Hyper-Systolic Parallel Computing

Thomas Lippert, Armin Seyfried, Achim Bode, and Klaus Schilling

Abstract—We introduce a new class of parallel algorithms for the exact computation of systems with pairwise mutual interactions of n elements, so called n^2-problems. Hitherto, practical conventional parallelization strategies could achieve a complexity of $O(np)$ with respect to the inter-processor communication, p being the number of processors. Our new approach can reduce the inter-processor communication complexity to a number $O(np^{\frac{1}{2}})$. In the framework of Additive Number Theory, the determination of the optimal communication pattern can be formulated as h-range minimization problem that can be solved numerically. Based on a complexity model, the scaling behavior of the new algorithm is numerically tested on the connection machine CM5. As a real life example, we have implemented a fast code for globular cluster n-body simulations, a generic n^2-problem, on the CRAY T3D, with striking success. Our parallel method promises to be useful in various scientific and engineering fields like polymer chain computations, protein folding, signal processing, and, in particular, for parallel level-3 BLAS.

Index Terms—Systolic algorithm, hyper-systolic algorithm, n-body computation, n^2-loop computation, parallel computer, connection machine CM5 and Cray T3D, novel complexity class.

1 Introduction

Within many scientific and engineering applications, one is faced with the intermediate computation of bilocal objects or functions, $f(x_i, x_j)$, on a given set of numbers x_i, $i = 1, \ldots, n$. Think, for example, of the exact treatment of two-body forces in n-body molecular dynamics [1] as employed in astrophysics or thermodynamics of instable systems, convolutions in signal processing [2], autocorrelations in time series [3], n-point polymer chains with long-range interactions, protein-folding [4], or genome analysis.

In general, computation of all $f(x_i, x_j)$ bilocals requires n^2 compute steps. With $f(x_i, x_j)$ being symmetric in i and j, the calculation needs $n(n-1)/2$ different pairings. Thus, the computational cost increases quadratically with the number of particles and, therefore, n^2-calculations are classified as *computationally hard problems* [5].

As one would wish to increase the computational speed, *massively* parallel computers become very attractive, as long as an appropriate speedup can be attained according to the number of processors, p. On parallel machines, the $O(n^2)$ complexity, in principle, can be segmented to $O(\frac{n^2}{p})$ computations of $f(x_i, x_j)$ per processor. This would suggest aiming for the low granularity limit, $g = \frac{n}{p}$ small.

Computations of n^2-problems are not only extremely demanding on memory-to-register (and cache) data movement, moreover, on massively parallel computers with distributed memory, they imply a substantial amount of interprocessor communication as the elements x_i of the array $\mathbf{x}$ are spread out over many processing elements, i.e., local memories.

An example taken from astrophysics will illustrate the computational and communicational efforts required. In globular cluster simulations, molecular dynamics techniques with gravitational forces are applied to track the time evolution of the cluster. A wide range of cluster sizes must be treated in order to extrapolate to large n. At the upper edge, exact state-of-the-art calculations are, at present, restricted to ensemble sizes $n < O(50{,}000)$. The reason is that exact computations are needed, rather than hierarchical approximations that would fail in modeling the relevant physics. In such applications, computations of all elements $f(x_i, x_j)$ have to be performed at each time step, with the number of time steps along the system trajectory in the range of 10^3 to 10^5 [6], [7]. Thus, each integration step amounts to up to $O(10^{10})$ floating point operations and $O(10^5\, p)$ elements to be communicated between the processors of the parallel machine.[1]

Various methods have been devised in the past to exactly solve the n-body problem. We mention two generic parallel approaches:

1) The replicated data method [1] deals with p identical copies of the entire array $\mathbf{x}$ that contains n particles. These copies are to be placed within each processor. On these data, the computation is performed such that each processor k calculates the elements $f(x_i, x_j)$ for $j = 1, \ldots, n$ and $i = (k-1)\frac{n}{p} + 1, \ldots, k\frac{n}{p}$, i.e., each

• T. Lippert and K. Schilling are with HLRZ, c/o FZ-Jülich, D-52425 Jülich, Germany. E-mail: lippert@theorie.physik.uni-wuppertal.de.
• A. Seyfried is with the Department of Physics, University of Wuppertal, D-42097 Wuppertal, Germany.
• A. Bode is with the Department of Physics, Humboldt University, D-10115 Berlin, Germany.

Manuscript received 28 July 1995; revised 22 Mar. 1997.
For information on obtaining reprints of this article, please send e-mail to:

1. We have to emphasize that this situation must be distinguished from simulations with short-range forces (where multi-million-particle simula-

processor performs $O(\frac{n^2}{p})$ compute operations, the total amount of communicated data is *np* elements.

2) The parallel organization of the evaluation in form of a systolic array computation [1], [11], [12], [13], on machines that allow for a one-dimensional ring connectivity, proceeds with *one* moving and *one* fixed array. The *n* elements are distributed across the processing nodes, i.e., $\frac{n}{p}$ elements are assigned to a given processor. In each systolic step, the moving array is shifted by one position along the processing elements and subsequently, the computations are performed. After $p - 1$ "pulse" and "move" operations, all pairs $f(x_i, x_j)$ have been generated. Again, the total amount of communicated data is *np* elements.

In astrophysics, such systolic array computations are known as Orrery-algorithms [14], [15], [16]. Various procedures which combine features of both methods are known in the literature. Common to such approaches is that the complexity of the interprocessor communication is $O(np)$.

In this work, we introduce a parallel computing concept that, to a certain extent, can be regarded as a generalization of systolic array computations with "pulse" and "circular movement." We will point out that a formulation of the computational problem in the context of Additive Number Theory—leading to an *h*-range optimization problem [17]—defines a new class of algorithms. With the base of the *h*-range problem suitably chosen, one can recover the systolic realization of the computational problem. In addition, various other bases can be contrived that allow the complexity of the interprocessor communication to diminish. We will present a selection of shortest bases and a "regular" base that both reduce the complexity of the interprocessor communication to $O(np^{\frac{1}{2}})$. The new method, in the following denoted as "hyper-systolic," has in common with the standard-systolic array computation that, during each step, only *one* array is moving in a regular communication pattern. Therefore, it applies equally to both SIMD and MIMD machines, just as standard-systolic computation. The distinctive feature as compared to the standard-systolic concept, however, lies in the fact that storing of shifted arrays is required, similar to the replicated data method. But, in contrast to the latter, where the storage increases like $n \times p$, a significant reduction in interprocessor communication is achieved with only a moderate amount of additional storage. To reach the optimal hyper-systolic speedup, one needs storage space of size $O(np^{\frac{1}{2}})$. We will demonstrate that, in view of the state-of-the-art problem sizes mentioned above, this is not an obstacle for today's parallel supercomputers.

In Section 2, we present the generic form of the computational problem to be solved. In Section 3, we review the standard-systolic concept and introduce a complexity model to set the stage for further comparisons. In order to provide a graphical illustration, we will phrase the issue in systolic automata models [12], [18]. A suitable generali-

structure, is worked out in Section 4. In Section 5, the communication of the hyper-systolic algorithm is benchmarked in comparison with standard-systolic computations on the Connection Machine CM5. In the last section, we consider a real life example as taken from astrophysics, namely, an *n*-body simulation code as implemented on the Cray T3D.

2 The Computational Problem

Our task is to compute the n^2-problem

$$y_h := \sum_{k=1}^{n} f(x_h, x_k), \quad h = 1, 2, 3, \ldots, n, \quad k \neq h, \tag{1}$$

with $f(x_h, x_k)$ being any long-range pair-interaction, e.g., gravitational or Coulomb forces, as well as more general types of interactions as occur in neural nets or protein folding. We further suppose that the forces are symmetric in *h* and *k* (up to a minus sign). The number of combinations $\{x_h, x_k\}$ required for the computation of all $f(x_h, x_k)$ is

$$\binom{n}{2} = \frac{n(n-1)}{2}, \tag{2}$$

i.e., the computational complexity inherent in (1) is a number $O(n^2)$.

On a parallel computer equipped with *p* compute nodes, this problem can be split into $\frac{n}{p}$ subtasks represented by the generic problem,

$$y_i := \sum_{j=1}^{p} f(x_i, x_j), \quad i = 1, 2, 3, \ldots, p, \quad i \neq j, \tag{3}$$

with the input data

$$\mathrm{x} = (x_1, x_2, x_3, \ldots, x_p) \text{ and } \mathrm{y} = (y_1, y_2, y_3, \ldots, y_p), \tag{4}$$

as the resulting set written as one-dimensional arrays. In molecular dynamics terms, the sequence x can, e.g., be thought of as the coordinates of a number of *n* particles, whereas the sequence y describes the potential (or equivalently a component of the force) by which particle # *i* is influenced in the presence of all the other particles. In the following, we will concentrate on the generic problem (3).

The number of processors *p* of a given parallel machine and the number of elements *n* will differ in practical applications. Here, we will partition the array of *n* elements into $\frac{n}{p}$ subarrays, such that each processor deals with $\frac{n}{p}$ data elements. Each subarray of *p* elements is distributed across the *p* processors as in the generic case. The implementation of (1) with $n > p$, i.e., granularity $g = \frac{n}{p} > 1$, is a straightforward generalization of the generic case, because (1) can be rewritten as:

$$y_{(l-1)p+i} = \sum_{m=1}^{\frac{n}{p}} \sum_{j=1}^{p} f\left(x_{(l-1)p+i}, x_{(m-1)p+j}\right),$$

$$l = 1, 2, 3, \ldots, \frac{n}{p}, i = 1, 2, 3, \ldots, p,$$

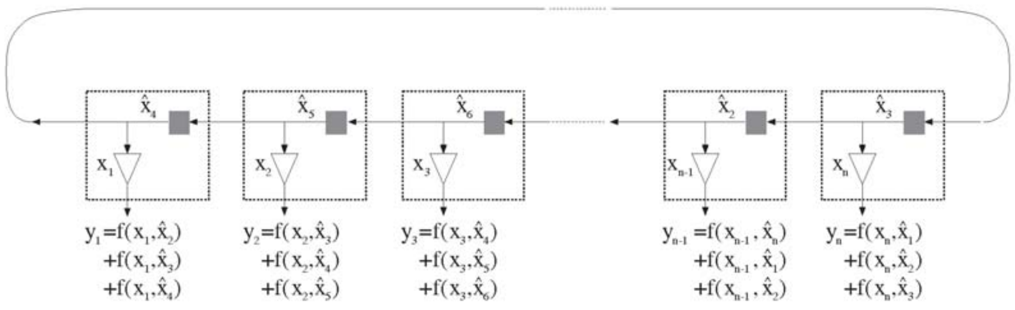
$\hat{x}_4$
$\hat{x}_5$
$\hat{x}_6$
$\hat{x}_2$
$\hat{x}_3$
x_1
x_2
x_3
x_{n-1}
x_n
$y_1 = f(x_1,\hat{x}_2) + f(x_1,\hat{x}_3) + f(x_1,\hat{x}_4)$
$y_2 = f(x_2,\hat{x}_3) + f(x_2,\hat{x}_4) + f(x_2,\hat{x}_5)$
$y_3 = f(x_3,\hat{x}_4) + f(x_3,\hat{x}_5) + f(x_3,\hat{x}_6)$
$y_{n-1} = f(x_{n-1},\hat{x}_n) + f(x_{n-1},\hat{x}_1) + f(x_{n-1},\hat{x}_2)$
$y_n = f(x_n,\hat{x}_1) + f(x_n,\hat{x}_2) + f(x_n,\hat{x}_3)$

Fig. 1. A systolic automaton cell (large dotted square) is an arrangement of a processing element (open triangle) and a delay element (black square). The data element x_i resides stationary in the processing element i. In each clock step, the delay element delivers another data element $\hat{x}_j$ of the moving array. The function $f(x_i, x_j)$ is computed and successively added to y_i that is resident in the processing element as well. The plot shows the data location on the systolic array after the third clock step.

3 Systolic Algorithm

First, we describe the one-dimensional systolic implementation of (3), with $n = p$, to set the stage for our further considerations. We use the concept of systolic automaton models to illustrate parallel computing structures [12], [18].

A systolic automaton consists of cells with data transfer and processing units. The cells are coupled to each other in a uniform next-neighbor wiring pattern, as depicted in Fig. 1. The processing elements (PE) are drawn as open triangles. They realize functions of equal load between consecutive communication events. The data transfer elements are drawn as black squares. They are delay elements (DE) that represent abstractions for memory locations in which data is shifted in and out in regular clock steps, which means the clock step of the abstract automaton, rather than a physical computer clock step. Data processing and transfer are completely pipelined. More precise definitions of systolic arrays and algorithms along with many examples and applications can be found in two monographs by Petkov [12], [18].

The graphical structural counterpart of (3) is given in Fig. 1. The cells are consecutively arranged in a linear order. Each cell is connected with its left and right next neighbors. Note that the systolic computation of (3) requires a toroidal topology of the linear array i.e., a ring connectivity. At clock step 0, a sequence x of p data elements is distributed over p PEs. This sequence will stay fixed in the following process. Initially, a copy $\hat{x}$ of the array x is made. The array $\hat{x}$ is shifted and the processing on all cells is performed subsequently, clock step by clock step. Fig. 1 illustrates the state of the systolic computation after three clock steps.

The small black boxes stand for the DEs, the function of which is to shift the data element $\hat{x}_i$ in one clock step from cell # i to cell # $(i + 1)$. Subsequently, the PEs perform the numerical computation of the function f, i.e., they combine the elements of the fixed array x with the elements of the shifted array $\hat{x}$. The result in the ith cell is added to the resulting data element y_i of the array y such that, after $p - 1$ steps, the resulting data elements are completely computed and are distributed over all processing elements. In terms of the molecular dynamics example, each particle coordinate

The mapping of the abstract automaton model onto a real parallel computer appears straightforward: The PEs are identified with the processors of the parallel machine, the DEs symbolize registers or memory locations exchanging data via the interprocessor communication network. A connectivity of the parallel system is required that allows the embedding of a logical ring to realize the communication pattern of the systolic automaton.

The explicit systolic realization of (3) is given in Algorithm 1 for the generic case $n = p$.

Algorithm 1. Conventional systolic algorithm.

```
foreach processor $i = 1 : p \in$ systolic ring
    $\hat{x}_i^0 = x_i$
    for $j = 1 : p - 1$
        $\hat{x}_i^j = \hat{x}_{i \bmod p+1}^{j-1}$
        $\hat{y}_i = \hat{y}_i + f(\hat{x}_i^0, \hat{x}_i^j)$
    end for
end foreach
```

3.1 Communication Model

For further evaluation and comparison of complexities between systolic and various hyper-systolic methods, we introduce a communication complexity model.

The time required for a processor to communicate a message of m words to a neighbor processor node is modeled as $t_l + mt_t$, where t_l is the start-up time that is assumed to be independent of the message length and t_t is the data transmission time per word.

For the systolic computation of (3), $p - 1$ systolic steps are performed, where, in the generic case, each processor sends exactly one word to its neighbor. Thus, the total number of words to send is a number order p^2, one word per processor in one systolic step, and a total communication time of

$$T = (p - 1)(t_l + t_t) \tag{6}$$

is found.[2]

2. One can improve on the scaling of the systolic realization by taking into account the symmetries. The computations then can be reduced to $p(p - 1)/2$ processing operations, but $p(p + 1)$ communication events have to be per-

For the case $n > p$, the array will be partitioned into $\frac{n}{p}$ parts. One can organize the computation such that, successively, all required computations between all n data elements can be constructed, according to (5). Each inner loop for each array is carried out as described for the generic case $n = p$; additionally, combinations between rows must be computed. In each systolic step, a given processor has to send $\frac{n}{p}$ words to its neighbor. As communication is pipelined, the total communication time thus turns out to be

$$T = (p-1)\left(t_l + \frac{n}{p} t_t\right). \tag{7}$$

4 Hyper-Systolic Algorithm

The main focus of this work is on the reduction of the total time expense for the interprocessor communication in the parallel computation of (3). We will show for the generic case, (3), that for the communicational part a complexity of $O(p^{\frac{3}{2}})$ can be achieved by reorganization of the data-movement. Going to the case $n > p$, the conventional complexity can be reduced from $O(np)$ to $O(np^{\frac{1}{2}})$ for the hyper-systolic algorithm. The amount of processing operations is not altered compared to conventional methods.

4.1 Motivation

Consider the matrix S, called systolic matrix, which explicitly shows the data elements of the shift-array $\hat{x}_t$, (4), for the clock steps $t = 0, 1, 2, \ldots, p-1$, as delivered successively in the systolic implementation, Section 3:

$$S = \left(\begin{array}{ccccccc} 1 & 2 & 3 & \ldots & & p \\ 2 & 3 & 4 & \ldots & p & 1 \\ 3 & 4 & 5 & \ldots & 1 & 2 \\ \cdot & & & & & \cdot \\ \cdot & & & & & \cdot \\ \cdot & & & & & \cdot \\ p & 1 & 2 & \ldots & p-2 & p-1 \end{array}\right) \begin{array}{l} t=0 \\ t=1 \\ t=2 \\ t=3 \\ \\ \\ t=p-1. \end{array} \tag{8}$$

We observe in (8) that all combinations of two different elements within the columns of S actually occur p times, if each shifted array $\hat{x}_t$ is stored for $t = 0, 1, 2, \ldots, p-1$ and combinations between all the rows are allowed. Note that storing the shifted arrays is not required in the systolic concept with only two arrays, *one* fixed and *one* moving array. Here, as time proceeds, an increasing number of "resident" arrays is stored, with only one array in movement.

The matrix S shows that a p-fold redundancy of pairings is encountered if all shifted arrays are stored. It appears natural to reduce this redundancy of pairings and, thus, to save on interprocessor communication by storing some of the arrays $\hat{x}_t$ only, for a suitably selected subset of time steps t.

To illustrate the idea, let us consider a simple example for $p = 16$; from matrix S in (8) we extract only five rows:

$$\mathcal{H} = \left(\begin{array}{cccccccccccccccc} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 \\ 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 1 \\ 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 1 & 2 & 3 \\ 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 & 16 & 1 & 2 & 3 & 4 & 5 \end{array}\right) \tag{9}$$

We call this reduced matrix "hyper-systolic" matrix $\mathcal{H}$. From this matrix, it is easy to see that, indeed, all required pairs come together within the processors: Let us, e.g., consider combinations with element # 1: In the first column, the combinations {1, 2}, {1, 4}, {1, 6}, and {1, 10}, in the 16th column, {1, 3}, {1, 5}, {1, 9}, and {1, 16}, in the 14th column, we have {1, 15}, {1, 14}, and {1, 7}, in the 12th column, {1, 12} and {1, 13}, and, in the eighth column, the combinations {1, 11} and {1, 8} can be found. As the system is toroidally closed, it is evident that all pairing are present. As a result, we need only four shifts and five arrays to be stored, to get all pairs done. In the following, it will be important that the structure presented implies a homogeneous load of all processors in each step of the hyper-systolic computation, as was the case in the systolic computation.

We note, further, that in order to realize (3), we cannot just add the outcome of the computations to *one* resulting array: For every row $\hat{x}_t$ to be stored, a separate, intermediate array $\hat{y}_t$ is required. In a process that involves the inverse sequence of shifts, a moving "collector" array is shifted back, while step by step, the results in the arrays $\hat{y}_t$ are accumulated. Finally, y contains the desired results.

Therefore, for this small example, in total, eight shift-operations are required. In the standard systolic process, 15 shifts of the moving array must be carried out, while, for Half-Orrery, 17 shifts are to be performed [15].

4.2 Hyper-Systolic Breviary

Next, we give a general prescription to implement (3) according to the ideas exposed in the last section. The new algorithm is called *hyper-systolic*, the name referring to the particular features of the underlying communicational structure.

1) For a given array x of length p, k replicas are generated by shifting the array x k times by a stride a_t and storing the resulting arrays as $\hat{x}_t$, $1 \le t \le k$. The variable strides a_t of the shifts are chosen such that
   a) all pairs of data elements (taken within columns of $\mathcal{H}$) occur at least once and the number of equal pairings is minimized,
   b) the number of shifts k is minimized, under the constraint that the strides a_t can be realized efficiently by the given parallel computer's connectivity.
2) The $p(p-1)/2$ results y_i, according to (3), are successively computed and are added to k arrays $\hat{y}_t$.
3) Finally, applying the inverse shift sequence, a collector array y is shifted back by the inverse sequence of shifts, while the intermediate results in the array $\hat{y}_t$ is added to y in step $k - t$.

4.3 Graphical Representation

The concept of traditional systolic automata models can be extended to the hyper-systolic parallel computing structure. For the graphical representation of the latter, we use the objects introduced in Section 3. Fig. 2 shows the ith hyper-systolic cell unit. The DEs again represent input and output. Additionally, we have drawn open rectangles to symbolize

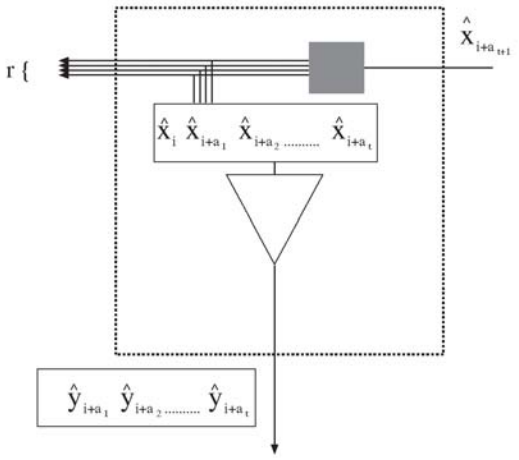
r {
$\hat{x}_{i+a_{t+1}}$
$\hat{x}_i$ $\hat{x}_{i+a_1}$ $\hat{x}_{i+a_2}$ .... $\hat{x}_{i+a_t}$
$\hat{y}_{i+a_1}$ $\hat{y}_{i+a_2}$ .... $\hat{y}_{i+a_t}$

Fig. 2. Processing element (PE) (open triangle), delay element (DE) (black square), and store elements (SE) (open squares), arranged in a hyper-systolic automaton cell (large dotted square). The element $\hat{x}_i$ resides stationary. In each clock step *t*, the DE delivers a new data element from location $[(i + a_t - 1),\text{mod}_p + 1]$ to be stored in the local SE and transmitted further to the next DE. Subsequently, the computation is performed on elements stored in the SE. The results are added to memory locations $[\hat{y}_{(i+a_t-1)\text{mod}_p+1}]$. Note that each DE is equipped with *r* connectors to account for *r* different strides a_t. Second, the results $\hat{y}_{[i+a_t-1\text{mod}_p+1]}$ are step by step added to the array **y** that is shifted by the inverse sequence of strides.

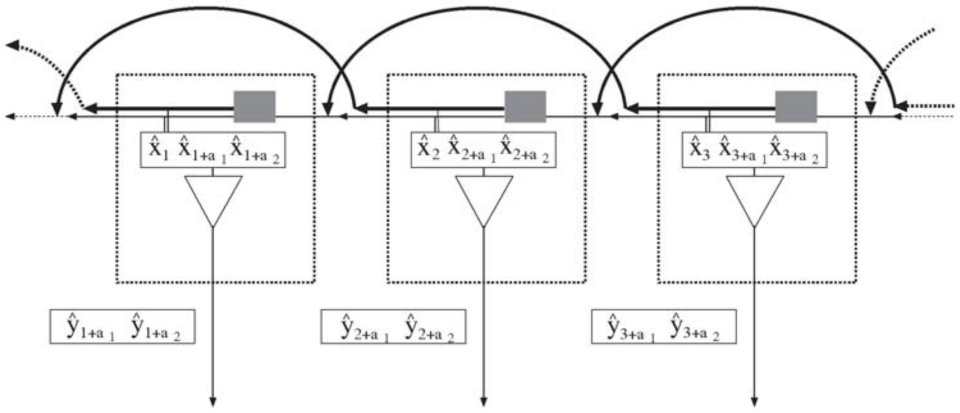
$\hat{x}_1$ $\hat{x}_{1+a_1}$ $\hat{x}_{1+a_2}$
$\hat{x}_2$ $\hat{x}_{2+a_1}$ $\hat{x}_{2+a_2}$
$\hat{x}_3$ $\hat{x}_{3+a_1}$ $\hat{x}_{3+a_2}$
$\hat{y}_{1+a_1}$ $\hat{y}_{1+a_2}$
$\hat{y}_{2+a_1}$ $\hat{y}_{2+a_2}$
$\hat{y}_{3+a_1}$ $\hat{y}_{3+a_2}$

Fig. 3. Three abstract automaton cells as part of a hyper-systolic array with *r* = 2 The data location on a part of the hyper-systolic array is shown after the second systolic cycle.

$\hat{y}_t$, with $1 \leq t \leq k$. The DEs now have one input and *r* output connectors, one separate connector for each required shift width a_t.

Again, the cells are arranged as a uniform grid; in addition to the next-neighbor connectivity, the *r* new nonlocal regular connections between the processors are indicated in Fig. 3. They correspond to the shifts of data elements by strides a_t. According to the number *r* of different strides a_t, *r* direct connections would be optimal. From the point of view of the original one-dimensional systolic array structure—due to these additional communication connections—the (one-dimensional) space of processing elements has acquired topologically nontrivial interconnections. Fast regular interprocessor data transfer to distant cells can be achieved in one clock step via these interconnections that are used as shortcuts. Thus, the characterization of our new algorithm as *hyper-systolic*. We emphasize that hyper-systolic concept. From the uniform structure of the hyper-systolic automaton model and the regularity of the interprocessor communication, it is evident that the PEs can perform functions of equal load in each hyper-systolic step. The amount of processing operations that is equal for each processor increases with an increasing number *t* of stored arrays $\hat{y}_{t'}, 1 \leq t' \leq t$.

Fig. 3 represents the graphical structural counterpart of (3) for the hyper-systolic algorithm. At clock step *t* = 0, a sequence x of *p* data elements is distributed over *p* processing elements. In every clock step *t* = 0, 1, 2, …, *k*, a copy $\hat{x}_t$ of the array $\hat{x}_{t-1}$ is generated, and, in the next clock step, the array is shifted by a stride a_t. Subsequently, the new array $\hat{x}_t$ is stored in the SEs. Then, the processing can be done by the PEs involving ever more of the stored arrays for increasing *t*. Fig. 3 shows the situation after two clock steps. Note, however, that only *one* array is moving just as

DOCKET ALARM

Explore Litigation Insights

Docket Alarm provides insights to develop a more informed litigation strategy and the peace of mind of knowing you're on top of things.

Real-Time Litigation Alerts

Keep your litigation team up-to-date with **real-time alerts** and advanced team management tools built for the enterprise, all while greatly reducing PACER spend.

Our comprehensive service means we can handle Federal, State, and Administrative courts across the country.

Advanced Docket Research

With over 230 million records, Docket Alarm's cloud-native docket research platform finds what other services can't. Coverage includes Federal, State, plus PTAB, TTAB, ITC and NLRB decisions, all in one place.

Identify arguments that have been successful in the past with full text, pinpoint searching. Link to case law cited within any court document via Fastcase.

Analytics At Your Fingertips

Learn what happened the last time a particular judge, opposing counsel or company faced cases similar to yours.

Advanced out-of-the-box PTAB and TTAB analytics are always at your fingertips.

API

Docket Alarm offers a powerful API (application programming interface) to developers that want to integrate case filings into their apps.

LAW FIRMS

Build custom dashboards for your attorneys and clients with live data direct from the court.

Automate many repetitive legal tasks like conflict checks, document management, and marketing.

FINANCIAL INSTITUTIONS

Litigation and bankruptcy checks for companies and debtors.

E-DISCOVERY AND LEGAL VENDORS

Sync your system to PACER to automate legal marketing.