Jens Volkert (Ed.)

Parallel Computation

Second International ACPC Conference
Gmunden, Austria, October 4-6, 1993
Proceedings

Springer-Verlag
Berlin Heidelberg New York
London Paris Tokyo
Hong Kong Barcelona
Budapest

Series Editors

Gerhard Goos
Universität Karlsruhe
Postfach 69 80
Vincenz-Priessnitz-Straße 1
D-76131 Karlsruhe, Germany

Juris Hartmanis
Cornell University
Department of Computer Science
4130 Upson Hall
Ithaca, NY 14853, USA

Volume Editors

Jens Volkert
Institut für Informatik, Johannes Kepler Universität Linz
Altenbergerstr. 69, A-4040 Linz, Austria

CR Subject Classification (1991): D.1.3, D.2.6, F.2.1-2, D.3.2

ISBN 3-540-57314-3 Springer-Verlag Berlin Heidelberg New York
ISBN 0-387-57314-3 Springer-Verlag New York Berlin Heidelberg

Typesetting: Camera-ready by author
Printing and binding: Druckhaus Beltz, Hemsbach/Bergstr.
45/3140-543210 - Printed on acid-free paper

Preface

The Austrian Center for Parallel Computation (ACPC) is a co-operative research organization founded in 1989 to promote research and education in the field of software for parallel computer systems. The areas in which the ACPC is active include algorithms, languages, compilers, programming environments, and applications for parallel and high-performance computing systems.

The partner institutions of the ACPC come from the University of Vienna, the Technical University of Vienna, and the Universities of Linz and Salzburg. They carry out joint research projects, share a pool of hardware resources, and offer a joint curriculum in Parallel Computation for graduate and postgraduate students. In addition, an international conference is organized every other year. The Third International Conference of the ACPC will take place in Vienna in 1995.

The *Second International Conference of the ACPC* took place in Gmunden, Austria, from October 4 to October 6, 1993. The conference attracted many participants from around the world. Authors from 17 countries submitted 44 papers, from which 15 were selected and presented at the conference. In addition, 4 distinguished researchers presented invited papers. The papers from these presentations are contained in this proceedings volume.

The organization of the conference was the result of the dedicated work of a large number of individuals, not all of whom can be mentioned here. I would like, in particular, to acknowledge the efforts made by the members of the Program Committee and the referees. The organizational and administrative support from Alfred Spalt, Romana Schiller, Irmgard Husinsky and Bernhard Knaus was exceptionally valuable.

Finally, we gratefully acknowledge the following organizations which have supported the conference:

The Austrian Ministry for Science and Research
The Austrian Science Foundation (FWF)
The Governor of the Province of Upper Austria
The Mayor of Gmunden
Amt der O.Ö. Landesregierung
Kammer der Gewerblichen Wirtschaft für Oberösterreich
Linzer Hochschulfonds
Vereinigung Österreichischer Industrieller Landesgruppe O.Ö

Bacher Systems EDV GmbH (Vienna, A)
CRAY Research GmbH (Munich, D)
Digital Equipment Corp. (Vienna, A)
GE.PAR.D, Ges. f. Parallele Datenverarbeitung GmbH (Vienna, A)
IBM (Vienna, A)
Intel Corporation Ltd. (Swindon, U.K)
MasPar Computergesellschaft (Neubiberg, D)
nCUBE Deutschland GmbH (Munich, D)
Siemens Nixdorf Informationssysteme GmbH (Vienna, A)
Silicon Graphics GmbH (Vienna, A)

Linz, August 1993 Jens Volkert

Contents

Programming Environments

High-Performance Computing on a Honeycomb Architecture *

Borut Robič and Jurij Šilc

Institute Jožef Stefan, Laboratory for Computer Architectures
Jamova 39, 61111 Ljubljana, Slovenia
e-mail: borut.robic@ijs.si

Abstract. We explore time and space optimization problems involved in the mapping of parallel algorithms onto a honeycomb architecture. When a well-known mapping is used, mapped algorithms generally exhibit execution slow-down and require too large area. We design several optimization techniques and enhance the mapping process. Experimental results show more than 50 % saving in processor resources and 30 % saving in execution time, on average. Since computing performances are improved, also the applicability of the honeycomb architecture is wider.

1 Introduction

In sequential computation special purpose machines have no major advantage over general purpose machines, since the later can perform the same functions almost as fast as the former. This means that, in sequential computation, the universality can also be made efficient. In parallel computing, however, the question of whether efficient universality can be found has no clear answer yet, in spite of the fact that several models of realistic machines have been proposed and many parallel computers have been built. For example, systolic and wavefront VLSI processor arrays proved efficient in execution of decision-free algorithms which are characterized by having a dataflow pattern and computation order independent of the data values [1]. Yet, these algorithmically specialized arrays are not suitable for execution of many other algorithms that do not exhibit such a high regularity.

For this reason, a universal processor array, i.e. an array capable of executing arbitrary algorithms, was suggested in [2]. The array consists of cells (processing elements) which are arranged in rows, each pair of rows being separated by the communication bus (Fig. 1a). The bus enables cells to communicate with a host computer for algorithm down-loading and data I/O. Each cell has six immediate neighbors, and is connected to them by point-to-point links. The computation is data-driven, i.e. each cell is capable of testing for the presence of its operands and executing only the instructions for which all the necessary operands have arrived. The name *honeycomb* is suggested by Fig. 1b, where only cells are depicted. An

* This research is supported by the Ministry of Science and Technology of the Republic of Slovenia under grant P2-5092-106/93.

arbitrary parallel algorithm is written in a dataflow language, and mapped onto honeycomb architecture using a four-stage process. During the first stage the program is translated to a dataflow graph (DFG). The second stage, referred to as DFG-*mapping*, maps the DFG onto a potentially unbounded honeycomb. The third stage partitions the array into chips while the last stage down-loads the tasks to the cells of the array. This idea of direct mapping an arbitrary algorithm on a hexagonally connected VLSI array was introduced in [3] and improved in [4] where medium grained parallelism was used.

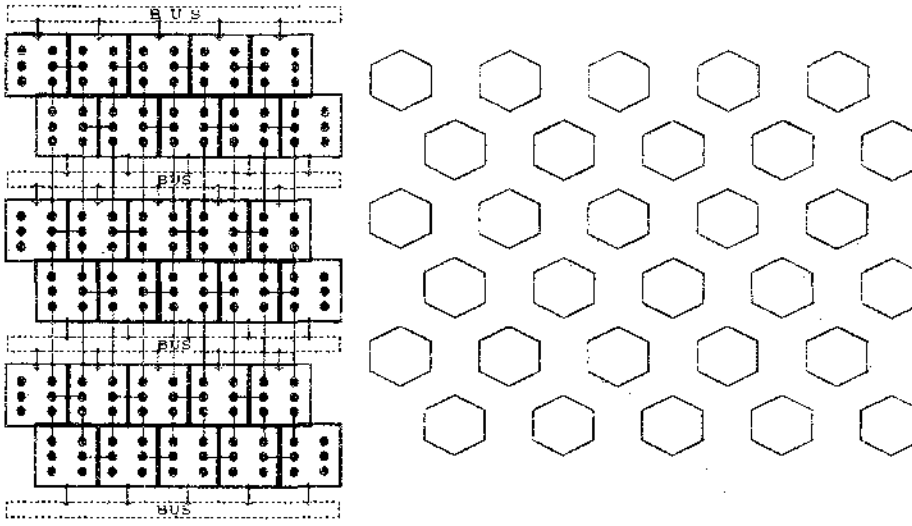

Fig. 1. Honeycomb architecture: (a) Implementation, (b) Scheme.

However, the universality of the array is not as attractive as it seems at first sight because not all mapped algorithms use the array efficiently. In particular, some may require a too large chip area, others may not execute fast enough. The mapping process may be improved by changing array architecture, thus making it more amenable to the mapping. In [5], for example, blocks of cells (characterized by tighter coupling) are created, and array connectivity is improved by addition of wires and switching elements between blocks. In [6] cells contain up to 16 nodes and two communication links connect every two adjacent cells. In this paper, we improve computing performance of the honeycomb by adding the optimization process to the DFG-mapping while keeping the basic cell architecture.

2 Mapping

The mapping process starts with the translation of a program, written in the dataflow language VAL, to a DFG. The translator was built utilizing Unix tools Lex and YACC [7]. Fig. 2 shows a simple **if-then-else** VAL program [8] and its translation to the DFG.

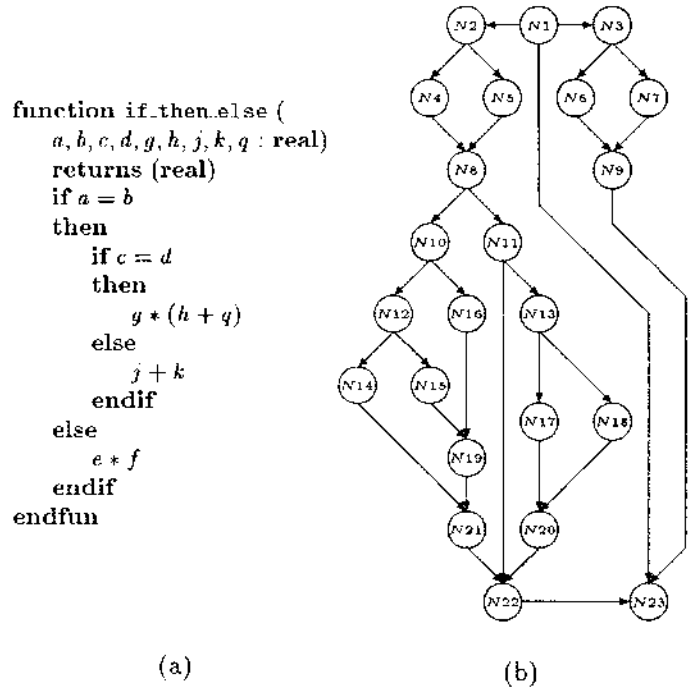

Fig. 2. if-then-else: (a) VAL program. (b) Its DFG.

2.1 DFG-mapping

The DFG is mapped onto theoretically unbounded honeycomb during the DFG-mapping stage. At this point, some fundamental constraints, imposed by technological characteristics of the architecture, are to be obeyed. In particular, each cell is connected to six adjacent cells, communicates with them simultaneously, and is capable of performing computational and routing tasks concurrently.

The DFG-mapping consists of three steps, referred to as layer construction, placement of the layers, and path construction.

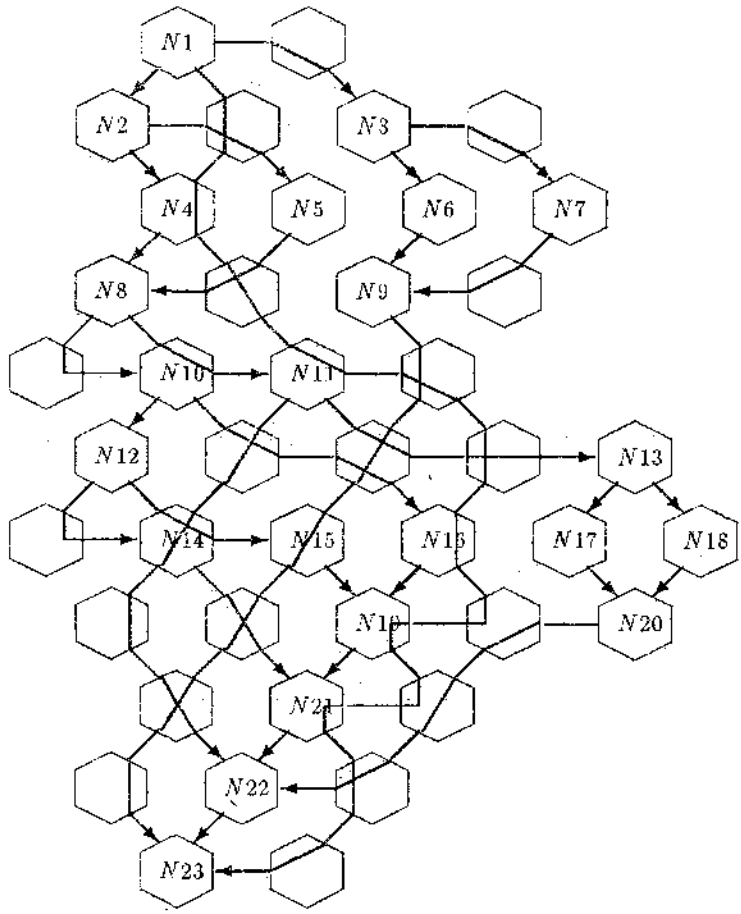

Fig. 3. if-then-else: Direct mapping.

During the first step, the set of vertices of DFG is partitioned into *layers*. Layer 0 consists of vertices accepting only external input data. A vertex is in layer $i > 0$, if one of its inputs comes from a vertex at layer $i - 1$, and none come from layers $k \geq i$. In order to construct layers, arcs that close loops are first detected applying depth-first traversal of DFG. These backward arcs are temporarily cut. This results is an acyclic subgraph of DFG whose vertices are then partitioned into layers according to labels obtained by topological sort. In the second step, each of the layers is placed on its own row of cells in the array. The placement is performed according to the associated labels, i.e. layer i is placed onto the i-th row. Next, the vertices in each row are permuted so that the paths which still remain to be built in the next step are expected to be short.

Vertices which are to be connected are arranged as much as possible one under another to allow the paths between them to be as vertical as possible. The final step of DFG-mapping determines how the arcs of DFG are represented by paths in the array. The paths are built in a heuristic piece-meal fashion moving from one cell to another. At each cell, a single communication link is added to one or more incomplete paths which go through it. The processing of the incomplete paths is performed according to their dynamically assigned priorities. This makes it possible to choose a suitable routing heuristic. However, situations still develop where a large number of paths have to be routed through a small area on the array. In such cases only the last decision in the routing process is changed instead of performing an extensive backtracking to search for other routings. If this does not solve the problem, a row or column of cells is simply added to the array. The result of DFG-mapping for `if-then-else` example is depicted on Fig. 3.

Clearly, the depth-first traversal of the DFG cuts the loops at the last possible moment and thus constructs many layers. Since each layer is placed onto its own row the mapped DFG is overstretched downwards across the array. Moreover, each arc of DFG that has been temporarily cut connects vertices whose label-difference is proportional to the length of the loop. The corresponding path in the array may thus be very long.

In summary, the mapped graph is exaggeratedly stretched downwards across the hosting array. It generally has vertical paths some of which connect cells of very distant rows. As a consequence, such DFG mappings exhibit two disadvantages. First of all, since time complexity of routing operation is not negligible [9], execution slow-down may occur as a result of remote communication between some cells (especially for cyclic DFGs). Secondly, DFG mappings may require a too large area.

2.2 Optimization

To evaluate the applicability of the proposed array, and to use it efficiently, one has to be able to map parallel algorithms onto it as optimally as possible. Hence, an optimization stage has to be designed.

Let μ be a mapping and G a DFG. The cost of a mapped graph $M = \mu(G)$ is $E(M) = \sum_p \ell(p)$, where the sum is taken over all paths of M which are mappings of arcs in G, and $\ell(p)$ is the number of cells along each such path without endpoints. The object is to find mapping μ^* which minimizes the cost E. Instead of a direct searching for μ^* an existing mapping μ_0 is fixed, and an additional transformation τ^* of $\mu_0(G)$ is searched to minimize the cost by a composed mapping $\mu = \tau^* \circ \mu_0$. For μ_0 the mapping of DFG-mapping from [3, 4] is used. Fig. 4 shows the result of applying such a transformation to the mapping on Fig. 3. The cost was reduced from $E = 40$ to $E = 9$.

Given some mapping M of a DFG we say that a transformation of M is *regular* if it moves a single vertex of M to an adjacent cell and reroutes all possible paths [10]. Generally, there are several regular transformations applicable to M. The neighborhood $\mathcal{N}(M)$ of M consists of all mappings obtained from

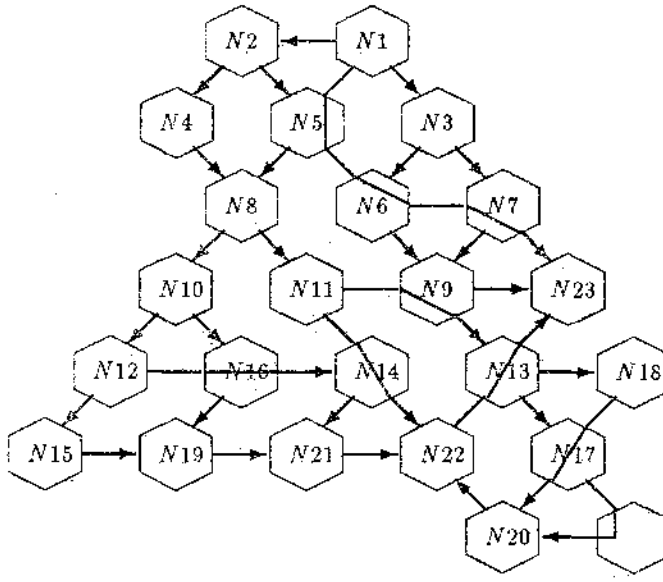

Fig. 4. `if-then-else`: "Optimal" mapping.

M by regular transformations. *Local search* $\mathcal{LS}$, which is a general method for designing approximation algorithms [11], starts at initial mapping $M = \mu_0(G)$ and searches for a mapping M' in $\mathcal{N}(M)$, so that $E(M') < E(M)$. As long as an improved mapping M' exists, the algorithm adopts it and repeats the search from it. When a local minimum is reached, the algorithm stops. Fig. 5 shows the result of applying $\mathcal{LS}$ to the `if-then-else` example. The cost was reduced from $E = 40$ to $E = 27$.

Probabilistic algorithms. Local search is a fast method but it generally gets trapped in a local optimum. To escape from local optima several probabilistic algorithms have been proposed in the past, including *simulated annealing* $\mathcal{SA}$ [12], *threshold accepting* $\mathcal{TA}$ [13], and *quenching* $\mathcal{Q}$ [14].

$\mathcal{SA}$ has become a popular tool for solving a wide class of combinatorial optimization problems [12]. In our case the algorithm runs as follows. As in local search it starts at initial mapping $\mu_0(G)$. After some mapping M has been reached, another mapping M' is *randomly* chosen from the neighborhood $\mathcal{N}(M)$. If M' is better than M, it is accepted. If it is worse, however, it is accepted only with a certain probability. This enables the algorithm to escape from trapping into a local minimum. The acceptance probability depends on a time-dependent parameter T and on the decrease in E. Since T is slowly

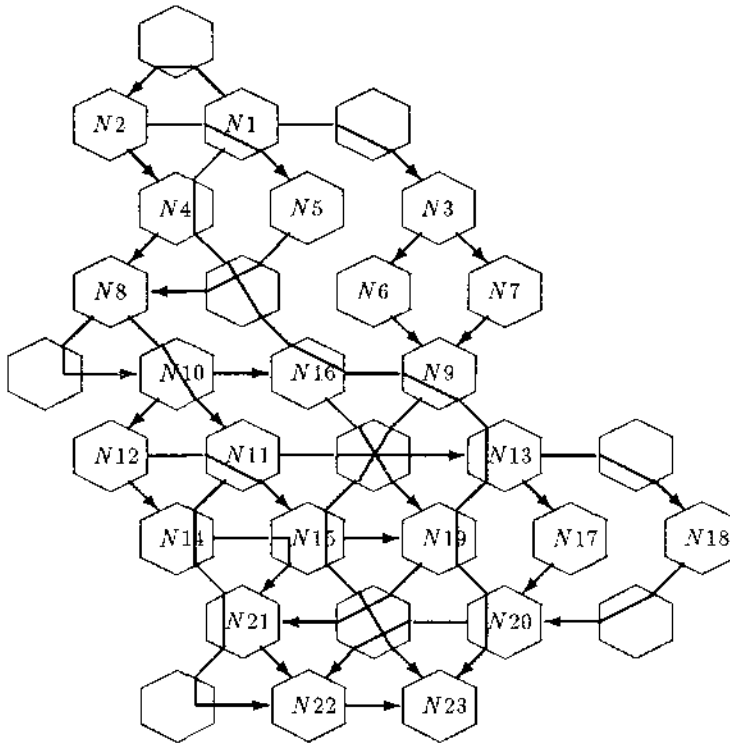

Fig. 5. `if-then-else`: $\mathcal{LS}$ mapping.

lowered during the running time, also the acceptance probability is lowered, making "bad" acceptances more and more improbable. The algorithm has been implemented with the following annealing schedule: T is lowered by a constant value (*temperature function*), at each T changes are attempted until a number of acceptances have occurred (*number of repetitions*), and the algorithm stops when sufficiently few acceptances are occurring at successive temperatures (*stopping criteria*). Fig. 6 shows the result of applying $\mathcal{SA}$ to the `if-then-else` example where the cost was reduced to $E = 24$.

A similar yet simpler algorithm is threshold accepting $\mathcal{TA}$. It was reported in [13] that the algorithm yields better results than $\mathcal{SA}$. Moreover, it was observed that with the same number of steps $\mathcal{TA}$ solutions were on the average better than $\mathcal{SA}$ solutions.

In our case, while $\mathcal{SA}$ accepts worse mappings with certain probabilities, $\mathcal{TA}$ accepts every new mapping M' which is "not much worse" than the old mapping M. To be more specific, after some mapping M has been reached, another mapping M' is randomly chosen from the neighborhood $\mathcal{N}(M)$. If M' is

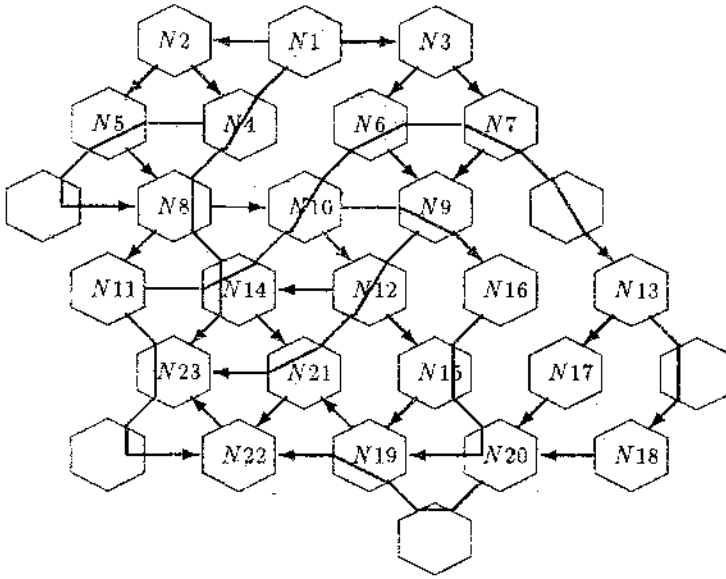

Fig. 6. `if-then-else`: $\mathcal{SA}$ mapping.

better than M, i.e. if $E(M') < E(M)$, it is accepted. If it is worse, it is accepted only if $E(M') - E(M) < t$, for some *threshold* $t > 0$. This enables the algorithm to escape from trapping into a local minimum. However, the acceptances of worse mappings become more and more rare because the threshold t is slowly lowered during the running time.

Deterministic modifications. The choice of a "good" neighborhood for the problem is usually guided by intuition. There is a trade-off between large and small neighborhoods, since large neighborhoods may provide better local optima but generally take longer to search [12]. If the neighborhoods are too small, the process may not be able to move around quickly enough to reach the minimum in a reasonable time. On the other hand, if the neighborhoods are too large, the process essentially performs a random search. In our case, the neighborhood $\mathcal{N}(M)$ as defined above (by regular transformations of M) is an adequate compromise. However, the neighborhood structure is not the only aspect that is free to be chosen so that the convergence speed of the algorithms is improved. In this paper we design two modifications of $\mathcal{SA}$ and $\mathcal{TA}$, which are still capable of escaping local minima but have improved convergence speed. Informally, after some mapping M has been reached, *deterministic* selection of the next mapping $M' \in \mathcal{N}(M)$ is performed instead of a random one. This selection is made by using "the most promising" transformation of M, i.e. such a regular transforma-

tion γ, that the greatest decrease in the cost E is expected after applying γ on M. We call γ the *guiding* regular transformation, and the modified algorithms *Guided Simulated Annealing* $\mathcal{GSA}$ and *Guided Threshold Accepting* $\mathcal{GTA}$.

$\mathcal{GSA}$ algorithm:

let M be initial mapping $\mu_0(G)$
choose an initial temperature $T > 0$
loop: construct guiding transformation γ
  $M' := \gamma(M)$
  $\Delta E := E(M') - E(M)$
  if $\Delta E < 0$
    then $M := M'$
    else if $random(0,1) \leq e^{-\Delta E/T}$
      then $M := M'$
  if a long time no decrease in E
    or too many iterations
    then lower T
  if some time no change in E
    then stop
goto loop.

$\mathcal{GTA}$ algorithm:

let M be initial mapping $\mu_0(G)$
choose an initial threshold $t > 0$
loop: construct guiding transformation γ
  $M' := \gamma(M)$
  $\Delta E := E(M') - E(M)$
  if $\Delta E < t$
    then $M := M'$
  if a long time no decrease in E
    or too many iterations
    then lower t
  if some time no change in E
    then stop
goto loop.

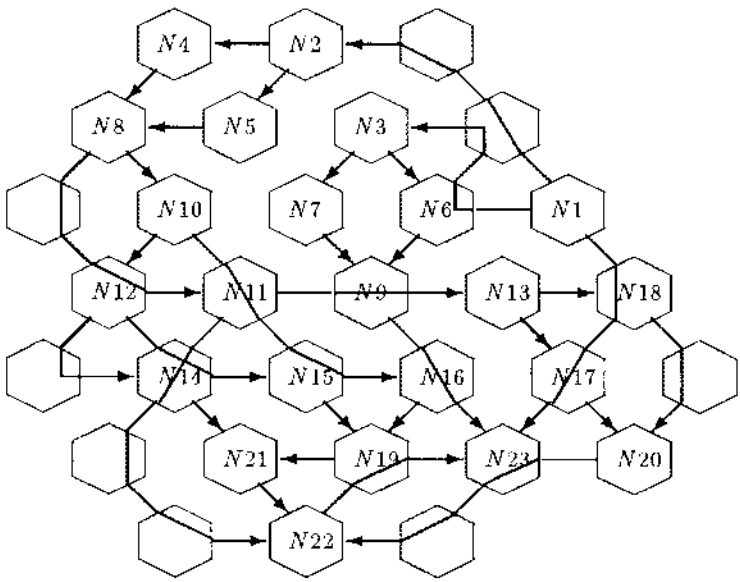

Fig. 7. `if-then-else`: $\mathcal{GSA}$ mapping.

Heuristic construction of the guiding regular transformation of M proceeds as follows. For each vertex v of M and each path p incident to v a physical force of attraction is assigned to v, having its magnitude proportional to $\ell(p)$ and direction from v to the next processor on p. Let $\overrightarrow{F}(v)$ be the vector sum of all such forces acting upon v, $F(v)$ its magnitude, and $d(v)$ one of six directions from v to adjacent processors which is closest to $\overrightarrow{F}(v)$. Now, let $v_1, v_2, \ldots$ be ordering of vertices of M, such that $F(v_i) \geq F(v_{i+1})$, for $i = 1, 2, \ldots$. Then the guiding regular transformation γ is described by a pair $(v_k, d(v_k))$, where k is the smallest index such that v_k can be moved in direction $d(v_k)$ to the adjacent processor rerouting all incident paths.

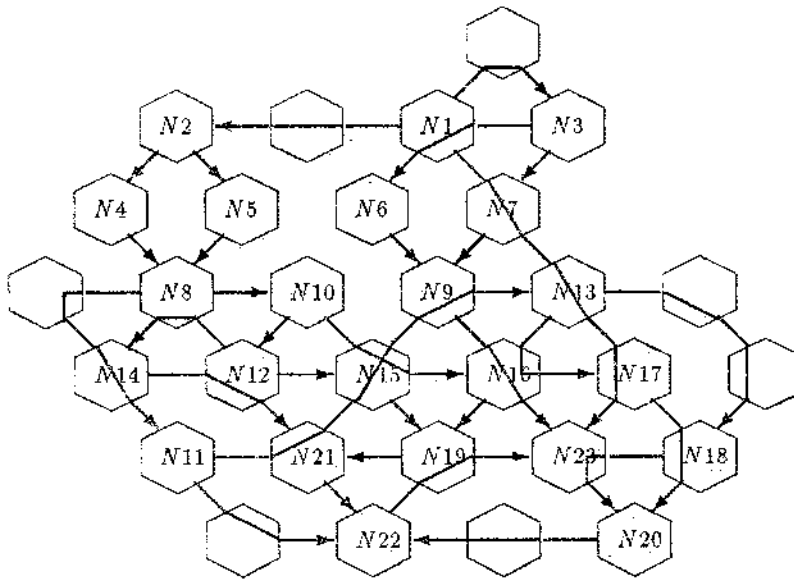

Fig. 8. `if-then-else`: $\mathcal{GTA}$ mapping.

Fig. 7 and Fig. 8 show the results of applying $\mathcal{GSA}$ and $\mathcal{GTA}$ to the `if-then-else` example, respectively. $\mathcal{GTA}$ returned $E = 23$ while $\mathcal{GSA}$ returned $E = 21$. Moreover, these results were obtained 3 to 5-times quicker compared to $\mathcal{SA}$.

3 Experimental Results

We obtained several experimental results utilizing *GraphCompactor*, a tool which we designed to perform optimization with algorithms $\mathcal{LS}$, $\mathcal{GTA}$, $\mathcal{GSA}$, and $\mathcal{SA}$ [15]. We considered fourteen DFGs from [3, 9, 16]. For each DFG probabilistic

algorithms have been run 3 - 5 times with different annealing schedules. Algorithms $\mathcal{SA}$ and $\mathcal{GSA}$ were implemented with the following annealing schedule. Initial temperature is $T_0 = aE_0$, where E_0 is the cost of initial mapping and $a > 0$. At each temperature T_i changes are attempted until cn of acceptances have occurred, after which T_i is lowered to $T_{i+1} = bT_i$, where n is the number of nodes in graph, $0 < b < 1$, and $c > 0$. Algorithms stop when sufficiently few acceptances are occurring at successive temperatures. The annealing schedule for $\mathcal{GTA}$ algorithm was the same as above with threshold t instead of temperature T. We considered $0.1 \leq a \leq 0.2$, $0.8 \leq b \leq 0.9$, and $0.2 \leq c \leq 0.3$.

The quality of mappings $M = \mu(G)$ was evaluated by the following measures:

- *area* $A(M)$, i.e. the number of cells utilized by mapping M,
- *chip-size* $C(M)$, the number of cells in the smallest rectangle enclosing M,
- *maximum path length* $\widehat{\ell}(M)$, i.e. the number of cells on the longest path of the mapping M,
- *average path length* $\bar{\ell}(M)$, i.e. the average number of cells used to map an arc of G,
- *average execution time* $\omega(M)$ of the mapping M.

Table 1. Average relative improvement of the DFG-mapping (in %).

	area	*chip-size*	*max.path length*	*avrg.path length*	*avrg.execution time*
Algorithm	A	C	$\widehat{\ell}$	$\bar{\ell}$	ω
$\mathcal{LS}$	30	**32**	61	56	**23**
$\mathcal{GTA}$	43	**47**	80	75	**27**
$\mathcal{GSA}$	44	**46**	84	78	**38**
$\mathcal{SA}$	47	**51**	82	81	**36**
"optimal"	51	**61**	87	90	**46**

Table 1 shows average relative improvements of the DFG-mapping which were obtained by algorithms $\mathcal{LS}$, $\mathcal{GTA}$, $\mathcal{GSA}$, and $\mathcal{SA}$. The average reduction in the chip size C (i.e. processor resources) was between 32 % and 51 %, depending on the algorithm used. Note that at the same time the execution time ω was reduced by 23 % to 36 %. Generally, it is difficult to say how close to optimal the improvements are, since there is no efficient way of knowing what the best mapping for a specific DFG is. However, using interactive mode of our *Graph-Compactor* we were able to obtain "optimal" mappings, i.e. mappings having the energy E "close" to 0 (Table 1). Note that the results obtained automatically are near-"optimal".

4 Concluding remarks

Since the honeycomb data-driven array is capable of executing arbitrary algorithms, it can be connected to a host system to function as a slave processor to

handle specialized computationally intensive tasks. In the paper, we have experimentally shown that the efficiency of the array may be considerably improved. As a result, the improved performance of the array makes its universality much more attractive. This makes possible to compare this pure data-driven architecture to other ones, especially the new hybrid data/control-driven architectures [17, 18].

In discussing the problem of mapping parallel algorithms onto honeycomb architecture we have shown that there is a lot of possibilities for improvement of initial mappings produced by the DFG-mapping from [3, 4]. In particular, these mappings exhibit execution slow-down and require too many cells (processing elements). We designed an optimization process and enhanced the DFG-mapping. The execution efficiency was thus considerably improved (more than 50 % saving in honeycomb space and 30 % saving in execution time, on average). The applicability of the array is therefore much wider, since larger parallel algorithms can be mapped onto it. For example, algorithm `Even process` [7] originally used 14 × 16 cells. After optimization, however, it was mapped onto an array consisting of 8 × 8 cells and executed twice faster. Similarly, the area and execution time for algorithm `Newton` [3] were reduced by 62 % and 43 %, respectively. For `Runge-Kutta` [16] these improvements are 50 % and 45 %.

Since electronic components will never be totally reliable, a possibility of production or run-time failures should also be considered. For that reason, all the algorithms mentioned above should take into account the presence of faulty cells in the array. Finaly, to speed up even these new optimization algorithms, the possibility of their parallelization should be analyzed.

References

1. Kung, S.Y.: *VLSI Array Processors*. (Prentice Hall. 1988).
2. Koren, I., Peled, I.: The Concept and Implementation of Data-Driven Array. *IEEE Computer* **20** (July 1987) 102–103.
3. Koren, I., Silberman, G.M.: A Direct Mapping of Algorithms onto VLSI Processing Arrays Based on the Data Flow Approach. *Proc. 1983 Int'l Conference on Parallel Processing*, August 1983, 335–337.
4. Mendelson, B., Silberman, G.M.: An Improved Mapping of Data Flow Programs on a VLSI Array of Processors. *Proc. 1987 Int'l Conference on Parallel Processing*, August 1987, 871–873.
5. Weiss, S., Spillinger, I., Silberman, G.M.: Architectural Improvements for Data-Driven VLSI Processing Arrays. *Proc. 1989 Conference on Functional Programming Languages and Computer Architecture*, September 1989, 243–259.
6. Mendelson, B., Koren, I.: Using Simulated Annealing for Mapping Algorithms onto Data-Driven Arrays. *Proc. 1991 Int'l Conference on Parallel Processing*, August 1991, I-123–I-127.
7. Koren, I., Mendelson, B., Peled, I., Silberman, G.M.: A Data-Driven VLSI Array for Arbitrary Algorithms. *IEEE Computer* **21** (October 1988) 30–43.
8. Mendelson, B., Koren, I.: Estimating the Potential Parallelism and Pipelining of Algorithms for Data Flow Machines. *Journal of Parallel and Distributed Computing* **14** (January 1992) 15–28.