with partial vector blocks on each process.

3 Parallel matrix-vector multiplication algorithms and implementations

3.1 Row-wise matrix decomposition

3.1.1 Algorithm

The first presented parallel matrix-vector multiplication algorithm is based on row-wise matrix decomposition and replicated vectors. This means, that every process gets one or more contiguous rows of the matrix A and a copy of the complete vector x. Assuming that every single row vector is part of a distinct task, i.e. scalar product computation, a process can incorporate several tasks. For every task, the involved process has to calculate the scalar product of the concerned row vector and vector x. The result of each scalar product computation is an element of the result vector y. Thus, task i possesses the i-th row of matrix A and a replication of vector x. After the scalar multiplication, it has element y_i of the result vector y [Qu03, pp. 181 et seq.]. And hence, every process owns an element block of y. Every task has to communicate its element y_i to all other tasks so that each process and task get an entire copy of the result vector y. This communication step is called *all-gather* and will be specified in the implementation section. As soon as every task has a replication of the complete result vector, the algorithm terminates.



Source: [Qu03, p. 182]

Figure 3: Example of main row-wise decomposition algorithm steps.

The main steps of the described algorithm are illustrated in Figure 3 by means of the example of task 1 according to the matrix and vector values of Figure 1. At first, task 1 has the first row of A and a copy of x. After the computation of the scalar product it also has the first element of the result vector. Due to the all-gather communication step, all tasks have a copy of the complete result vector y and the algorithm terminates.

3.1.2 Analysis

To analyze the complexity of the presented algorithm, we assume that $m = n$, i.e. that matrix A is square and has n rows and n columns. According to its two main steps, we split the algorithm's complexity specification into a computational and a communicational part.

Every process has at most $\left\lceil \frac{n}{p} \right\rceil$ rows of the matrix. Hence, its maximum calculation charge is $\left\lceil \frac{n}{p} \right\rceil$ scalar multiplications. In combination with the above shown scalar product complexity of $\Theta(n)$, the computation complexity of the parallel algorithm results in $\Theta\left(n \cdot \frac{n}{p}\right) = \Theta\left(\frac{n^2}{p}\right)$ [Qu03, p. 183].

For the all-gather communication operation, QUINN shows in [Qu03, pp. 83 et seqq.] on the basis of a hypercube network, i.e. p is a power of two, that $\lceil \log p \rceil$ messages are send per process, each in the context of a data exchange step between the processes. Furthermore, the number of elements passed per exchange step increases: in exchange

step i, $2^{i-1} \cdot \frac{n}{p}$ elements are transferred. Consequently, the total number of elements communicated is $\sum_{i=1}^{\log p} 2^{i-1} \cdot \frac{n}{p} = \frac{n(p-1)}{p}$.

Accordingly, the communication complexity is $\Theta\left(\log p + \frac{n(p-1)}{p} + \frac{n}{p}\right) = \Theta(\log p + n)$. Consequently, we get a total complexity of $\Theta\left(\frac{n^2}{p} + \log p + n\right)$ for the parallel algorithm [Qu03, p. 183].

In terms of the isoefficiency of the parallel algorithm, we examine its overhead, which is only composed of the all-gather communication. For a reasonably large n, the communication complexity can be simplified to $\Theta(n)$ because the real communication time exceeds the communication latency time. Accordingly, the isoefficiency function is $n^2 \geq Cpn \Rightarrow n \geq Cp$ [Qu03, p. 183] with the constant $C = \frac{efficiency}{1 - efficiency}$, $efficiency = \frac{speedup}{p}$ and $speedup = \frac{duration\ of\ serial\ algorithm}{duration\ of\ parallel\ algorithm}$ [Qu03, pp. 160, 171].

The memory utilization function for a $n \times n$ matrix is $M(n) = n^2$, because n^2 elements have to be stored in memory. Hence, we get the scalability function $\frac{M(Cp)}{p} = \frac{C^2p^2}{p} = C^2p$ that "indicates how the amount of memory used per processor must increase as a function of p in order to maintain the same level of efficiency" [Qu03, p. 171]. As a conclusion, the considered algorithm is not highly scalable because the required memory per processor does not increase linearly with the number of processors [Qu03, p. 183].

3.1.3 Implementation

A possible implementation of the described parallel algorithm is presented by QUINN in [Qu03, p. 188] (see Listing 1 in appendix A). It concerns a program written in the programming language C using the Message Passing Interface (MPI) for inter-process communication. An introduction to MPI and more detailed information can be found in [GLS07] and are not intended to be part of this paper.

The implemented program expects two arguments: first the name of the file in which the matrix is stored, and second the filename of the document that contains the input vector.

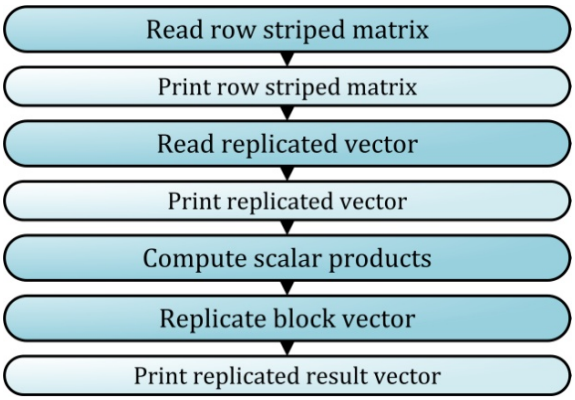


Figure 4: Operational overview of the row-wise decomposition based implementation.

An overview of the mentioned implementation is depicted in Figure 4 (for a detailed version see Figure 20 in appendix A). The light-coloured operations are printing calls that are not actually required for the implementation of the algorithm and will be ignored at this point. The most relevant operation steps are highlighted by a stronger blue background colour. They will be discussed in more detail in the following paragraphs.

Read row-striped matrix

The first important step in the implementation of the row-wise decomposition based parallel matrix-vector multiplication is the *read row-striped matrix* operation. Its source code is listed in [Qu03, pp. 495 et seq.]. The left-hand side of Figure 5 shows its constituent parts. As a result of the read row striped matrix operation, every process has got its block of matrix rows.

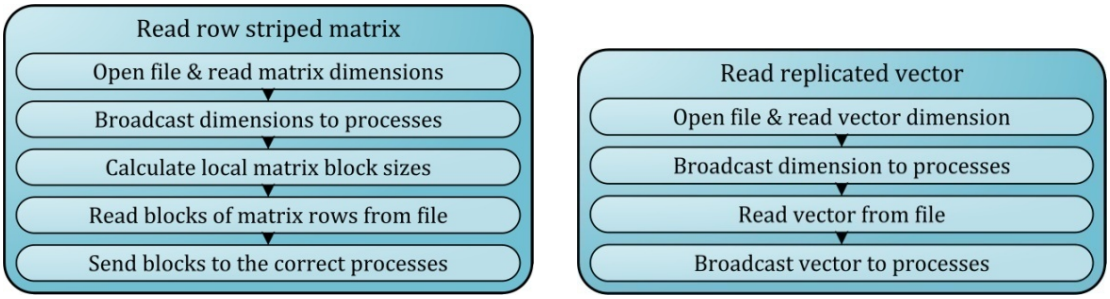


Figure 5: Read row striped matrix and read replicated vector operation steps.

Read replicated vector

All processes need to know the input vector to compute the elements of the output vector. The *read replicated vector* operation makes sure, that every process gets a copy of the vector. It works analogously to the read row striped matrix operation as illustrated in Figure 5 right hand. At the end of the read replicated vector operation, each process

possesses its required matrix block and a copy of the input vector for computing the scalar products.

Compute scalar products

In contrast to the other operations, the *compute scalar products* operation step is not a discrete method but directly implemented in the main program. For each row vector, the scalar product with the input vector is calculated. Consequently, every process holds a block of contiguous elements of the result vector after the scalar product computation is finished.

Replicate block vector

The last important step of the implementation is the *replicate block vector* operation, whose structure is depicted in Figure 6. Its main part is the *all-gather* communication step which is led by the *create mixed xfer arrays* operation.

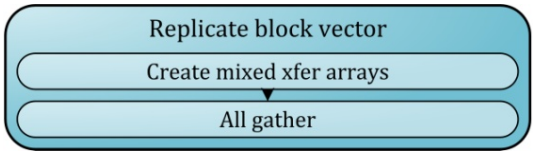
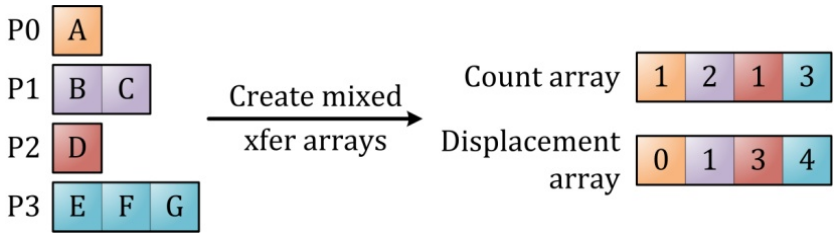


Figure 6: Replicate block vector procedure step.

The *create mixed xfer arrays* operation is a preliminary for the all-gather step. It creates two arrays which are required by all-gather. On the one hand, it creates the *count array* which contains the number of result vector elements each process has, as shown in the example of Figure 7. On the other hand, it creates the *displacement array* that consists of the starting positions of each process' element block in the result vector. In other words, it is the successively cumulated count array that starts with zero. The array indexes of both arrays comply with the processes' ranks.

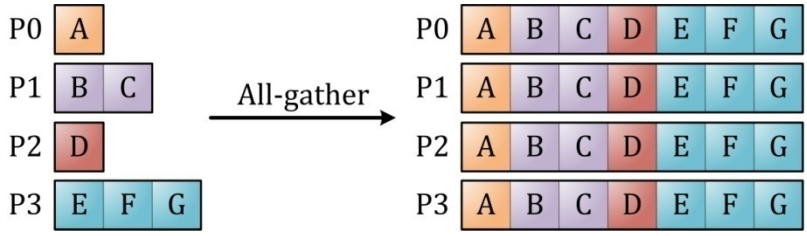


Source: [GLS07, p. 197], [Qu03, p. 184]

Figure 7: Create mixed xfer arrays example.

Based on the two created arrays, the all-gather communication operation can proceed. It implements that, at the end, every process holds a copy of the complete result vector. The effect of the all-gather operation is illustrated by means of the previous example in

Figure 8. When the operation is called, the result vector is block-distributed among the processes. After the execution of all-gather, the result vector is replicated on each process. Accordingly, the aim of the overall implementation is achieved and the program terminates.



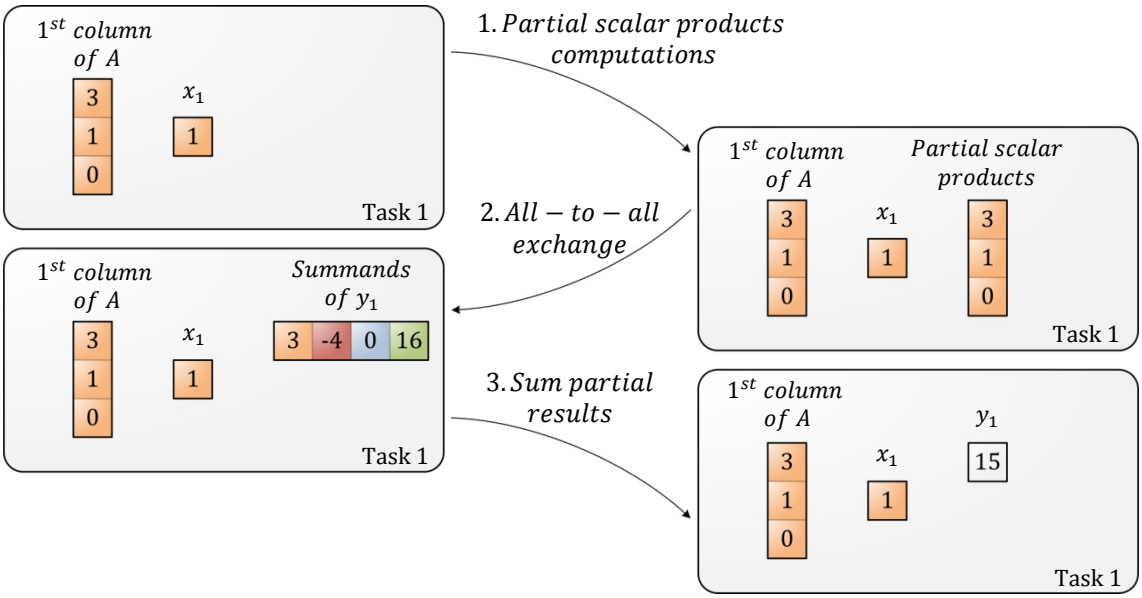
Source: [GLS07, p. 197], [Qu03, p. 184]

Figure 8: All-gather communication step example.

3.2 Column-wise matrix decomposition

3.2.1 Algorithm

The second presented parallel matrix-vector multiplication algorithm is based on column-wise matrix decomposition and block distributed vectors. Hence, every process receives one or more contiguous columns of the matrix A and a contiguous element block of the vector x. A process covers several tasks, a single task involves one column and the corresponding element of vector x. Every task multiplies the vector element with the corresponding column vector and gets an incomplete result vector including partial scalar products. Each task sends it partial scalar product results to the correct task so that every task is able to compute an element of the result vector y. Thus, at the beginning, task i possesses the i-th column of matrix A and the i-th vector element x_i. After the multiplication it has a vector of partial results of output vector y [Qu03, p. 189]. Then, an all-to-all exchange takes place, i.e. task i sends all partial results except for the i-th one to the corresponding tasks and receives the i-th partial result in each case from the other tasks. After that, task i owns all summands of the result vector element y_i and computes it by summation. Figure 9 exemplifies the main steps of the presented algorithm on the basis of the continuous example values.



Source: [Qu03, p. 190]

Figure 9: Example of main column-wise decomposition algorithm steps.

3.2.2 Analysis

Again, we assume that A is square with $m = n$ for the ascertainment of the algorithm's complexity. Each process holds at most $\left\lceil \frac{n}{p} \right\rceil$ matrix columns. Thus, in combination with the above shown scalar product complexity of $\Theta(n)$, the computation complexity of the parallel algorithm results in $\Theta\left(n \cdot \frac{n}{p}\right) = \Theta\left(\frac{n^2}{p}\right)$ [Qu03, p. 189].

For an all-to-all communication operation, $\lceil \log p \rceil$ communication steps are required, applying a hypercube communication pattern [Qu03, p. 190] as shown in [Qu03, pp. 83 et seqq.]. Per step, $\frac{n}{2}$ elements are sent and $\frac{n}{2}$ elements are received by each process. Hence, in total $\left(\frac{n}{2} + \frac{n}{2}\right) \cdot \lceil \log p \rceil = n \cdot \lceil \log p \rceil$ elements are transmitted. Accordingly, the communication complexity is $\Theta(n \cdot \log p)$.

However, there is another way of doing an all-to-all communication. In this case, $p - 1$ messages are communicated because every process sends a message to all other $p - 1$ processes. Every process thereby only gets its supposed elements. Hence, each process transmits at most n elements. Accordingly, the communication complexity would be $\Theta(p + n)$ as already presented in [Qu03, pp. 190 et seq.].

Consequently, we get a total complexity of $\Theta\left(\frac{n^2}{p} + n \cdot \log p\right)$ or $\Theta\left(\frac{n^2}{p} + p + n\right)$ for the parallel algorithm [Qu03, pp. 190 et seq.].

Regarding the isoefficiency of the parallel algorithm, we consider its overhead, which is only composed of the all-to-all communication step. In the case of the second all-to-all communication performance for a reasonably large n, the communication complexity can be simplified to $\Theta(n)$ as well because the real transmission time is higher than the latency time. Thus, the isoefficiency function is $n^2 \geq Cpn \Rightarrow n \geq Cp$ which equals the isoefficiency function of the row-wise decomposition based algorithm. Hence, this algorithm is not highly scalable either [Qu03, p. 191].

3.2.3 Implementation

QUINN offers an implementation of the previous algorithm in [Qu03, pp. 197 et seq.], which is written in C and uses the MPI library (see Listing 2 in appendix A). Figure 10 represents an overview of the implemented program steps. A more detailed demonstration can be found in Figure 21 in appendix A.

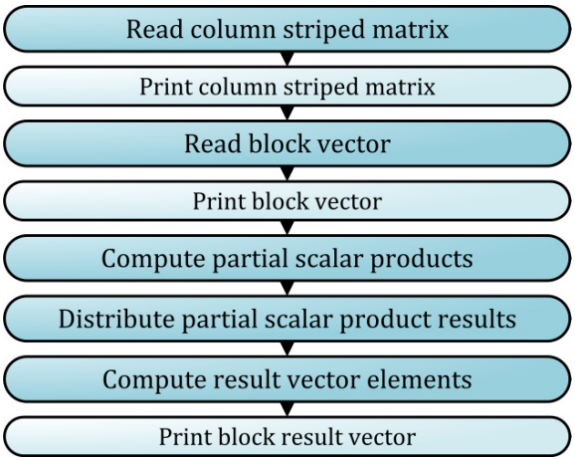


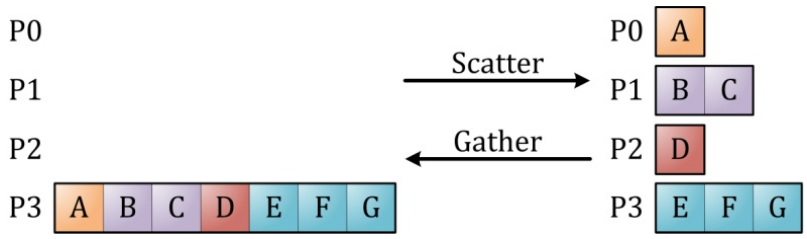
Figure 10: Operational overview of the column-wise decomposition based implementation.

The program expects the same two arguments as the first one: the names of the files in which the matrix and vector are stored. The matrix file stores the matrix row-wise which becomes relevant when the matrix has to be decomposed in a column-wise manner.

Read column striped matrix

The first important step in the implementation of the column-wise decomposition based parallel matrix-vector multiplication is the *read column striped matrix* operation, whose source code is listed in [Qu03, pp. 493 et seq.]. Its components are summarized in Figure 21 in Appendix A. The matrix rows thereby have to be scattered among the processes. The *scatter* communication step is exemplary demonstrated in Figure 11. In

the end, after all rows have been scattered, every process holds a contiguous block of matrix columns.



Source: [GLS07, p. 197], [Qu03, pp. 192, 194]

Figure 11: Scatter and gather communication step example.

Print column striped matrix

We will not go into details about the printing steps of the implementation. However, at this point, we will briefly point out that for printing a column decomposed matrix row-wise, an inverse communication step of scatter is needed, namely the *gather* communication step as illustrated in Figure 11.

Read block vector

The *read block vector* operation reads the vector from a file and sends the needed vector blocks to the corresponding processes. The detailed procedure is equivalent to the previous read operations and is pictured in Figure 21 in Appendix A.

Compute partial scalar products

The *compute partial scalar products* operation step is not a discrete method but directly implemented in the main program. It consists of the partial scalar product calculation of each column element with the corresponding input vector element. If a process has more than one column and input vector element, the partial scalar product results of the same matrix row are summed to partial sums. Consequently, after this operation, every process holds a vector of partial sums of the result vector.

Distribute partial scalar product results

The next step is the *distribute partial scalar product results* operation. It is directly implemented in the main program and can be separated into four sub-items as Figure 12 shows. However, the main part is the last one, the all-to-all communication. The preceding steps only conduce to prearrangements for the communication step. The aim of the operation is, that the partial sums of each result vector element, which are scattered among the processes, get together at the same process.

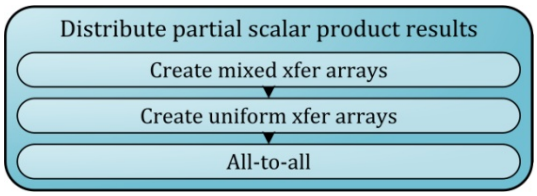
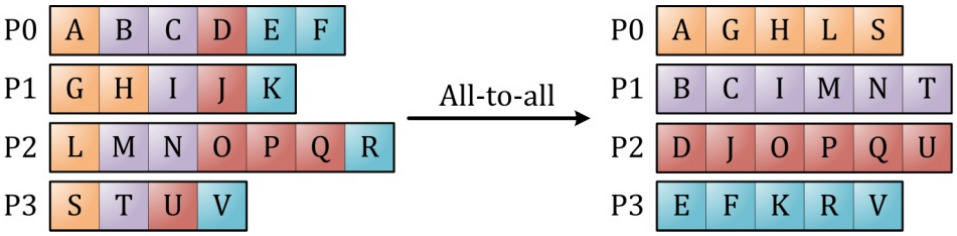


Figure 12: Distribute partial scalar product results operation step.

For the realization of all-to-all, four arrays have to be created by each process. The first two arrays result from the *create mixed xfer arrays* operation (see page 10 and Figure 7). The resulting count and displacement arrays refer to the outgoing elements which a process sends to the other processes. The remaining two arrays are created by the *create uniform xfer arrays* operation. It acts analogously to *create mixed xfer arrays*, the only disparity is, that the *create uniform xfer arrays* operation assumes, that every process gets the same number of elements from each process. According to this, its resulting arrays are as well a count and a displacement array which, in this case, concern the incoming elements of each process, i.e. the elements, that each process receives from the other processes. On the basis of these four arrays, the all-to-all communication operation will be able to communicate the partial sums from every process to the correct processes. So first, every process has to allocate the required memory for its incoming summands. After that, the all-to-all communication takes place. Its impact is exemplary described in Figure 13. After its execution, all partial sums that belong to the same result vector element are at the same process.



Source: [GLS07, p. 197], [Qu03, p. 196]

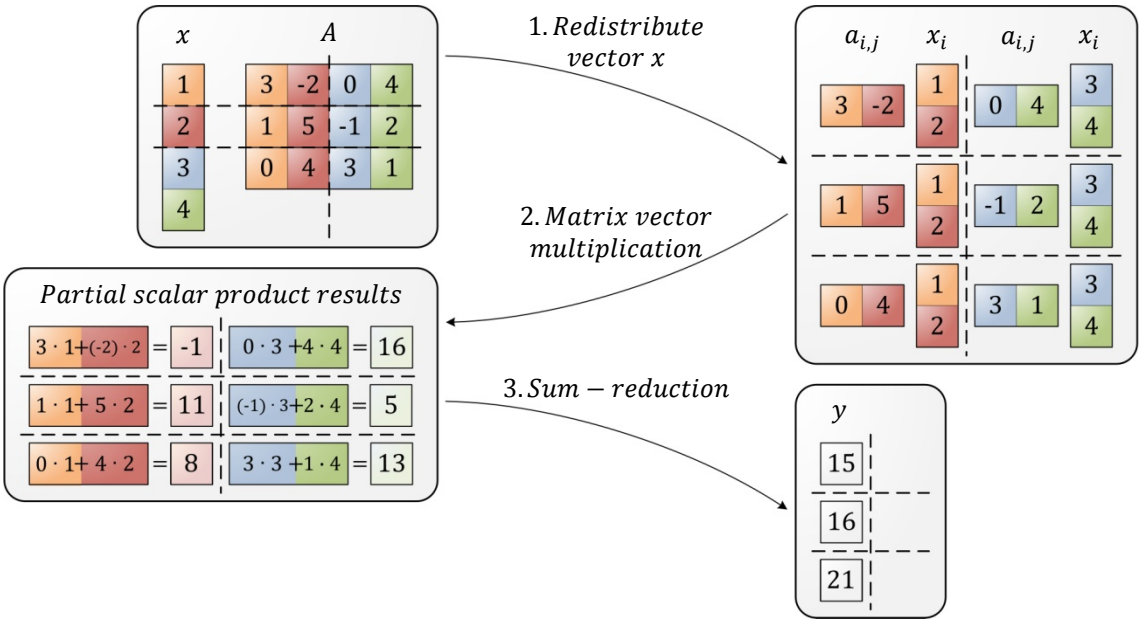
Figure 13: All-to-all communication step example.

Compute result vector elements

In the final step, the partial sums at each process can be summed up to the elements of result vector y. Every process computes the summation of the partial sums and summands. In the end, every process holds a contiguous element block of the result vector y.

3.3 Square block-wise matrix decomposition

3.3.1 Algorithm



Source: [Qu03, p. 200]

Figure 14: Example of main square block-wise decomposition algorithm steps ($p = 6$).

The basis of the third and last introduced parallel matrix-vector multiplication algorithm are a square block-wise matrix decomposition and block distributed vectors. For demonstrative and descriptive reasons, we imagine that the processes form a two-dimensional grid as shown in Figure 14. Every process constitutes one spacing of the grid. Hence, p has to be even. At the beginning, each process receives a contiguous block of matrix A and the processes of the first column of the imaginary grid also get a block of vector x (as illustrated in the first state of Figure 14). As a first step, vector x has to be redistributed among the processes, such that every process receives the required vector element x_j for each possessing matrix element $a_{i,j}$ (second state in Figure 14). After the successful redistribution, every process computes the partial scalar product results of its elements. Thus, each process multiplies its $a_{i,j}$ with x_j and sums the results of the same matrix row, i.e. with the same i. In the last step, the summands of the scalar products that are distributed among the processes of the same grid row are summed up and result each in an element y_i of the result vector [Qu03, p. 200], [HLP95, pp. 5 et seq.]. At the end of the algorithm, result vector y is block distributed among the processes of the first process grid column as shown in Figure 14 in the last state.

3.3.2 Analysis

For determining the complexity of the third algorithm, we also assume $m = n$ and additionally that p is square. Hence, every process has a matrix block with dimensions of at most $\left\lceil \frac{n}{\sqrt{p}} \right\rceil \times \left\lceil \frac{n}{\sqrt{p}} \right\rceil$. Thus, the computation complexity of the parallel algorithm results in $\Theta\left(\frac{n}{\sqrt{p}} \cdot \frac{n}{\sqrt{p}}\right) = \Theta\left(\frac{n^2}{p}\right)$ [Qu03, p. 202].

For redistributing the input vector in the beginning, every first grid column process sends its block, which consists of $\frac{n}{\sqrt{p}}$ elements, to the corresponding first grid row process, consequently with time complexity $\Theta\left(\frac{n}{\sqrt{p}}\right)$. The subsequent broadcast per grid column needs time $\Theta\left(\log\sqrt{p}\left(\frac{n}{\sqrt{p}}\right)\right) = \Theta\left(\frac{n \log p}{\sqrt{p}}\right)$. The final sum-reduction communication step per grid row possesses time complexity $\Theta\left(\log\sqrt{p}\left(\frac{n}{\sqrt{p}}\right)\right) = \Theta\left(\frac{n \log p}{\sqrt{p}}\right)$. Consequently, we get an overall complexity of $\Theta\left(\frac{n^2}{p} + \frac{n \log p}{\sqrt{p}}\right)$ for the third parallel algorithm [Qu03, p. 202].

The overhead of the parallel algorithm is constituted by its communication complexity for each processor, hence the communication complexity times p: $overhead = \frac{n \log p}{\sqrt{p}} \cdot p = \sqrt{p} \cdot \log p$. Thus, the isoefficiency function is $n^2 \geq Cn\sqrt{p} \log p \Rightarrow n \geq C\sqrt{p} \log p$ and the scalability function is $\frac{M(C\sqrt{p} \log p)}{p} = \frac{C^2 p \log^2 p}{p} = C^2 \log^2 p$. As a result, the third algorithm is more scalable than the first two presented algorithms [Qu03, p. 202].

3.3.3 Implementation

A complete implementation of the third presented algorithm is not given in [Qu03], as it is subject of an exercise. However, the main procedure steps are given, on whose basis a possible implementation has been developed (see listing in appendix A). An overview of the program steps are illustrated in Figure 15, a more detailed demonstration can be found in Figure 22 in appendix A. The program expects the known two filename arguments.

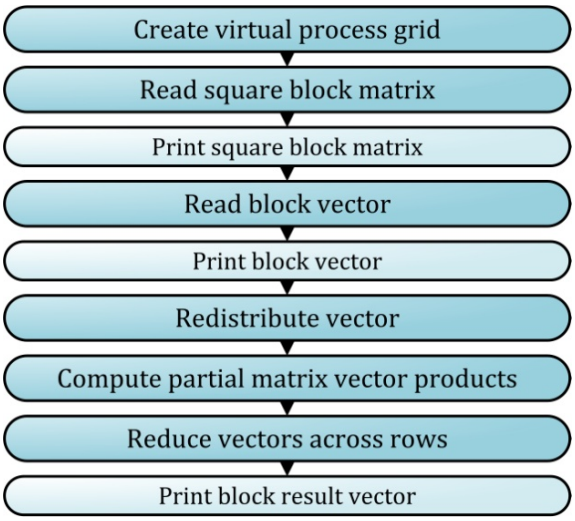


Figure 15: Operational overview of the square block-wise decomposition based implementation.

Create virtual process grid

The implementation of the square block-wise decomposed based parallel matrix-vector multiplication starts with function calls that can be summarized to the *create virtual process grid* operation step. First, based on the number of processes, the dimensions of the virtual grid are calculated. After that, the virtual grid is created in terms of a grid communicator, which enables the communication between processes with a Cartesian grid topology [Qu03, pp. 202 et seq.]. This special communicator is used for the following operation steps.

Read square block matrix

The *read square block matrix* operation components are shown in Figure 22 in Appendix A. Its source code is listed in [Qu03, pp. 490 et seqq.]. The proceeding is analogously to the reading operations discussed before. Particular at this point is, that the matrix is read row-wise from the file into a buffer. From the buffer, each row gets passed to the correct process grid row and is scattered among the processes of the current grid row. At the end, every process holds a contiguous block of elements of the matrix.

Read block vector

The *read block vector* operation step has already been explained before. However, in this implementation, the input vector gets only block distributed among the processes of the first grid column. Hence, the vector has to be redistributed among the processes, so that every process gets its required block of the vector.

Redistribute vector

The *redistribute vector* operation step correctly redistributes the vector blocks from the processes of the first grid column to all processes. In the following, we assume that p is square. If this is not the case, the redistribute vector procedure has a differing design as explicated in [Qu03, p. 201]. The main sub procedures for a square p are pictured in Figure 17 left-hand and Figure 16 with the help of an example. At first, each process of the first grid column at grid position $(i, 0)$ sends its local vector block to the corresponding diagonal process at position $(0, i)$. After that, every process in the first grid row broadcasts its received vector block to the other processes of the same grid column [Qu03, p. 201]. Finally, every process owns the fitting vector block for its matrix block.

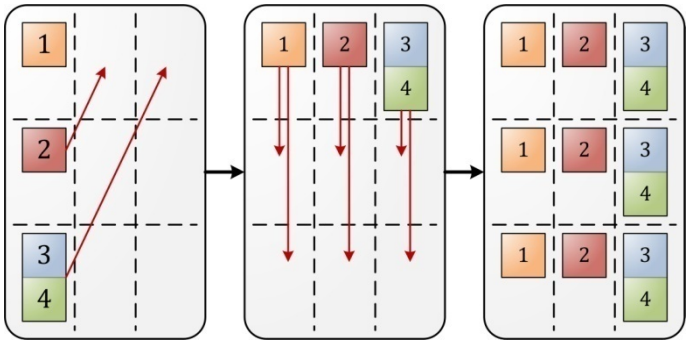


Figure 16: Redistribute vector procedure steps example.

Compute partial matrix-vector products

As a next step, every process computes the matrix-vector product of its local matrix and vector block. The result in each case is a vector block composed of partial sums of the overall result vector.

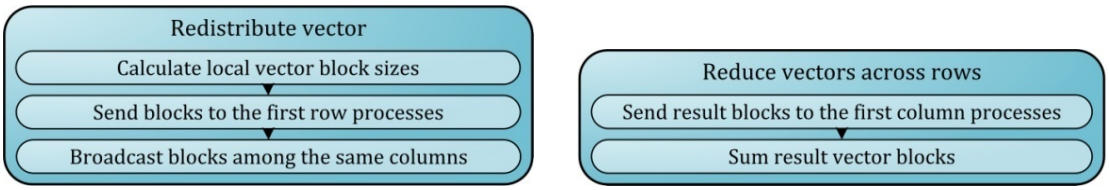


Figure 17: Redistribute vector and reduce vectors across rows operation steps.

Reduce vectors across rows

To get the elements of the result vector y, the partial result vector blocks of each process grid row have to be accumulated. To this end, every process sends its result block to the first process of its grid row (see Figure 17 right-hand). The processes in the first grid column receive all result blocks and add them to the blocks of the result

vector. In conclusion, the result vector is block distributed among the processes of the first grid column.

3.4 Comparison

	Row-wise	Column-wise	Square block-wise
Time complexity	$\Theta\left(\frac{n^2}{p} + \log p + n\right)$	$\Theta\left(\frac{n^2}{p} + p + n\right)$	$\Theta\left(\frac{n^2}{p} + \frac{n \cdot \log p}{\sqrt{p}}\right)$
Scalability function	$C^2 \cdot p$	$C^2 \cdot p$	$C^2 \cdot \log^2 p$
	⇒ not highly scalable	⇒ not highly scalable	⇒ more scalable than the others

Table 1: Analysis comparison of the three algorithms

A comparison of the three presented algorithms concerning their time complexity and scalability is clearly presented in Table 1. As a conclusion, the first two algorithms behave nearly in the same way. They are not highly scalable. In contrast, the third, square block-wise decomposition based algorithm is more scalable and hence more efficient for large data input.

3.5 Benchmarking

3.5.1 Results in the literature

	Row-wise		**Column-wise**		**Square block-wise**	
Processors	Duration	Speedup	Duration	Speedup	Duration	Speedup
1	0.0634	1.00	0.0638	1.00	0.0634	1.00
4	0.0178	3.56	0.0175	3.62	0.0174	3.64
16	0.0072	8.79	0.0076	8.33	0.0062	10.21

Table 2: Benchmarking results in [Qu03, pp. 189, 199, 209] with $m = n = 1{,}000$.

The three presented algorithms have been benchmarked by QUINN in [Qu03]. The programs have been executed on a commodity cluster of 450 MHz Pentium II processors that were connected by fast Ethernet. Selective results for $p = 1, 4, 16$ and for a square input matrix with $1{,}000$ rows and columns are listed in Table 2 for the row-wise, column-wise and square block-wise decomposition based parallel programs. The duration is computed by the average time of 100 executions of the program.

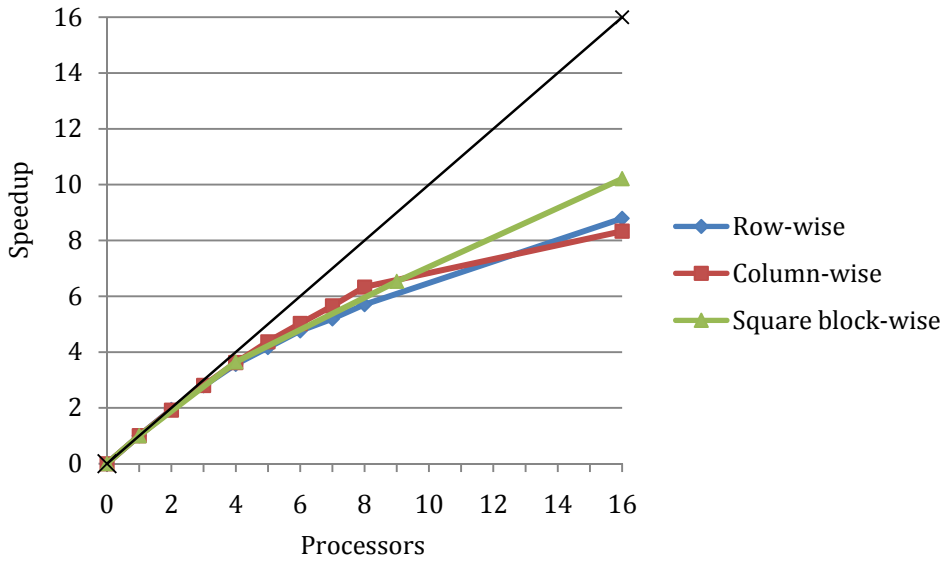


Figure 18: Speedup results for the three algorithms in [Qu03, p. 210].

The benchmark results of the three different programs are compared in Figure 18 based on their speedup $\left(= \frac{Duration\ of\ serial\ algorithm}{Duration\ of\ parallel\ algorithm}\right)$, which equals p in the optimal case. It becomes obvious, that all three programs have an equivalent speedup up to nine processors. However, from ten processors on the third parallel program, that is based on square block-wise matrix decomposition, yields a better performance than the other two programs. This can be explained by the fact, that in the first two programs $\Theta(n)$ elements of the input and output vector are transmitted by each process whereas only $\Theta\left(\frac{n}{\sqrt{p}}\right)$ elements per process are passed in the third algorithm. However, the number of transmitted messages is identical, only the amount of passed vector elements differs. As a result, the square block-wise decomposition based parallel program performs better than the other two algorithms preconditioned that the number of processors is large enough [Qu03, p. 209].

3.5.2 Self experienced results

The three programs have also been benchmarked on a modern computer cluster composed of Dual Quad Core Twin servers. Each server, i.e. compute node, consists of two 2.1 GHz Quad Core AMD Opteron CPUs and possesses eight processors.

The implementations have been run on $p = 1, 2, 4, 8, 9, 16$ processors of at most two different compute nodes. Table 3 shows a selection of the self experienced results. They are average values of 50 runs of the parallel programs computing the matrix-vector multiplication of a $10{,}000 \times 10{,}000$ matrix and a vector with 10,000 elements. For all

50 runs, the input data, which consists of matrix elements $a_{ij} = 1$ and vector elements $x_i = i$ for all i and j with $0 \leq i \leq m$ and $0 \leq j \leq n$, was the same.

	Row-wise		Column-wise		Square block-wise	
Processors	Duration	Speedup	Duration	Speedup	Duration	Speedup
1	0.3007	1.00	0.2900	1.00	0.2927	1.00
4	0.1134	2.65	0.1149	2.52	0.1134	2.58
16	0.0302	9.96	0.0406	7.14	0.0297	9.85

Table 3: Self experienced benchmarking results with m = n = 10,000.

The obtained results do not totally comply with the results presented by QUINN. However, they are very similar as the speedup illustration in Figure 19 in comparison to Figure 18 demonstrates. The main difference to QUINN's results is, that the square block-wise decomposition based implementation does not dominate the speedup results for large numbers of processors. The speedup of the row-wise decomposition based program keeps up with it, even for 16 processors. In contrast, the performance of the column-wise based algorithm is clearly the worst. The fact, that the row-wise decomposition based program performs as well as the square block-wise decomposition based program, could possibly be explained by the case that $p = 16$ is not large enough anymore for modern computer clusters and that the latter program would perform better for larger numbers of processors.

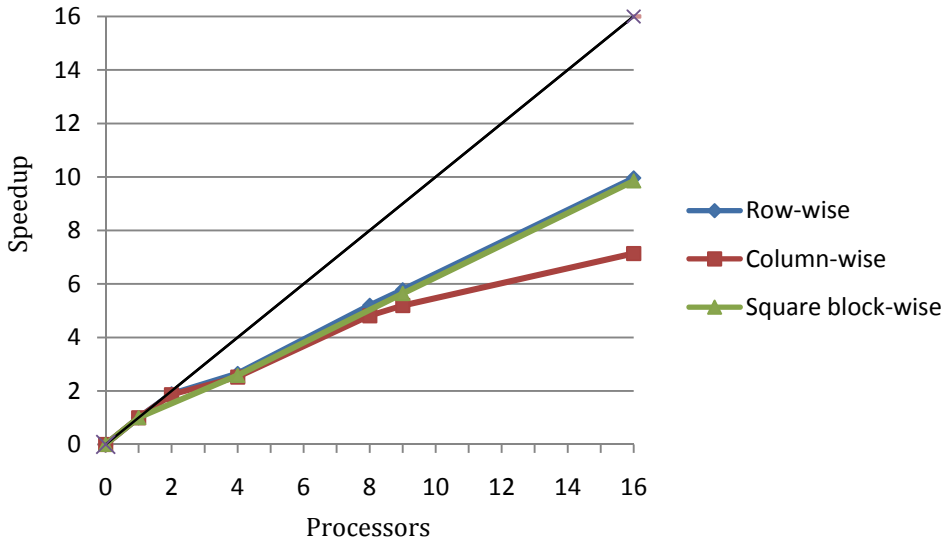


Figure 19: Self experienced speedup results for the three algorithms.

4 Conclusion

Matrix-vector multiplication has a major importance in a variety of application areas including not only science but also every day environment. The development of parallel computer technologies and its currency in the private sector ask for a parallelization of the matrix-vector multiplication. In this context, the three presented different data decomposition approaches *row-wise*, *column-wise* and *square block-wise* matrix decomposition emerge. We have discussed the resulting algorithms based on each approach including its constitution, an analysis of its time complexity and scalability, and a possible implementation in C using the MPI library. The analysis of all three algorithms and their comparison have shown that the square block-wise decomposition based algorithm is more scalable than the two other ones. Hence, its performance for large problem sizes and large numbers of used processors is dominating.

The benchmarking results in the literature have confirmed the analysis results. However, the presented benchmarking is more than six years old and took place on long outdated hardware. Therefore, we have benchmarked the developed programs on a modern computer cluster. The results were not totally equal to the older ones. Surprisingly, the row-wise decomposition based program performed as well as the as best predicted square block-wise decomposition based program. Possible explanations could be that the number of processors chosen ($p = 16$) is not adequately large any more. However, this is only an attempt of explanation without any functional foundation. Consequently, the self-experienced benchmarking results offer a basis for further analysis and research.

This paper deals with the parallelization of matrix-vector multiplication on a very basic level without looking at the specific today's implementations and designs of usage. Furthermore, the presented algorithms only consider problems of dense matrices. However, the sparse matrix case is more interesting for research and performance improvements of parallel matrix-vector multiplication.

A Implementation listings and overviews

```
/*
 *   Matrix-vector multiplication, Version 1
 *
 *   This program multiplies a matrix and a vector input from
 *   separate files. The result vector is printed to standard
 *   output.
 *
 *   Data distribution of matrix: rowwise block striped
 *   Data distribution of vector: replicated
 *
 *   Programmed by Michael J. Quinn
 *
 *   Last modification: 9 September 2002
 */

#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"
#include "MyMPI.c"

/* Change these two definitions when the matrix and vector
   element types change */

typedef double dtype;
#define mpitype MPI_DOUBLE

int main(int argc, char *argv[]) {
   dtype **a;        /* First factor, a matrix */
   dtype *b;         /* Second factor, a vector */
   dtype *c_block;   /* Partial product vector */
   dtype *c;         /* Replicated product vector */
   double max_seconds;
   double seconds;   /* Elapsed time for matrix-vector multiply */
   dtype *storage;   /* Matrix elements stored here */
   int    i, j;      /* Loop indices */
   int    id;        /* Process ID number */
   int    m;         /* Rows in matrix */
   int    n;         /* Columns in matrix */
   int    nprime;    /* Elements in vector */
   int    p;         /* Number of processes */
   int    rows;      /* Number of rows on this process */

   MPI_Init(&argc, &argv);
   MPI_Comm_rank(MPI_COMM_WORLD, &id);
   MPI_Comm_size(MPI_COMM_WORLD, &p);

   read_row_striped_matrix(argv[1], (void *) &a, (void *) &storage,
mpitype, &m, &n, MPI_COMM_WORLD);
   rows = BLOCK_SIZE(id,p,m);
   print_row_striped_matrix((void **) a, mpitype, m, n,
MPI_COMM_WORLD);
   read_replicated_vector(argv[2], (void *) &b, mpitype, &nprime,
MPI_COMM_WORLD);
   print_replicated_vector(b, mpitype, nprime, MPI_COMM_WORLD);

   c_block = (dtype *) malloc (rows * sizeof(dtype));
   c = (dtype *) malloc (n * sizeof(dtype));
   MPI_Barrier(MPI_COMM_WORLD);
```

```
   seconds = - MPI_Wtime();
   for (i = 0; i < rows; i++) {
      c_block[i] = 0.0;
      for (j = 0; j < n; j++)
         c_block[i] += a[i][j] * b[j];
   }

   replicate_block_vector(c_block, n, (void *) &c, mpitype,
MPI_COMM_WORLD);
   MPI_Barrier (MPI_COMM_WORLD);
   seconds += MPI_Wtime();

   print_replicated_vector(&c, mpitype, n, MPI_COMM_WORLD);

   MPI_Allreduce(&seconds, &max_seconds, 1, mpitype, MPI_MAX,
MPI_COMM_WORLD);
   if (!id) {
      printf ("MV1) N = %d, Processes = %d, Time = %12.6f sec,", n, p,
max_seconds);
      printf ("Mflop = %6.2f\n", 2*n*n/(1000000.0*max_seconds));
   }
   MPI_Finalize();
   return 0;
}
```

Source: [Qu02, .../mpi/chapter8/mv1.c]

Listing 1: Parallel row-wise matrix-vector multiplication in C.

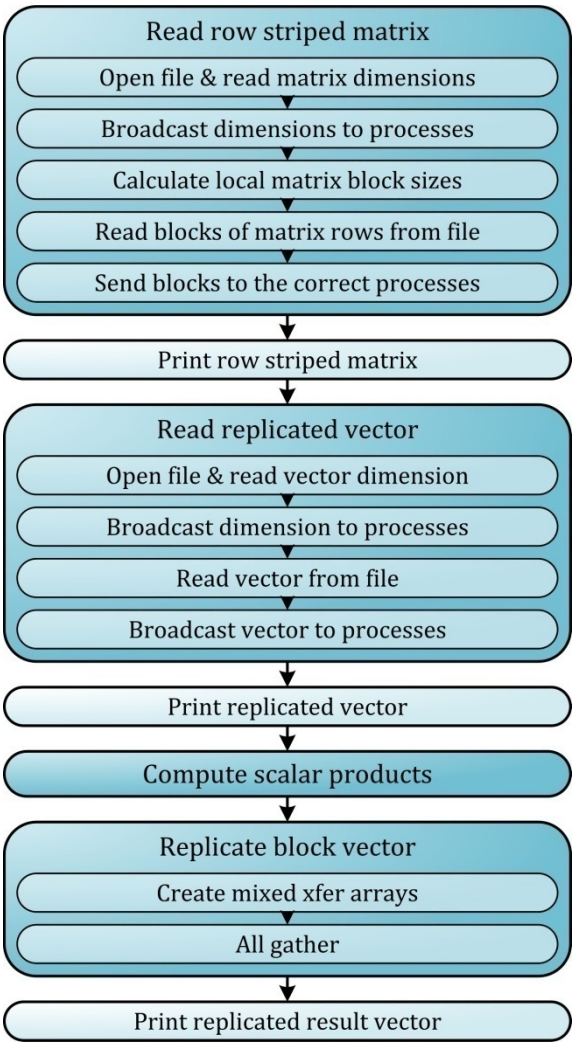


Figure 20: Detailed parallel row-wise implementation scheme.

```
/*
 *   Matrix-vector multiplication, Version 2
 *
 *   This program multiplies a matrix and a vector.
 *   The matrix and vector are input from files.
 *   The answer is printed to standard output.
 *
 *   Data distribution of matrix: columnwise block striped
 *   Data distribution of vector: block
 *
 *   Programmed by Michael J. Quinn
 *
 *   Last modification: 9 September 2002
 */

#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"
#include "MyMPI.c"

/* Change these two definitions when the matrix and vector
   element types change */
```

```
typedef double dtype;
#define mpitype MPI_DOUBLE

int main (int argc, char *argv[]) {
   dtype **a;              /* The first factor, a matrix */
   dtype  *b;              /* The second factor, a vector */
   dtype  *c;              /* The product, a vector */
   dtype  *c_part_out;     /* Partial sums, sent */
   dtype  *c_part_in;      /* Partial sums, received */
   int    *cnt_out;        /* Elements sent to each proc */
   int    *cnt_in;         /* Elements received per proc */
   int    *disp_out;       /* Indices of sent elements */
   int    *disp_in;        /* Indices of received elements */
   int     i, j;           /* Loop indices */
   int     id;             /* Process ID number */
   int     local_els;      /* Cols of 'a' and elements of 'b' held by
this process */
   int     m;              /* Rows in the matrix */
   int     n;              /* Columns in the matrix */
   int     nprime;         /* Size of the vector */
   int     p;              /* Number of processes */
   double  max_seconds;
   double  seconds;        /* Elapsed time */
   dtype  *storage;        /* This process's portion of 'a' */

   MPI_Init (&argc, &argv);
   MPI_Comm_rank (MPI_COMM_WORLD, &id);
   MPI_Comm_size (MPI_COMM_WORLD, &p);

   read_col_striped_matrix (argv[1], (void ***) &a, (void **)
&storage, mpitype, &m, &n, MPI_COMM_WORLD);
   print_col_striped_matrix ((void **) a, mpitype, m, n,
MPI_COMM_WORLD);
   read_block_vector (argv[2], (void **) &b, mpitype, &nprime,
MPI_COMM_WORLD);
   print_block_vector ((void *) b, mpitype, nprime, MPI_COMM_WORLD);

   /* Each process multiplies its columns of 'a' and vector
      'b', resulting in a partial sum of product 'c'. */

   c_part_out = (dtype *) my_malloc (id, n * sizeof(dtype));
   local_els = BLOCK_SIZE(id,p,n);
   MPI_Barrier (MPI_COMM_WORLD);
   seconds = -MPI_Wtime();
   for (i = 0; i < n; i++) {
      c_part_out[i] = 0.0;
      for (j = 0; j < local_els; j++)
         c_part_out[i] += a[i][j] * b[j];
   }

   create_mixed_xfer_arrays (id, p, n, &cnt_out, &disp_out);
   create_uniform_xfer_arrays (id, p, n, &cnt_in, &disp_in);
   c_part_in =
      (dtype*) my_malloc (id, p*local_els*sizeof(dtype));
   MPI_Alltoallv (c_part_out, cnt_out, disp_out, mpitype, c_part_in,
cnt_in, disp_in, mpitype, MPI_COMM_WORLD);

   c = (dtype*) my_malloc (id, local_els * sizeof(dtype));
   for (i = 0; i < local_els; i++) {
      c[i] = 0.0;
      for (j = 0; j < p; j++)
         c[i] += c_part_in[i + j*local_els];
```

```
    }
    MPI_Barrier (MPI_COMM_WORLD);
    seconds += MPI_Wtime();
    MPI_Allreduce (&seconds, &max_seconds, 1, mpitype, MPI_MAX,
MPI_COMM_WORLD);
    if (!id) {
       printf ("MV3) N = %d, Processes = %d, Time = %12.6f sec,", n, p,
max_seconds);
       printf ("Mflop = %6.2f\n", 2*n*n/(1000000.0*max_seconds));
    }
    print_block_vector ((void *) c, mpitype, n, MPI_COMM_WORLD);
    MPI_Finalize();
    return 0;
}
```

Source: [Qu02, .../mpi/chapter8/mv2.c]

Listing 2: Parallel column-wise matrix-vector multiplication in C.

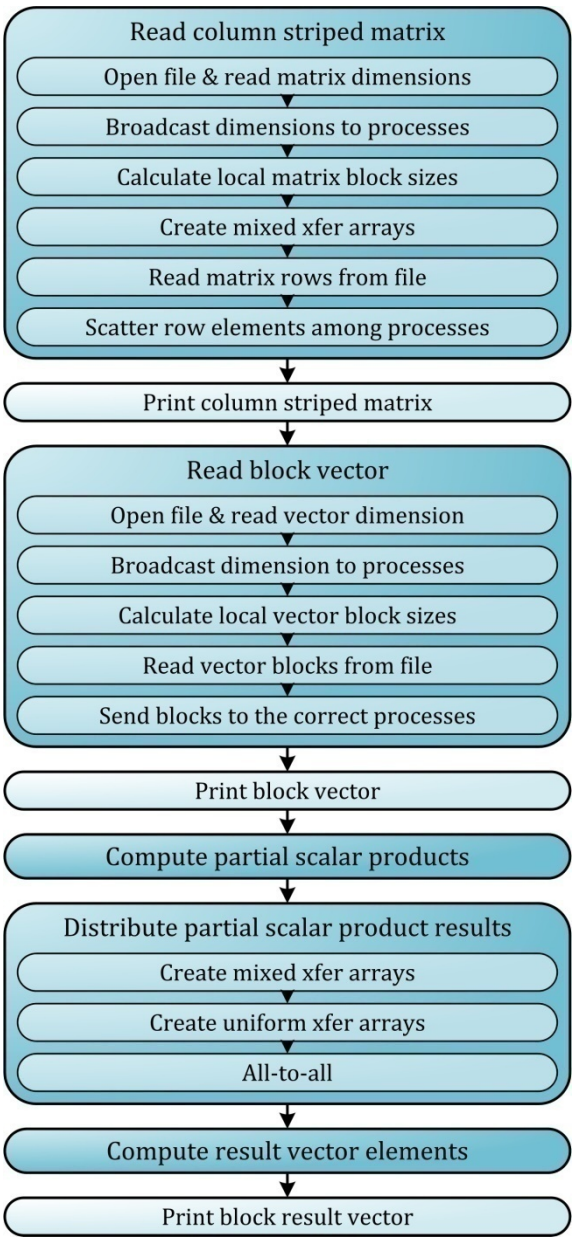


Figure 21: Detailed parallel column-wise implementation scheme.

```
/*
 *   Matrix-vector multiplication, Version 3
 *
 *   This program multiplies a matrix and a vector.
 *   The matrix and vector are input from files.
 *   The answer is printed to standard output.
 *   Does only work for square p (number of processors).
 *
 *   Data distribution of matrix: checkerboard block-wise
 *   Data distribution of vector: block
 *
 *   Programmed by Annika Biermann
 *
 *   Last modification: 24 November 2009
 */

#include <stdio.h>
#include <mpi.h>
#include "MyMPI.h"
#include "MyMPI.c"

/*    Change these two definitions when the matrix and vector
      element types change */

typedef double dtype;
#define mpitype MPI_DOUBLE

int main (int argc, char *argv[]) {
      dtype **a;        /* First factor, a matrix */
      dtype *b;         /* Second factor, a vector */
      dtype *v;         /* Local vector block */
      dtype *c_part_out;/* Partial sums, sent */
      dtype *storage;   /* Matrix elements stored here */
      int   p;          /* Number of processes */
      int   size[2];    /* Vector with size of each grid dimension*/
      int   periodic[2];/* Message wraparound flags*/
      int   id;         /* Rank of process in virtual grid */
      int   i, j;       /* Loop indice */
      int   grid_coords[2];   /* Location of process in grid */
      int   m;          /* Rows in the matrix */
      int   n;          /* Columns in the matrix */
      int   nprime;     /* # of elements in the local initial vector
block */
      int   nfinal;     /* # of elements in the local final vector
block */
      int   dims;       /* # of dimensions of the grid */
      int   local_cols; /* Local # of columns */
      int   local_rows; /* Local # of rows */
      double max_seconds;
      double seconds;   /* Elapsed time for matrix-vector multiply */
      MPI_Comm grid_comm;     /* 2-D process grid, Cartesian topology
communicator  */
      MPI_Comm row_comm;      /* Processes in same row */
      MPI_Comm column_comm;   /* Processes in same column */
      MPI_Status status;      /* Result of receive */

      /* Initialization */
      MPI_Init (&argc, &argv);
      MPI_Comm_rank (MPI_COMM_WORLD, &id);
      MPI_Comm_size (MPI_COMM_WORLD, &p);
      dims = 2;
      nprime = 0;
```

```
      /* Create virtual grid */
      size[0] = size[1] = 0;
      MPI_Dims_create(p, dims, size);
      periodic[0] = periodic[1] = 0;
      MPI_Cart_create(MPI_COMM_WORLD, dims, size, periodic, 1,
&grid_comm);

      if(size[0] == size[1])
      {
            /* Read and print checkerboard matrix */
            read_checkerboard_matrix (argv[1], (void ***) &a, (void **)
&storage, mpitype, &m, &n, grid_comm);
            print_checkerboard_matrix ((void **) a, mpitype, m, n,
grid_comm);
            /* Read and print vector */
            MPI_Cart_coords (grid_comm, id, dims, grid_coords);
            MPI_Comm_split (grid_comm, grid_coords[1], grid_coords[0],
&column_comm);
            if(grid_coords[1] == 0) { // only processes of the first
column
                  read_block_vector (argv[2], (void **) &b, mpitype,
&nprime, column_comm);
                  print_block_vector ((void *) b, mpitype, nprime,
column_comm);
            }
            /* Redistribute vector */
            redistribute_vector((void *) b, (void **) &v, mpitype, n,
nprime, (int *) &nfinal, size[0], size[1], grid_coords[0],
grid_coords[1], grid_comm);

            /* Matrix-vector multiply */

            /* Each process multiplies its submatrix of 'a' and block
of vector 'b', resulting in a partial block sum of product 'c'. */

            local_rows = BLOCK_SIZE(grid_coords[0], size[0], m);
            local_cols = BLOCK_SIZE(grid_coords[1], size[1], n);

            c_part_out = (dtype *) my_malloc (id, local_rows *
sizeof(dtype));
            MPI_Barrier (MPI_COMM_WORLD);
            seconds = -MPI_Wtime();

            for (i = 0; i < local_rows; i++) {
                  c_part_out[i] = 0.0;
                  for (j = 0; j < local_cols; j++)
                  c_part_out[i] += a[i][j] * v[j];
            }

            /* Reduce vectors across rows */
            /* Split grid_communicator into row_communicators */
            MPI_Comm_split (grid_comm, grid_coords[0], grid_coords[1],
&row_comm);
            dtype c_part_in[size[1]-1][local_rows];  /* Partial sums,
received */

            /* If process not in first grid column */
            if (grid_coords[1] != 0) {
                  /* Send block of c to first process in row */
                  MPI_Send (c_part_out, local_rows, mpitype, 0,
DATA_MSG, row_comm);
```

```
            }
            /* if process in first grid column */
            else {
                  // for each column except the first one
                  for (i = 1; i < size[1]; i++) {
                        /* Receive blocks of c from other processes and
do summation */
                        MPI_Recv (c_part_in[i-1], local_rows, mpitype,
i, DATA_MSG, row_comm, &status);
                  }
                  for (i = 0; i < local_rows; i++) {
                        for(j=0;j<size[1]-1;j++){
                              c_part_out[i] += c_part_in[j][i];
                        }
                  }
            }

            MPI_Barrier (MPI_COMM_WORLD);
            seconds += MPI_Wtime();
            MPI_Allreduce (&seconds, &max_seconds, 1, mpitype, MPI_MAX,
MPI_COMM_WORLD);
            if (!id) {
                  printf ("MV3) N = %d, Processes = %d, Time = %12.6f
sec,", n, p, max_seconds);
                  printf ("Mflop = %6.2f\n",
2*n*n/(1000000.0*max_seconds));
            }
            if(grid_coords[1] == 0)
                  print_block_vector ((void *) c_part_out, mpitype, n,
column_comm);
      }

      MPI_Finalize();
      return 0;
}

/*
 *     This function is used to redistribute a vector block
 *     from the first process grid column processes
 *     to all processes within a communicator.
 */

void redistribute_vector(
      void  *b,               /* IN - Address of vector */
      void  **v,              /* OUT - Subvector */
      MPI_Datatype dtype,     /* IN - Vector element type */
      int   n,                /* IN - Size of (whole) input vector */
      int   nprime,          /* IN - Elements in initial block vector */
      int   *nfinal,         /* OUT - Elements in final block vector */
      int   grid_rows,        /* IN - Number of rows in the virtual
process grid */
      int   grid_cols,        /* IN - Number of columns in the virtual
process grid */
      int   grid_row_coord,   /* IN - row index of process in virtual
process grid */
      int   grid_col_coord,   /* IN - column index of process in
virtual process grid */
      MPI_Comm grid_comm)     /* IN - Communicator */
{
      int   datum_size;       /* Bytes per vector element */
      MPI_Status status;      /* Result of receive */
      int   id;               /* Process rank */
```

```
      int   p;                /* Number of processes */
      int   dest_coords[2];   /* Coordinates of receiving process */
      int   dest_id;          /* Rank of receiving process */
      int   sent_coords[2];   /* Coordinates of sending process */
      int   sent_id;          /* Rank of sending process */
      MPI_Comm column_comm;   /* Virtual grid column communicator */

      MPI_Comm_size (grid_comm, &p);
      MPI_Comm_rank (grid_comm, &id);
      datum_size = get_size (dtype);

      *nfinal = BLOCK_SIZE(grid_row_coord, grid_cols, n);

      /* Allocate memory for final local vector block */
      *v = my_malloc (id, (*nfinal) * datum_size);

      /* if p is a square number (grid_rows == grid_cols) */
      if(grid_rows == grid_cols)
      {
            /* Send/receive blocks of vector */
            /* if process of first grid column */
            if(grid_col_coord == 0) {
                  /* if process 0 */
                  if (grid_row_coord == 0) {
                        /* copy vector block */
                        memcpy(*v, b, (*nfinal) * datum_size);
                  }
                  else {
                        /* Set coordinates */
                        dest_coords[0] = grid_col_coord;
                        dest_coords[1] = grid_row_coord;
                        /* Get rank of destinating process */
                        MPI_Cart_rank (grid_comm, dest_coords,
&dest_id);
                        /* Send vector block */
                        MPI_Send (b, *nfinal, dtype, dest_id, DATA_MSG,
grid_comm);
                  }
            }
            /* if process of first grid row (except process 0) */
            else if (grid_row_coord == 0) {
                  /* Set coordinates */
                  sent_coords[0] = grid_col_coord;
                  sent_coords[1] = grid_row_coord;
                  /* Get rank of sending process */
                  MPI_Cart_rank (grid_comm, sent_coords, &sent_id);

                  /* Receive vector block */
                  MPI_Recv (*v, *nfinal, dtype, sent_id, DATA_MSG,
grid_comm, &status);
            }

            /* Broadcast blocks from first row per column */
            /* Split grid communicator into column communicator */
            MPI_Comm_split (grid_comm, grid_col_coord, grid_row_coord,
&column_comm);
            /* Broadcast vector block per column */
            MPI_Bcast (*v, *nfinal, dtype, 0, column_comm);
      }
      /* if p is not a square number (grid_rows != grid_cols) */
      else {
            //not implemented
```

```
        }
}
```

Listing 3: Parallel square block-wise matrix-vector multiplication in C.

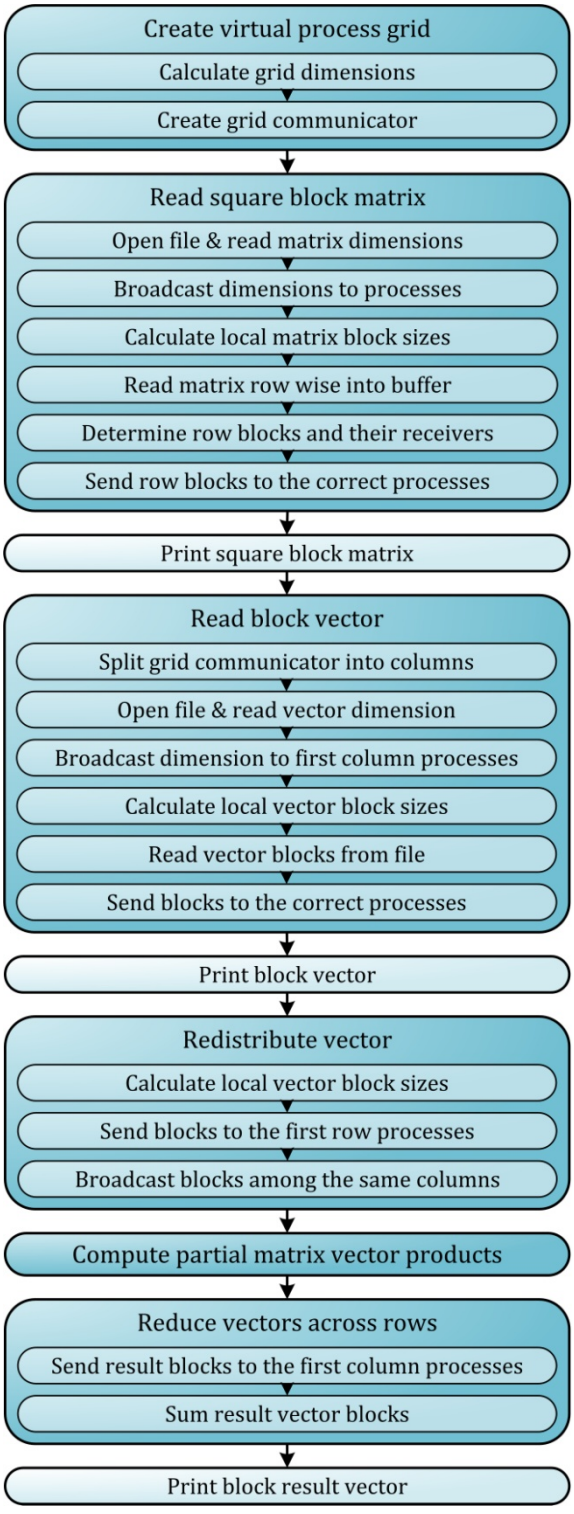


Figure 22: Detailed parallel square block-wise implementation scheme.

Bibliography

[GLS07] William Gropp, Ewing Lusk, Anthony Skjellum: *MPI - Eine Einführung, Portable parallele Programmierung mit dem Message-Passing Interface*, Oldenbourg, 2007.

[HLP95] Bruce Hendrickson, Robert Leland, Steve Plimpton: *An Efficient Parallel Algorithm for Matrix-Vector Multiplication*, International Journal of High Speed Computing 7(1), pp. 73-88, 1995.

[HW94] Bruce A. Hendrickson, David E. Womble: *The torus-wrap mapping for dense matrix calculations on massively parallel computers*, SIAM Journal on Scientific Computing 15(5), pp. 1201-1226, Society for Industrial and Applied Mathematics, 1994.

[LG93] John G. Lewis, Robert A. Van de Geijn: *Distributed memory matrix-vector multiplication and conjugate gradient algorithms*, Proceedings of the 1993 ACM/IEEE Conference on Supercomputing, pp. 484-492, ACM, 1993.

[Ll96] John Lloyd: *Formulations of Matrix-Vector Multiplication for Matrices with Large Aspect Ratios*, 4th Euromicro Workshop on Parallel and Distributed Processing, pp. 102-108, PDP ’96, 1996.

[Qu03] Michael J. Quinn: *Parallel Programming in C with MPI and OpenMP*, McGraw Hill Higher Education, 2003.

[Qu02] Michael J. Quinn: *Source code appearing in Parallel Programming in C with MPI and OpenMP*. http://fac-staff.seattleu.edu/quinnm/web/education/ParallelProgramming/

[SK02] Martin Schmollinger, Michael Kaufmann: *Algorithms for SMP-Clusters Dense Matrix-Vector Multiplication*, International Parallel and Distributed Processing Symposium, vol. 2, pp. 148-154, IPDPS 2002 Workshops, 2002.

[Te05] Ingolf Terveer: *BWL-Crash-Kurs Mathematik*, UTB, 2005.