ELSEVIER

Parallel Computing 27 (2001) 737–759

PARALLEL COMPUTING

www.elsevier.com/locate/parco

Hyper-systolic matrix multiplication

Th. Lippert [a,*], N. Petkov [b], P. Palazzari [c], K. Schilling [a]

[a] Department of Physics, University of Wuppertal, D-42097, Wuppertal, Germany
[b] Institute of Mathematics and Computing Science, University of Groningen, PO Box 800, 9700 AV, Groningen, The Netherlands
[c] ENEA, HPCN Project, C.R. Casaccia, Via Anguillarese, 301, S.P. 100 00060 S.Maria di Galeria, Rome, Italy

Received 23 October 1998; received in revised form 9 July 1999; accepted 20 June 2000

Abstract

A novel parallel algorithm for matrix multiplication is presented. It is based on a 1-D hyper-systolic processor abstraction. The procedure can be implemented on all types of parallel systems. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Matrix multiplication; Hyper-systolic; Parallel computer

1. Introduction

Matrix multiplication is a fundamental operation in many numerical linear algebra applications. Its efficient implementation on parallel computers is an issue of prime importance when providing such systems with scientific software libraries [1]. Consequently, considerable effort has been devoted in the past to the development of efficient parallel matrix multiplication algorithms, and this will remain a task in the future as well.

The choice of a proper parallel algorithm strongly depends on the architecture of the parallel computer on which the algorithm is to run. System aspects, such as SIMD or MIMD mode of operation, distributed or shared memory organization, cache or memory bank structure, construction, throughput and latency of the communication network, processor performance and size as well as throughput of local memory, etc., may render an algorithm which is highly efficient for one system

[*] Corresponding author.
E-mail address: lippert@theorie.physik.uni-wuppertal.de (T. Lippert).

rather impractical for another one. Even on a given system it may be necessary to use different algorithms in different problem size domains.

As a consequence, one needs a variety of parallel algorithms for one and the same operation. For this purpose systematic design approaches are needed which allow to construct new algorithms or to modify existing ones in such a way that they suit both a given system and problem size domain. In the following, one such design approach is presented and applied to matrix multiplication. This approach is used to construct a novel class of parallel matrix multiplication algorithms for distributed memory computers with ring interconnection pattern. The approach is based on the hyper-systolic parallel computing concept [2] which can be generalized for any kind of commutative and associative operation on abstract data types [3]. The communication complexity of the considered class of hyper-systolic matrix multiplication algorithms is $O(n^2 p^{1/2})$, with n being the matrix dimension and p the number of processors. It is thus comparable to the communication complexity of the best standard parallel methods known.

Systolic arrays are cellular automata models of parallel computing structures in which data processing and transfer are pipelined and the cells carry out functions of equal load between consecutive communication events. Systolic algorithms are parallel algorithms which, as far as abstract automata models are concerned, make efficient use of systolic arrays. For more precise definitions of systolic algorithms and arrays and for many examples, the reader is referred to the monographs in [4] and [5] (for a number of systolic matrix multiplication algorithms see Chapter 3 of [5]).

The original motivation behind the systolic array concept was its suitability for VLSI implementation [6,7]. Only a few systolic algorithms, however, have been implemented in VLSI chips or hardware devices. With the advent of commercially available distributed memory parallel computers, systolic algorithms found an attractive implementation medium. Systolic algorithms can be implemented efficiently even within the restricting SIMD model. Apart from these implementation issues, a very attractive aspect is the availability of methodologies for the systematic design of systolic algorithms. Projection of regular dependence graphs has evolved as one such technique [4,5,8–17].

As shown elsewhere [18–20], systolic algorithms can easily be transformed into data-parallel programs. Such a program has certain characteristic features. In particular, it consists of a sequence of identical steps organized in a loop whose counter corresponds to the clock of the underlying systolic array automaton model. Further, the local regular interconnection pattern of a systolic array results in the use of only local synchronized communication in the respective data-parallel program as exemplified by the *shift*-type operations (e.g. cshift and eoshift).

The concept of hyper-systolic algorithm has been introduced in order to reduce the communication overhead of systolic algorithms [2]. The three main differences are: (i) use of a changing interconnection pattern throughout the execution of the algorithm, (ii) use of multiple auxiliary data arrays for storage of intermediate results, and (iii) the possible separation of communication and computation. The combination of these three features leads to a reduction of the communication overhead.

A changing communication pattern is used for the communication of data by different strides along a 1-D ring. The regularity of the communication pattern is however retained. As an example of a regular but changing communication pattern one can think of an algorithm in which each processor of a ring communicates data to its first, second, fourth, etc., neighbours in the first, second, etc., steps of the algorithm, respectively.

Auxiliary data arrays are needed for temporary storage of intermediate results. In conventional systolic algorithms such results are either accumulated in place or on the move by shifting them from processor to processor. In a hyper-systolic algorithm they are generated and kept in place for many cycles, using multiple auxiliary data arrays, which are subsequently used to compute the final results.

The use of regular but changing communication patterns can be found in some (conventional) systolic algorithms, such as the systolic implementation [5] of Eklund's matrix transposition algorithm on a hyper-cube [21]. Use of multiple data arrays for solving specific tasks, e.g. problem partitioning, can also be found in systolic algorithm literature (see Chapter 12 in [21]). However, the purposes of these techniques differ from those aimed at in hyper-systolic algorithms: a substantial reduction of the communication overhead.

The advantage of the hyper-systolic over the systolic data flow has already been demonstrated elsewhere for the case of the so-called n^2-problems which involve $O(n^2)$ computation events on pairs of elements in a system of n elements [22].

The systolic computation of n^2-problems on a parallel computer of p processors involves $O(np)$ communication events. The hyper-systolic algorithm can reduce the communication overhead to $O(np^{1/2})$, as has been successfully applied for a prototype n^2-problem, that involves the computation of all n^2 two-body forces for a system of n gravitatively interacting bodies [22]. This progress makes us confident that hyper-systolic processing can be applied to a variety of numerical problems which lead to n^2 computation events. An important application is found in astrophysics where the investigation of the dynamics and evolution of globular clusters is of prime importance [23]. Further examples of applications are protein folding, polymer dynamics, polyelectrolytes, global and local all-nearest neighbours problems, genome analysis, signal processing etc. [24].

The paper is organized as follows: In Section 2, we illustrate the development of a hyper-systolic matrix multiplication algorithm for a 1-D processor array. In Section 3, the concept of hyper-systolic algorithm which involves two moving data arrays is introduced. In Section 4, we present a pseudo-code representation of the hyper-systolic matrix multiplication. Section 5 deals with blocking multiplication and the mapping of the problem onto a parallel system. Finally, Section 6 presents results from real implementations on both a SIMD parallel system and a workstation cluster.

2. Matrix multiplication on a 1-D processor array

Given an $n \times m$ matrix A and an $m \times n$ matrix B, the matrix–matrix product $\mathsf{C} = \mathsf{AB}$ reads

$$c_{i,j} = \sum_{k=1}^{m} a_{i,k} b_{k,j}, \quad i,j = 1, \ldots, n. \tag{1}$$

In this section, we develop a hyper-systolic matrix multiplication algorithm for a 1-D processor array starting from an algorithm for a 2-D array which is related to the algorithm of Fox [25]. The design approach is illustrated for the case of 4×4 matrices.

2.1. Matrix multiplication on a 2-D processor array

2.1.1. Data alignment

In the following, we make use of the concept of an abstract processor array (APA) as defined in HPF [26]. The grid of boxes shown in Fig. 1 represents such a 2-D APA on which the matrices A, B and C involved in the operation C = AB are aligned in a column-skewed fashion (i.e. the first column remains unchanged, the second is rotated upward by 1 position, ..., the kth column is rotated upward by $k - 1$ positions). This allows to carry out the computation in parallel without even requiring indexed addressing functionality for the target parallel computer. Furthermore, no reordering is required in the course of the computation because the skew representation is preserved.

2.1.2. Semi-systolic algorithm

The algorithm consists of p (here $p = 4$) steps: in the first step, the matrix elements $b_{1,1}, b_{2,2}, \ldots, b_{p,p}$ of the matrix B in the first row of the APA are broadcast to all the processors in the corresponding columns and are subsequently multiplied with the elements of the matrix A which reside in these processors. The products which are shown in Fig. 2(a) are accumulated in the corresponding elements of the matrix C

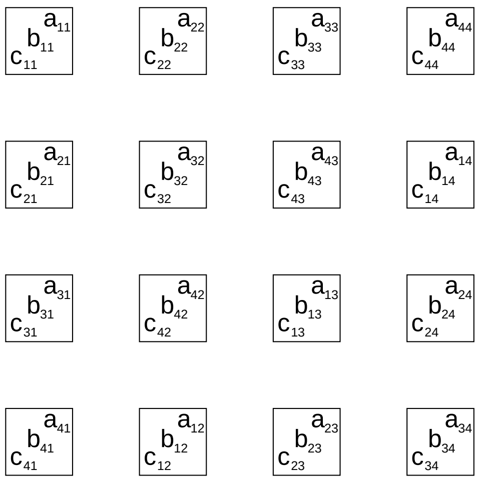


Fig. 1. Column-skewed distribution of the matrices A, B and C on a 2-D APA.

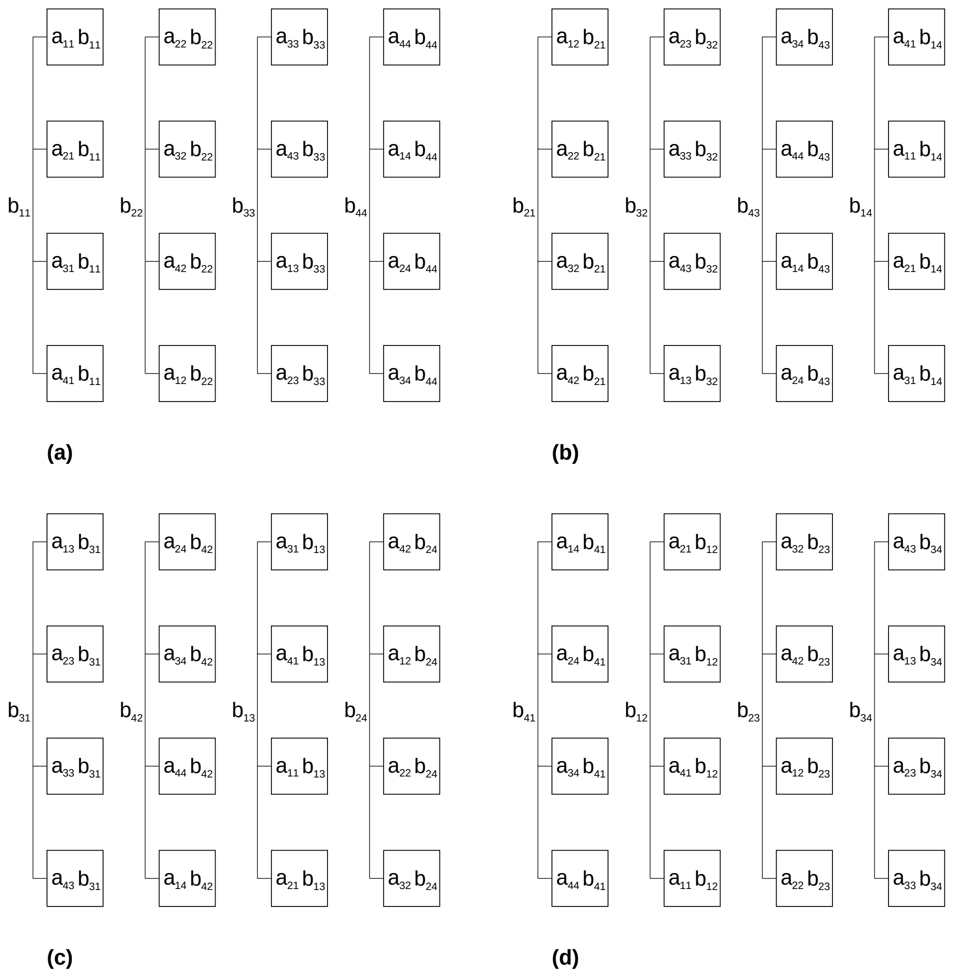


Fig. 2. Computation of partial products on a 2-D APA. At the end of each step, the elements of A are circularly shifted by one position to the left and downwards. (a) $t = 1$; (b) $t = 2$; (c) $t = 3$; (d) $t = 4$.

which are distributed as shown Fig. 1. At the end of the step, the elements of the matrix A are circularly shifted by one position to the left along the rows and by one position downwards along the columns (compare the positions of the elements of matrix A in Fig. 2(a) and (b)).

In the second step, the processors of the second row of the APA broadcast their corresponding elements of the matrix B to all the other processors of the corresponding columns of the APA, Fig. 2(b). This operation is followed by the same multiplication and accumulation operations and circular shifting of the elements of A as in the first step.

The algorithm proceeds with similar steps in which the processors of the third through pth row of the array broadcast in turn their elements of the matrix B, cf.

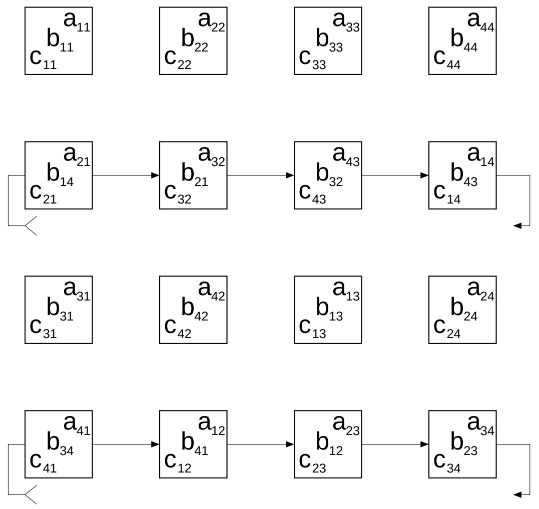
Fig. 2(c)–(d). After a total number of p such steps all partial products which belong to the elements of the matrix C are accumulated in the corresponding processors.

The algorithm is classified as semi-systolic because it involves broadcast operations as well as systolic nearest neighbour shift operations. For general definitions of the term systolic and semi-systolic we refer to [5].

2.1.3. Semi-hyper-systolic algorithm

We next derive a semi-hyper-systolic algorithm from the semi-systolic algorithm just described. The initial distribution of data is shown in Fig. 3. The distribution of the matrices A and C is the same as the one shown in Fig. 1. The distribution of the matrix B is obtained from the distribution shown in Fig. 1 by circular shifting of the elements in the ith row of the processor array by a stride of $(i-1)\text{mod}_K$, with $K = 2$ for $p = 4$. In the particular example of $p = 4$, the second and the fourth row of B are shifted by one position.

The algorithm consists of $p(4)$ steps as shown in Fig. 4. In every step, each processor multiplies the element of the matrix A it holds with the element of the matrix B which it receives via the associated broadcast line. The partial product result is accumulated in one of two local variables which are used to accumulate partial results for the computations of the elements of the matrix C. The local variables represent elements of two auxiliary arrays $C^{(1)}$ and $C^{(2)}$ which are distributed across the processor array. Similar to the original semi-systolic algorithm the processors in the first, second, ..., pth row broadcast the elements of the matrix B which they contain, to all the other processors of the corresponding columns in the first, second, ..., pth step of the algorithm, respectively, see Fig. 4.



a_{11} b_{11} c_{11}
a_{22} b_{22} c_{22}
a_{33} b_{33} c_{33}
a_{44} b_{44} c_{44}
a_{21} b_{14} c_{21}
a_{32} b_{21} c_{32}
a_{43} b_{32} c_{43}
a_{14} b_{43} c_{14}
a_{31} b_{31} c_{31}
a_{42} b_{42} c_{42}
a_{13} b_{13} c_{13}
a_{24} b_{24} c_{24}
a_{41} b_{34} c_{41}
a_{12} b_{41} c_{12}
a_{23} b_{12} c_{23}
a_{34} b_{23} c_{34}

Fig. 3. Initial distribution of data for a semi-hyper-systolic algorithm on a 2-D APA.

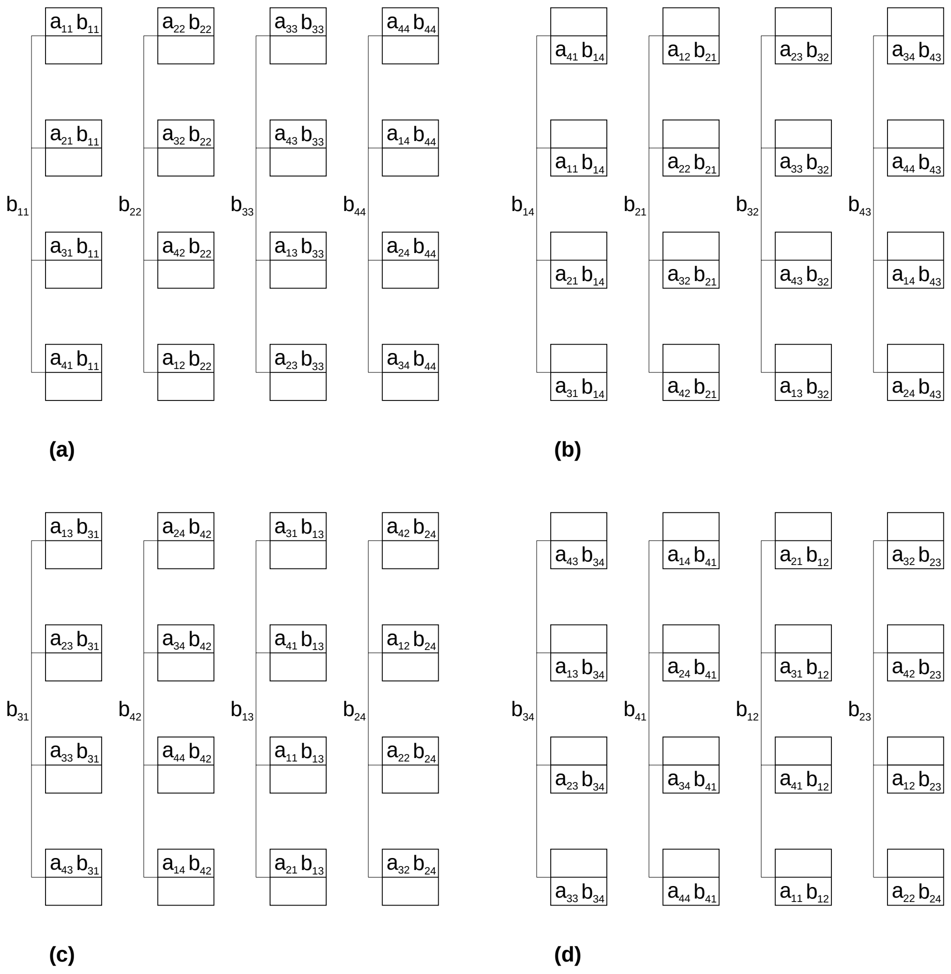


Fig. 4. The products shown in the upper and the lower halves of the processors are accumulated in the auxiliary arrays $C^{(1)}$ and $C^{(2)}$, respectively. The elements of A are cyclically shifted downwards in every step; next to this, they are cyclically shifted to the left by two positions in the even steps only. (a) $t = 1$; (b) $t = 2$; (c) $t = 3$; (d) $t = 4$.

At the end of each step the matrix A is cyclically shifted downwards by one position. Unlike the original algorithm the horizontal shift of the matrix A is performed only every second step by a stride of two elements.

The algorithm is completed by elemental addition of the auxiliary arrays $C^{(1)}$ and $C^{(2)}$ which is preceded by circular shifting of $C^{(2)}$ by one position to the left.

Note that, compared to the original semi-systolic algorithm, the number of communication operations has been reduced: the elements of A are shifted in horizontal direction only every second step. This is however achieved at the expense of increased memory usage. This is a general feature of the class of hyper-systolic algorithms.

We emphasize that the algorithm features a regular communication pattern which makes it similar to a systolic algorithm. The attribute "hyper" is added to refer to the differences which are: data is moved in a regular but not necessarily local pattern, both in space and time, and auxiliary arrays are used to store intermediate results. For a more formal definition we refer to [24].

2.2. Matrix multiplication on a 1-D processor array

Next, we transform the semi-systolic and the semi-hyper-systolic algorithm given above into full systolic ones for a 1-D processor array by mapping the processors in each column of the 2-D array onto one processor of a 1-D array. The operations which are executed in parallel in a given step by the processors in one column of the 2-D array are executed in series by the corresponding processor of the 1-D array.

2.2.1. Systolic algorithm for a 1-D array

The layout of the matrices A, B and C is shown in Fig. 5. The dotted lines indicate the location of the elements in the processors of the original 2-D array.

The systolic 1-D algorithm needs p (here $p = 4$) steps. Since all elements of a given column of B reside on one processor, broadcasting the elements of the matrix B is not required. In the first step, the matrix elements $b_{1,1}, b_{2,2}, \ldots, b_{p,p}$ of the matrix B which reside in the first, second, ..., pth processor, respectively, are multiplied with those elements of the matrix A which reside in the corresponding processors. The products (see Fig. 6(a)) are accumulated in the corresponding elements of the matrix C, according to the arrangement shown in Fig. 5. At the end of the first step, the

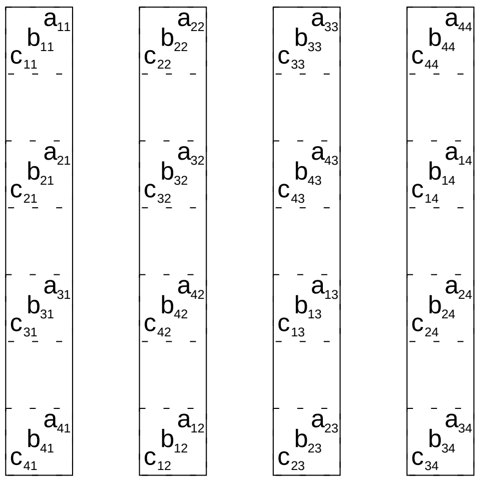


Fig. 5. Data alignment on a 1-D APA. The rectangles correspond to abstract processors.

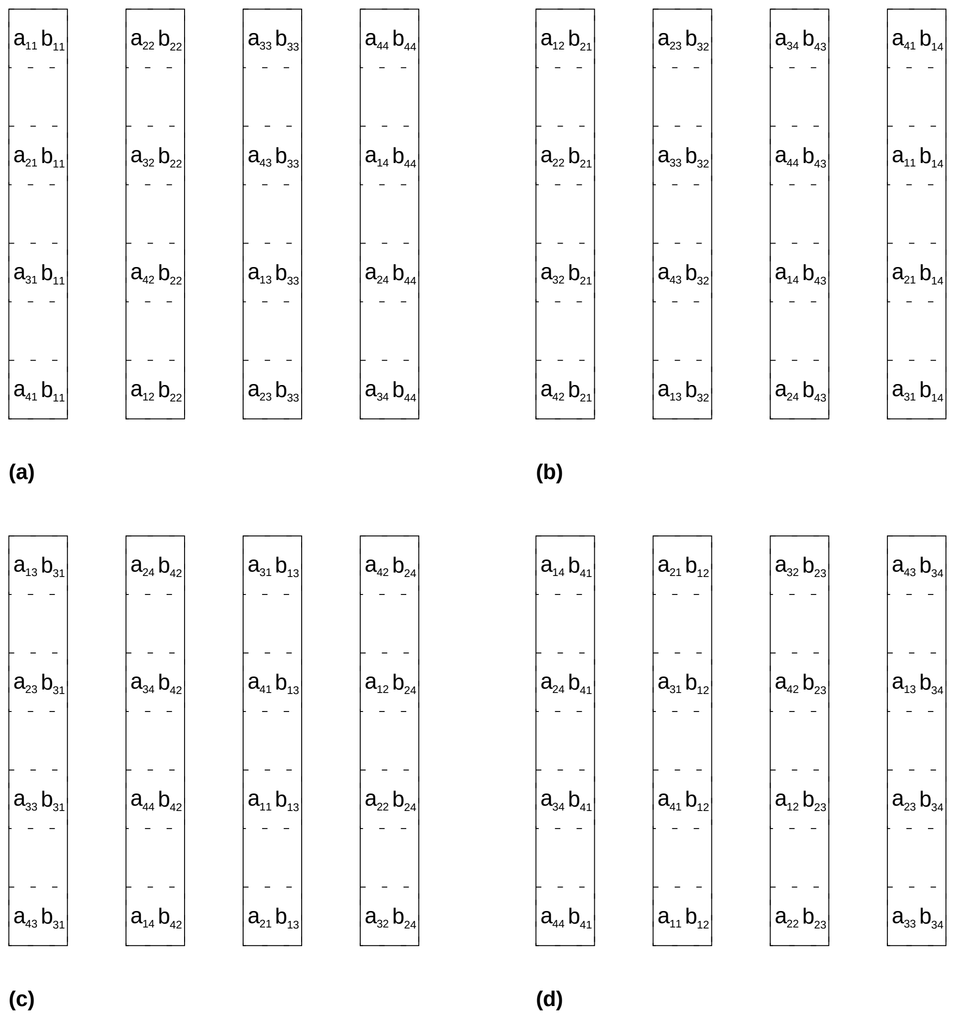


Fig. 6. Systolic computation of partial products on a 1-D APA. The products shown in one abstract processor are computed in series in one abstract time step. (a) $t = 1$; (b) $t = 2$; (c) $t = 3$; (d) $t = 4$.

elements of the matrix A are circularly shifted by one position to the left along the rows.

In the second step, the second row of the matrix B is involved in the computation of partial products, see Fig. 6(b). Its elements are multiplied by the elements of A which reside in the corresponding processors and the partial products are accumulated in their proper locations, i.e., in the second step the products are copied to the elements of C which are circularly assigned one position downwards. At the end of the step, A is circularly shifted to the left by one position.

The algorithm proceeds with similar steps in which the elements of the third, through pth row of the matrix B are multiplied with the elements of A, the products

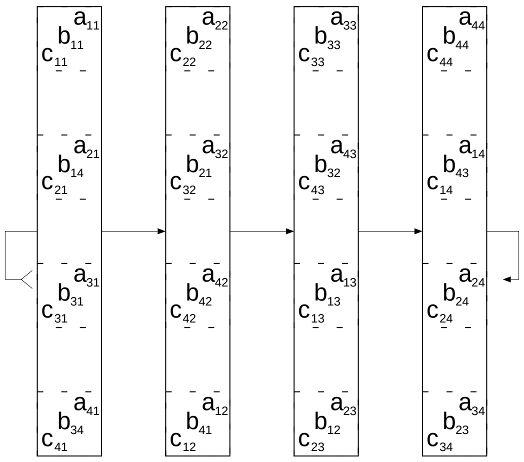


Fig. 7. Initial distribution of data for the hyper-systolic matrix product on a 1-D APA.

being assigned to a row of C at locations which are circularly shifted $i-1$ positions downwards the row in the ith step. At the end of each step the elements of A are circularly shifted to the left, cf. Fig. 6(c)–(d). After a total number of p steps all partial products which belong to the elements of the matrix C are accumulated in the corresponding processors (see Fig. 7).

2.2.2. Hyper-systolic algorithm for a 1-D processor array

Let us turn now to the 1-D realization of the hyper-systolic algorithm. The initial data distribution is obtained in a similar way as in Figs. 3 and 4; here, each column is assigned to one processor.

Step by step each processor multiplies the elements of the matrix A it contains with the corresponding element of the matrix B. The partial products thus computed are accumulated alternately in one of two local variables. In the ith step, the product is assigned to a row located $i-1$ elements downwards, cf. Fig. 8. The elements of A are shifted only every second step in horizontal direction to the left by two elements. [1]

The algorithm is completed by elemental addition of the auxiliary arrays $C^{(1)}$ and $C^{(2)}$ which is preceded by circular shifting of $C^{(2)}$ by one position to the left.

This mapping eliminates the broadcasting used in the semi-hyper-systolic 2-D algorithm, and moreover, control structures become simpler. Downward shifts of A are in reality merely a re-assignment within one processor and do not involve interprocessor communication.

[1] For the general case with p processors, we will show below that, for p steps, the number of shifts is reduced to a number $\tilde{K} < p$; the minimal possible number of shifts is $\tilde{K} = \sqrt{p}$.

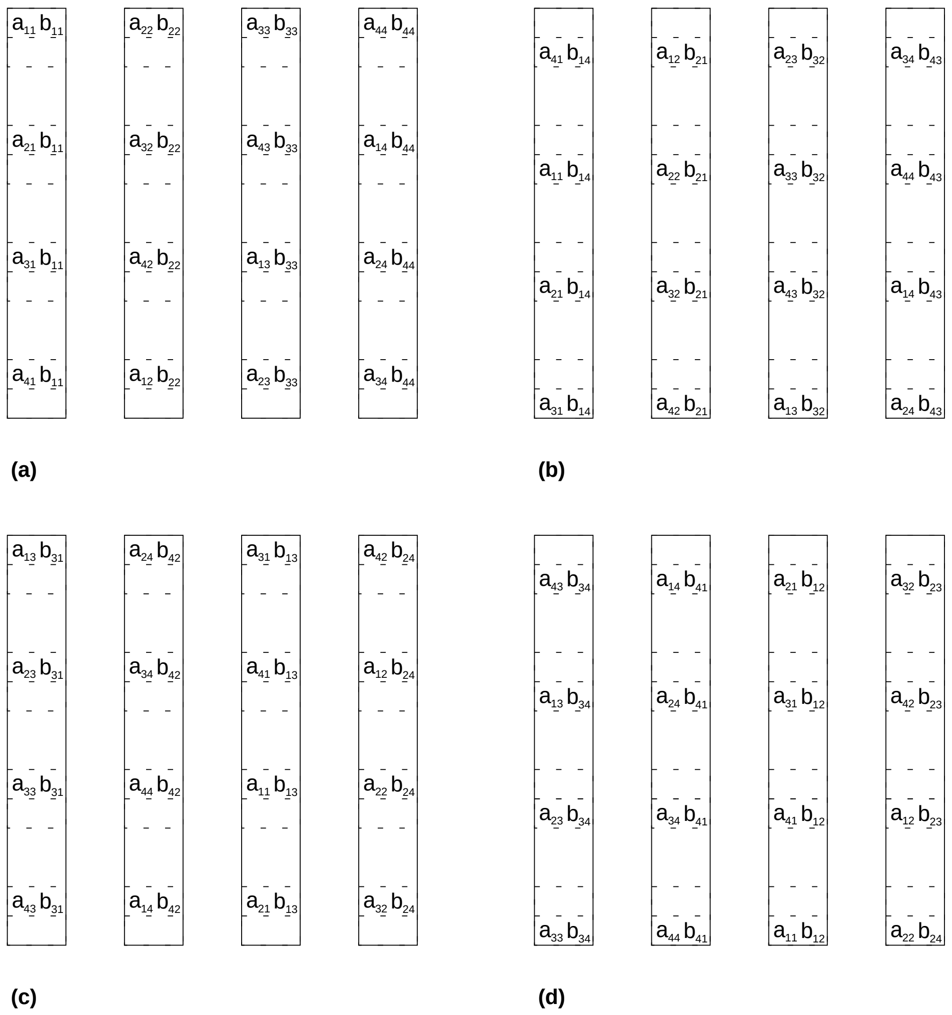


Fig. 8. Partial products computed in the 1-D hyper-systolic algorithm (cf. Fig. 4). (a) $t = 1$; (b) $t = 2$; (c) $t = 3$; (d) $t = 4$.

2.2.3. Summary

Starting from an algorithm for a 2-D APA, we developed a hyper-systolic algorithm on a 1-D APA, with the following properties:
- Regular interprocessor communication is used.
- The number of interprocessor communication operations is less or equal to that of known 1-D systolic algorithms.
- The data layout is identical for the matrices A, B, and C.

Next we generalize the 4×4 problem to a $p \times p$ system, distributed on a p processor array.

3. Hyper-systolic bases

For matrix multiplication, the hyper-systolic algorithm involves two moving data streams. Note that the hyper-systolic algorithm defined in [2] is formulated for problems that involve only one moving data stream.

3.1. Recipe

Let $\vec{x}$ and $\vec{z}$ be two 1-D arrays both of length n. Assume that functions

$$F_i = \oplus_{j=1}^{n} f(x_i, z_j), \quad i = 1, \ldots, n \tag{2}$$

are computed for each i, with $\oplus$ being an associative and commutative operator. A typical example encountered in n-body problems is the computation of the force F_i exercised on a particle with coordinates x_i by all other particle. The computation F_i, $i = 1, \ldots, n$, can readily be carried out employing a systolic algorithm on a ring of processors [22].

In a straightforward approach, the elements of the arrays $\vec{x}$ and $\vec{z}$ are distributed across n processors of a ring, one element of each array per processor. While the elements of $\vec{z}$ stay in place, the elements of $\vec{x}$ are shifted cyclically by one position, and after each shift, the function f is evaluated for each pair (x_k, z_j) that is found in a given processor together with a partial result of the function F.

The computation is completed in n steps. Note that while the memory requirements are minimal – one memory location is needed for each element of the arrays involved – the communication is abundant. Each processor sends and receives data in each step.

The communication can be reduced on the expense of increased memory requirements. In an extreme case, one copy of the complete coordinate array could be communicated to each processor and thus, each processor could compute one element of the result array F without any further communication.

Hyper-systolic algorithms take on an intermediate position between the two extreme cases in that they require less communication than the systolic method and less memory than the last method.

In general, a number of copies of the arrays are needed in a hyper-systolic algorithm. This number is, however, smaller than the number of processors involved. These copies are shifted at different distances. The sequence of distances is called the *hyper-systolic base*. Next, the scheme is explained in more detail.

The general recipe:

1. From the array $\vec{x}$, k replicas $\hat{\vec{x}_t}$, $1 \leqslant t \leqslant k$ are generated. They are shifted by strides a_t, $1 \leqslant t \leqslant k$ with respect to $\vec{x}$.
   For the array $\vec{z}$, k' replicas $\hat{\vec{z}_{t'}}$, $1 \leqslant t' \leqslant k'$ are generated. They are shifted by strides $b_{t'}$, $1 \leqslant t' \leqslant k'$ with respect to $\vec{z}$.
2. The sequences of strides $\{a_t\}$, $1 \leqslant t \leqslant k$ and $\{b_{t'}\}$, $1 \leqslant t' \leqslant k'$, called the hyper-systolic bases, are such that
   2.1. each pair of data elements is present at least once on the processor array,
   2.2. the total communication cost is minimized.

3. After each communication event the computations can be carried out and the results are assigned to $k+1$ intermediate result arrays $\hat{\vec{y}}_t$, $1 \leqslant t \leqslant k+1$. If elements occur more than once they are accounted for by a multiplicity table in order to avoid multiple counting.
4. The intermediate result arrays $\hat{\vec{y}}_t$, $1 \leqslant t \leq k+1$ contain partial results that are shifted to their proper location by strides a_t, $1 \leqslant t \leqslant k$. Therefore, an array $\vec{y}$, which collects intermediate results, is moved by strides that follow the inverse of the sequence $\{a_t\}$. In each step of the back-shift phase the required intermediate result arrays $\hat{\vec{y}}_t$ are added to $\vec{y}$.

3.2. The hyper-systolic optimization problem

Contemporary parallel machines support circular shifts of 1-D data arrays. The time needed for such an operation is a function of the shift distance which depends on the underlying communication structure. In general this function is non-linear and not necessarily monotonously increasing. On a hyper-cube, for instance, all shifts with strides that are powers of two take the same amount of time.

The optimal sequence of strides for minimal interprocessor communication will depend on the interprocessor communication cost for a given stride. In order to minimize the communication cost effect on a given machine, we introduce a cost function $C(a_i)$, as a function of the stride a_i.

For the sake of argumentation, let us first assume the costs of communication for each array $\vec{x}$ and $\vec{z}$ on the systolic ring to be constant for any stride a_i, b_i. $C(a_i) = C(b_{\hat{i}}) = \text{const}$.

Definition 1 (*Optimization problem for* $C(a_i) = C(b_{\hat{i}}) = \text{const.}$). Let I be the set of integers $m = \{0, 1, 2, \ldots, n-1\} \in \mathbb{N}_0^n, n \in \mathbb{N}$. Find the two ordered multi-sets $A_k = (a_0 = 0, a_1, a_2, a_3, \ldots, a_k) \in \mathbb{N}_0^{k+1}$ of $k+1$ integers and $B_{k'} = (b_0 = 0, b_1, b_2, b_3, \ldots, b_{k'}) \in \mathbb{N}_0^{K+1}$ of $k'+1$ integers, with $k+k'$ being a minimum, where each $m \in I$, $(0 \leqslant m \leqslant n-1)$, can be represented at least once as the sum of two ordered partial sums

$$m = (a_i + a_{i+1} + \cdots + a_{i+j}) + (b_{\hat{i}} + b_{\hat{i}+1} + \cdots + b_{\hat{i}+\hat{j}}), \tag{3}$$

with

$$0 \leqslant i + j \leqslant k, \quad i, j \in \mathbb{N}_0, \quad 0 \leqslant \hat{i} + \hat{j} \leqslant k', \quad \hat{i}, \hat{j} \in \mathbb{N}_0. \tag{4}$$

3.2.1. Lower bound on $k+k'$

A lower bound for the minimal number of non-zero elements of A_k can be derived that will deliver optimal complexity.

Theorem 1. *Let A_k and $B_{k'}$ be two bases solving the optimization problem for the hyper-systolic algorithm with 2 arrays. Then the minimal length $k+k'$ is given by*

$$k = k' = \sqrt{n} - 1. \tag{5}$$

Proof. The total number of combinations required is n^2 as each element of the first array must come into contact with the n elements of the second array. Let the matrices $\mathscr{H}_1$ and $\mathscr{H}_2$ be realized by $k-1$ and $k'-1$ shifts, respectively. In that case each element of the first matrix can be combined with k' elements of the second matrix, therefore the possible number of combinations will be nkk'. Given $n = kk'$, the minimum number of circular shifts $k + k'$ is attained for $k = k' = \sqrt{n} - 1$. □

Therefore, the complexity for the interprocessor communication of a hyper-systolic algorithm for $C(a_i) = C(b_i) = \text{const.}$ is bounded from below by $3(\sqrt{n} - 1)$ shifts, where we have already included the costs for the back-shifts.

We next assume that the cost for a circular shift is a function of the strides a_i, $C(a_i)$ and $C(b_{\hat{i}}) \neq \text{const.}$ The optimization problem of definition 1 is modified only slightly, however, the construction of an optimal base can be quite complicated.

Definition 2 (*Optimization problem for* $C(a_i) = C(b_{\hat{i}}) \neq \text{const.}$). Let I be the set of integers $m = \{0, 1, 2, \ldots, n-1\} \in \mathbb{N}_0^n, n \in \mathbb{N}$. Find the two ordered multi-sets $A_k = (a_0 = 0, a_1, a_2, a_3, \ldots, a_k) \in \mathbb{N}_0^{k+1}$ of $k+1$ integers and $B_{k'} = (b_0 = 0, b_1, b_2, b_3, \ldots, b_{k'}) \in \mathbb{N}_0^{K+1}$ of $k'+1$ integers, with the total cost

$$C_{\text{total}} = \sum_{i=1}^{k} C(a_i) + \sum_{\hat{i}=1}^{k'} C(b_{\hat{i}}) \tag{6}$$

being a minimum, where each $m \in I$, $(0 \leqslant m \leqslant n-1)$, can be represented at least once as the sum of two ordered partial sums

$$m = (a_i + a_{i+1} + \cdots + a_{i+j}) + (b_{\hat{i}} + b_{\hat{i}+1} + \cdots + b_{\hat{i}+\hat{j}}), \tag{7}$$

with

$$0 \leqslant i + j \leqslant k, \quad i, j \in \mathbb{N}_0, \quad \hat{i} + \hat{j} \leqslant k', \quad \hat{i}, \hat{j} \in \mathbb{N}_0. \tag{8}$$

3.3. Regular bases

The 4×4 matrix multiplication problem presented in Section 2 uses so-called regular bases. This prescription turns out to be optimal for equal cost of any stride executed in circular shift operations on the ring. Regular hyper-systolic bases are advantageous as they require only two distinct strides.

Definition 3 (*Regular Bases*). The regular bases are given by

$$A_{k=K-1} := \left(0, 1, \underbrace{1, \ldots, 1}_{K-1}\right) \quad B_{k'=\tilde{K}-1} := \left(0, K, \underbrace{K, \ldots, K}_{\tilde{K}-1}\right), \quad K \times \tilde{K} = n. \tag{9}$$

The completeness of a base pair is defined in terms of the h-range of the base, a notion borrowed from additive number theory [27,28]:

Theorem 2. *The h-range of a regular base is n.*

Proof. Let

$$r := m \operatorname{mod}_{\tilde{K}} \to r < K, \tag{10}$$

where $K\tilde{K} = n$. There are $K-1$ elements $a_i = 1 \in A_k$. Thus any r with $0 \leqslant r \leqslant K-1$ $r \in \mathbb{N}_0$ can be represented as partial sum by the elements $a_i = 1$, $a_i \in A_k$. The partial sums of $B_{k'}$,

$$\sum_{l=i}^{j} b_l = K \sum_{l=i}^{j} 1 = (j-i+1)K < n, \tag{11}$$

are integer multiples of K. Adding the partial sums to r we can therefore represent any element $m \in I$. Thus, the h-range of the base pair $A_{k=K-1}$ and $B_{k'=\tilde{K}-1}$ is n, *i.e.*, the base pair is complete. □

Theorem 3. *The lower bound to the minimal length of the regular bases for a given h-range n is* $K = \tilde{K} = \sqrt{n}$.

Proof. The regular base A_k is complete.

$$k = K + \tilde{K} - 1 \to k = K + \frac{n}{K} - 1. \tag{12}$$

Differentiation gives $K = \sqrt{n}$. □

Theorem 4. *The communication gain factor R that compares the regular hyper-systolic to the systolic algorithm is*:

$$R = \frac{n-1}{2K + \tilde{K} - 3} \approx \frac{\sqrt{n}}{3}. \tag{13}$$

Proof. One needs $K-1$ shifts by 1 and $\tilde{K}-1$ shifts by K in forward direction and again $K-1$ shifts by 1 in backward direction, respectively; therefore, the total number of shifts required is

$$T = (2K + \tilde{K} - 3). \tag{14}$$

The standard systolic computation requires $n-1$ shifts altogether. □

4. Hyper-systolic matrix product

Next we present the general formulation of the systolic and hyper-systolic matrix product in terms of a pseudo-code. The size of the matrices is $p \times p$ and the 1-D processor array consists of p nodes.

4.1. Systolic algorithm

The systolic version of the matrix product of two matrices A and B is given in Algorithm 1. The matrices are represented in skew order.

Algorithm 1 (*Systolic matrix-matrix multiplication*).

```
DO $j = 1, p$
    C = C + CSHIFT(A, DIM = 1, SHIFT = $1 - j$) $*$ SPREAD(B(j, :), DIM = 1)
    A = CSHIFT(A, DIM = 2, SHIFT = 1)
ENDDO
```

The algorithm is completely regular. Each cell executes one compute operation together with an assignment followed by a circular shift of the matrix A in each systolic cycle. The skew order is not destroyed during execution of the algorithm. Note that for each processor inner cell assignment operations CSHIFT operations of columns are executed using equal strides in a given step of the parallel algorithm. Hence, one global address suffices and further address computations are not required!

4.2. Hyper-systolic algorithm

4.2.1. Regular bases

We employ the regular bases constructed for the hyper-systolic system. We add a third base C to account for the back-shifts:

$$\begin{aligned} A_{k=\tilde{K}-1} &= (0, K, K, \ldots, K) \\ B_{k'=K-1} &= (0, -1, -1, \ldots, -1) \\ C_{k'=K-1} &= (0, 1, 1, \ldots, 1) \end{aligned} \tag{15}$$

4.2.2. Hyper-systolic matrix multiplication

Algorithm 2 (*Hyper-systolic matrix multiplication*).

```
! pre-shift of matrix B
B($j$, :) = CSHIFT(B(j, :), SHIFT = MOD($1 - j, K$))
! multiplication and shift of matrix A
DO $j = 1, \tilde{K} - 1$
    DO $l = 1, K$
        C(:, :, $l$) = C(:, :, $l$) + CSHIFT(A, DIM = 1, SHIFT = $1 - (j - 1) * K - l$) $*$
        &
        SPREAD(B($(j - 1) * K + l$, :), DIM = 1)
    ENDDO
    A = CSHIFT(A, DIM = 2, SHIFT = $K$)
ENDDO
DO $l = 1, K$
    C(:,:,l) = C(:,:,l) + CSHIFT(A, DIM = 1, SHIFT = $1 - (\tilde{K} - 1) * K - l$) $*$ &
    SPREAD(B($(\tilde{K} - 1) * K + l$, :), DIM = 1)
ENDDO
```

```
! back-shift and accumulation
DO $j = 1, K - 1$
   $\mathsf{C}(:,:,K-j) = \mathsf{C}(:,:,K-j) + \mathtt{CSHIFT}(C(K-j+1), \mathtt{DIM}=2, \mathtt{SHIFT}=1)$
ENDDO
```

The hyper-systolic matrix multiplication, as given in Algorithm 2, proceeds within three steps. In the first part of the algorithm, matrix B is shifted $K-1$ times by strides of 1 along the systolic ring and stored as B^i, $0 \leqslant i \leqslant K-1$. However, as motivated above, for the case of matrix products, we can spare communication: it suffices to shift B in $\tilde{K}$ row blocks of K rows each, where within each block the first row is shifted by a stride of 0 and the last by a stride of $K-1$.

After the preparatory shifts of B, the computation starts. $\tilde{K}$ times, the multiplication of A with K rows of the pre-shifted matrix B is carried out. After each step, A is moved to the left by a shift of stride K. The result is accumulated within K matrices C^i.

Finally, the K intermediate result matrices C^i are shifted back according to base $C_{k'}$ while summed up to the final matrix C. The algorithm is very regular. The skew order is not destroyed during execution, and in any stage, only global addresses are required.

4.2.3. Complexity

The gain factor for the matrix product reads (note that matrix B is only partially shifted):

Theorem 5. *The gain factor R that compares the regular hyper-systolic matrix multiplication to the systolic algorithm is*

$$R = \frac{p-1}{K+\tilde{K}-1} \approx \frac{\sqrt{p}}{2}. \tag{16}$$

Proof. One needs 1 shift of the full matrix B, $\tilde{K}-1$ shifts by K of matrix A and again $K-1$ shifts by 1 of matrix C. Therefore, the total number of shifts required is

$$T = (K+\tilde{K}-1). \tag{17}$$

The standard systolic computation requires $p-1$ shifts of the matrix A. For the $K = \tilde{K} = \sqrt{p}$, $R \approx \sqrt{p}/2$. □

5. Mapping on parallel systems

So far we discussed the generic situation of the matrix dimension p being equal to the number of processors p. We now turn to the general case of $n \times m$-matrices with $n, m > p$.

In order to map the systolic system onto the parallel implementation machine we choose *hierarchy mapping* of the systolic array onto the processors with the option for two different strategies, block and cyclic assignment.

5.1. Block mapping: Speeding up local computations

The block assignment is applied in all standard algorithms, as it allows to exploit local BLAS-3 routines, like dgemm, by which a very high efficiency of local computations can be achieved. While a small block of the matrix **A** is held in the cache or in the registers (thus avoiding cache-to-memory data transfer), in turn, only the columns of **B** and **C** must be exchanged, and all computations in which the given part of **A** is involved can be carried out. In this way, the ratio between the number of computations and the cache-to-memory traffic is minimized to nearly

$$\frac{2l^3}{(3nn+n^2)} = \frac{l}{2} \tag{18}$$

floating point operations per word for real data, with l being the dimension of the sub-block. Asymptotically, the full speed of the CPU should be exploitable.

A $n \times m$-matrix **M** is divided into $p \times p$ blocks of size $\left(\frac{n}{p} \times \frac{m}{p}\right)$ or $\left(\frac{m}{p} \times \frac{n}{p}\right)$,

$$\mathbf{M} \to \mathsf{M}_{i,j}, \quad i = 1, \ldots, p, \quad j = 1, \ldots, p. \tag{19}$$

The multiplication of **A** and **B** proceeds via sub-matrix multiplication denoted as $(\otimes)$:

$$\begin{aligned} \mathsf{C}_{i,j} &= \sum_{k=1}^{p} \mathsf{A}_{i,k} \otimes \mathsf{B}_{k,j}, \quad i = 1, \ldots, p, \quad j = 1, \ldots, p, \\ \mathbf{C} &= \mathbf{AB}. \end{aligned} \tag{20}$$

Altogether a system of $p \times p$ of such blocks is assigned to the p processor array. Now we can use each sub-matrix in the same manner as the scalar matrix elements before. Therefore, the $p \times p$ system of sub-matrices has to be row-skewed for **A** and column-skewed for **B**.

5.2. Cyclic mapping: Reduction of memory overhead

Each block is distributed across the processors as described above for the generic case. The blocking of the $n \times m$-matrix **M** into $\left(\frac{n}{p} \times \frac{m}{p}\right)$ blocks, leads to blocks of size $p \times p$,

$$\mathbf{M} \to \mathsf{M}_{i,j}, \quad i = 1, \ldots, \frac{n}{p}, \; j = 1, \ldots, \frac{m}{p}. \tag{21}$$

The multiplication of **A** and **B** proceeds via block-multiplication, $(\otimes)$:

$$\begin{aligned} \mathsf{C}_{i,j} &= \sum_{k=1}^{\frac{m}{p}} \mathsf{A}_{i,k} \otimes \mathsf{B}_{k,j}, \quad i = 1, \ldots, \frac{n}{p}, \quad j = 1, \ldots, \frac{n}{p}, \\ \mathbf{C} &= \mathbf{AB}. \end{aligned} \tag{22}$$

A skew representation is required for all blocks $\mathsf{M}_{i,k}$ separately. Cyclic mapping leads to a system of $\frac{n}{p} \times \frac{m}{p}$ systolic processes that run in parallel.

Cyclic assignment allows us to reduce the memory overhead of hyper-systolic computations. In general, K full intermediate matrices **C** are necessary. Using cyclic mapping, one can organize the computation in such a way that only one row of the blocks of the intermediate matrix **C** must be stored in a given phase of the algorithm. All the required shifts of the given part of **A** can be carried out while this part of **A** will not be involved in a further computation. Eventually, the corresponding row of **C** is shifted back and accumulated.

5.3. Block-cyclic mapping

One can combine block and cyclic mapping in a hybrid scheme that combines the advantages of both approaches. A good strategy is to choose the block size of the block mapping such that it is optimal for "local" BLAS-3. For the cyclic part one ends up with blocks of size $p \times p$, with the entries being the BLAS-3 blocks.

6. Benchmarks

Hyper-systolic matrix multiplication can be useful on any type of massively parallel system, ranging from mesh-based SIMD systems to work-station clusters.

This fact is demonstrated in the following by implementing the method on both a cluster computer consisting of 64 workstation nodes, and a 128 node SIMD parallel system, equipped with custom designed floating point units.

6.1. Results from a cluster computer

We have implemented the hyper-systolic matrix multiplication on 64 nodes of the 128-node Alpha-Linux cluster computer ALiCE, installed at Wuppertal university in Germany [30]. The machine is equipped with the Alpha 21264 processor.

ALiCE is connected via a Myrinet multi-stage crossbar. The communication latency and throughput are both constant between any two given processors. Thus, the amount of systolic and hyper-systolic interprocessor communication is directly given in terms of circular shifts, see Eq. (16) and does not depend on the specific stride of the circular shift.

Our implementation uses BLAS-3 routines from the "Compaq Extended Math Library" to achieve the largest possible local performance for given matrix block size.

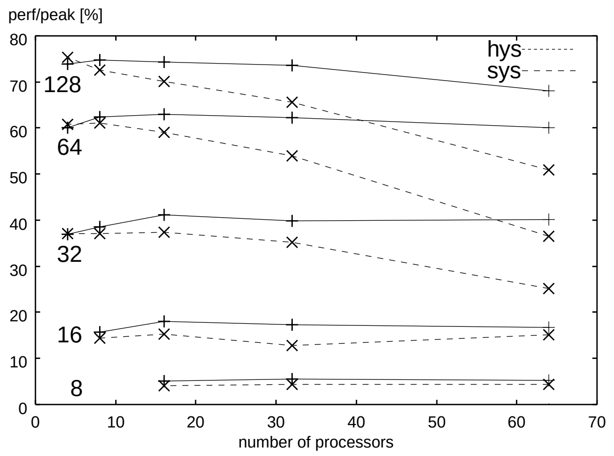


Fig. 9. Relative performance of systolic and hyper-systolic implementations of matrix multiplication on a workstation cluster versus the number of processors. The lines connect equal local block sizes. The results are given for a complex matrix in single precision arithmetics.

Fig. 9 shows the performances achieved in terms of the theoretical peak performance of the machine as function of the number of processors. The lines connect equal block sizes, the block matrix dimension ranges from 8 to 128. This representation allows to distinguish the dependency of the local BLAS-3 efficiency on the block size from the scaling of the parallel part of the algorithm. It is evident that large blocks achieve a high BLAS-3 performance.

The solid lines stand for the hyper-systolic algorithm where the dotted ones give the results for a systolic implementation. The latter shows a pronounced decrease at 64 processors while the hyper-systolic method scales well over the whole range. For 64 processors the hyper-systolic algorithm reduces the amount of interprocessor communication by a factor of 4. Note that at $p = 64$ and a block size of 128×128 the communication time needs half of the compute time in case of the systolic implementation.

6.2. Results from a SIMD computer

The second example demonstrates that the hyper-systolic matrix multiplication also works on less general type of machines: we have implemented the method on the APE100/Quadrics SIMD parallel computer. So far, on Quadrics, lack of indexed addressing has hindered an effective scalable implementation of matrix multiplication [29]. [2]

[2] The implementation on the Quadrics might be a model for the future use of the hyper-systolic method on ASIC chips.

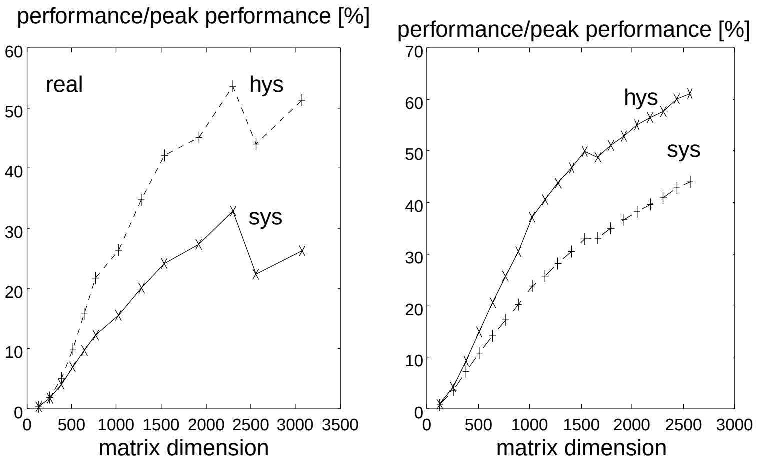


Fig. 10. Performance of systolic and hyper-systolic PBLAS-3 (block-cyclic mapping) on a 128-node APE100, for real and complex data.

We make use of a combination of block and cyclic mapping. Thus, we are able to employ local BLAS, exploiting the CPU with high efficiency, and at the same time, we avoid a memory overhead.

On Quadrics, for real data, the optimal elementary blocks are of size 6×6. For complex data, the size is 4×4. The full matrix is blocked to $p \times p$ matrices which are distributed on the ring and the elements of which are the elementary blocks. Only the $p \times p$ matrices are skew, the elementary matrices remain in normal order. The details of our implementation, in particular the realization of the ring, are given in [31].

We had access to a 128-node APE/100Quadrics QH1 at ENEA/Casaccia in Italy. Fig. 10 shows the performance results for real and complex matrices.

The theoretical peak performances (single node!) for Quadrics are 63% for real data and 88% for complex data, as can be inferred from the maximal ratio of computation versus memory-to-register data transfer times. Hyper-systolic matrix multiplication leads to a peak performance of 65% of peak speed, which translates into 75% of the theoretical performance.

7. Summary

The 1-D hyper-systolic matrix multiplication algorithm is a promising alternative to 2-D matrix product algorithms. Exhibiting equal communication overhead as standard methods like the 2-D Cannon algorithm, the hyper-systolic algorithm avoids non-regular communication and indexed local addressing. Hence, the hyper-systolic matrix product scheme is applicable on any type of parallel system, even on machines that cannot compute indexed addressing. Moreover, the method preserves the alignment of the matrices in the course of the computation. Additionally the alignment for the optimal hyper-systolic algorithm leads to efficient matrix-vector computations as well.

References

[1] J. Choi, J.J. Dongarra, D.W. Walker, The design of scalable software libraries for distributed memory concurrent computers, in: J.J. Dongarra, B. Tourancheau (Eds.), Environments and Tools for Parallel Scientific Computing, Elsevier, Amsterdam, 1992.

[2] T. Lippert, A. Seyfried, A. Bode, K. Schilling, Hyper-systolic parallel computing, IEEE Trans. on Parallel and Distributed Systems 9 (1998) 1.

[3] A. Galli, Generalized hyper-systolic parallel computing', preprint server hep/lat, http://xxx.lanl.gov/ps/hep-lat/9509011.

[4] N. Petkov, Systolische Algorithmen und Arrays, Akademie, Berlin, 1989.

[5] N. Petkov, Systolic Parallel Processing, Amsterdam, North-Holland, 1993.

[6] H.T. Kung, C.E. Leiserson, Systolic arrays (for VLSI), Sparse Matrix Proc., 1978 (Society for Industrial and Applied Mathematics, 1979) pp. 256–282; the same as Algorithms for VLSI processor arrays, in: C. Mead, L. Conway (Eds.), Introduction to VLSI Systems, Addison-Wesley, Reading, MA, 1980, (Sect. 8.3).

[7] H.T. Kung, Why systolic architectures, Computer 15 (1981) 37–47.

[8] P.R. Cappello, K. Steiglitz, Unifying VLSI design with geometric transformations, in: Proceedings of the International Conference on Parallel Processing, 1983, pp. 448–457.

[9] P.R. Cappello, Space time transformation of cellular algorithms, in: E.E. Swartzlander (Ed.), Systolic Signal Processing Systems, Dekker, NY, Basel, 1987, pp. 161–208.

[10] P. Quinton, Automatic synthesis of systolic arrays from uniform recurrent equations, in: Proceedings of the 11th Annual International Symposium on Computer Architecture, Ann Arbor, MI, 1984 (IEEE, NY, 1984), pp. 208–214.

[11] D.I. Moldovan, On the analysis and synthesis of VLSI algorithms, IEEE Trans. on Computers C 31 (1982) 1121–1126.

[12] D.I. Moldovan, On the design of algorithms for VLSI systolic arrays, Proc. IEEE 71 (1983) 113–120.

[13] P. Clauss, G.R. Perrin, Optimal mapping of systolic algorithms by regular instruction shifts, IEEE International Conference on Application-Specific Array Processors, ASAP, 1994, pp. 224–235.

[14] A. Darte, Y. Robert, Affine-by-Statement scheduling of uniform and affine loop nests over parametric domains, Journal of Parallel and Distributed Computing 29 (1995) 43–59.

[15] P. Clauss, V. Loechner, Parametric analysis of polyhedral iteration spaces, IEEE International conference on Application Specific Array Processors, ASAP, 1996.

[16] A. Marongiu, P. Palazzari, A New Memory-Saving Technique to Map System of Affine Recurrence Equations (SARE) onto Distributed Memory Systems, in: Proceedings of the International Parallel Processing Symposium IPPS99, San Juan, Puerto Rico, April, 12–16, 1999, to appear.

[17] A. Marongiu, P. Palazzari, Automatic Mapping of System of N-dimensional Affine Recurrence Equations (SARE) onto Distributed Memory Parallel Systems, accepted for publication in IEEE Transactions on SW Engineering.

[18] T. Dontje, T. Lippert, N. Petkov, K. Schilling, Statistical analysis of simulation-generated time series: Systolic versus semi-systolic correlation on the connection machine, Parallel Computing 18 (1992) 575–588.

[19] N. Petkov, Fuzzy number subtraction convolution on the CM-2, Int. J. of Mod. Phys. C 4 (1993) 181–196.

[20] N. Petkov, Fuzzy number subtraction convolution on the CM-2, in: T. Lippert, K. Schilling, P. Ueberholz (Eds.), Science on the Connection Machine, World Scientific, Singapore, 1993, pp. 181–196.

[21] J.O. Eklundth, A fast computer method for matrix transposing, IEEE Trans. on Computers C 21 (1972) 801–803.

[22] T. Lippert, U. Glaessner, H. Hoeber, G. Ritzenhöfer, K. Schilling, A. Seyfried, Hyper-systolic processing on APE100/quadrics, I. n^2-loop computations, Int. J. Mod. Phys. C 7 (1996) 485.

[23] G. Meylan, D.C. Heggie, Internal Dynamics of Globular Clusters, preprint http://xxx.lanl.gov/ps/astvo-ph/9610076

[24] Th. Lippert, Hyper-Systolic Parallel Computing – Theory and Applications, Ph.D thesis, University of Groningen, 1998.

[25] G.C. Fox, S.W. Otto, Matrix algorithms on a hyper-cube I: matrix multiplication, Parallel Computing 4 (1987) 17–31.

[26] High Performance Fortran Language Specification, Rice University, version 1.1, November 1994. High Performance Fortran, Scientific Programming, 2 (1993).

[27] M. Djawadi, G. Hofmeister, The postage stamp problem, mainzer seminarberichte, Additive Zahlentheorie 3 (1993) 187.

[28] R.K. Guy, Unsolved Problems in Number Theory, Springer, Berlin, New York, 1994.

[29] M. Beccaria, G. Cella, A. Ciampa, G. Curci, A. Viceré, Matrix Inversion on APE100 Machines, Preprint IFUP-TH 17/95.

[30] C. Best, N. Eicker, Th. Lippert, K. Schilling, Linux-clusters for lattice field theory, in: E.H. D'Hollander et al. (Eds.), Proceedings of the International Conference ParCo99, Delft, The Netherlands, 1999, World Scientific, Singapore, 2000.

[31] Th. Lippert, N. Petkov, K. Schilling, BLAS-3 for the Quadrics Parallel Computer, in: B. Hertzberger, P. Sloot (Eds.), Proceedings of the International Conference on High Performance Computing and Networking, HPCN '97, Vienna, Austria, April 1997, Springer, Berlin, 1997, pp. 332–341, 919–930.