EFFICIENT, SCALABLE ARCHITECTURES FOR LATTICE-GAS COMPUTATIONS

Richard K. Squier
(Thesis)

CS-TR-304-91

June 1991

Efficient, Scalable Architectures for Lattice-Gas Computations

Richard K. Squier

A DISSERTATION
PRESENTED TO THE FACULTY
OF PRINCETON UNIVERSITY
IN CANDIDACY FOR THE DEGREE
OF DOCTOR OF PHILOSOPHY

RECOMMENDED FOR ACCEPTANCE
BY THE DEPARTMENT OF
COMPUTER SCIENCE

JUNE 1991

to my parents

Efficient, Scalable Architectures for Lattice-Gas Computations
—Abstract—

Richard K. Squier
Thesis Advisor - Professor Kenneth Steiglitz

The subject of this dissertation is finding an architecture for two-dimensional cellular automata computations that is verifiably correct, and the fastest and cheapest possible. The motivating problems for this work are large-scale scientific computations; the hardware context is that of application-specific processors attached to general-purpose systems. While the conclusions are derived for a specific class of two-dimensional cellular automata, the so-called ''lattice gasses,'' they are applicable to a wide range of similar problems.

By developing and applying solutions to discrete isoperimetric problems to a pebbling game, upper bounds on throughput for machines computing problems with data dependency graphs based on the undirected graphs for the Hardy-de Pazzis-Pomeau (HPP) and Frisch-Hasslacher-Pomeau (FHP) lattice-gasses are shown. A particular architecture, the ''Wide Serial Architecture'' (WSA), is shown to be within a factor of approximately 6 of the bound for the HPP-like computations, and within a factor of about 4.5 of the bound for the FHP-like computations. Besides the insight they provide into the optimal computational strategy, the methods of solution of the isoperimetric problems, such as the use of the Wulff Crystal, are of interest in their own right.

An analytic study of the least-cost configuration for a multiple-pipeline WSA machine is undertaken. The use of overlap-save method is described, and the efficiency of the WSA architecture using this method is derived. A numerical search is used to find the least cost machine configuration as the problem size is scaled. A slightly super-linear speedup is shown over a moderate range of problem sizes, and the least-cost machine parameters are described.

Finally, a data-embedded, specification-based testing technique for FHP lattice gasses is introduced. The tests consist of limit cycles in the cellular automaton, and their error detection coverage is shown empirically to be good. Their use in software, hardware, and system debugging is described, as well as their use as runtime simulation error detectors. Machine design tradeoffs relating to testability are discussed.

Acknowledgements

First, I want to apologize to the many people that I cannot mention in this brief acknowledgement and to whom I owe a debt of gratitude. Please accept my thanks.

I want to thank my advisor, Kenneth Steiglitz, whose insight, knowledge, advice, humor, friendship, and occasional prodding were invaluable to me. I feel very lucky to have had the opportunity to be associated with him over this time, which now seems too brief. I would like to thank Andrea LaPaugh and Joel Friedman for their patient and careful reading of this work while it was still under revision and in a very disorganized form, and of course, for their valuable comments. Also, I would like to thank Richard Lipton for suggesting I look into pebbling games, Jeane Taylor and Fred Almgren for introducing me to the Wulff Theorem, and David Dobkin for encouragement and advice in using graphics tools.

I would like to thank fellow graduate students Neal Young, for many useful discussions, Mordecai Golin, for reading portions of the thesis and offering valuable comments, and Eleftherios Koutsofios, for essential help in producing graphic displays. Also, many thanks to Adam Buchsbaum, Jeff Westbrook, Steve Kugelmass, Rob Abbott, and Heather Booth, for their helpful discussions.

I would like to thank all the staff of the Computer Science Department for the help I have received from them in many ways. Special thanks to Sharon Rodgers, Grace Kehoe, Melissa Lawson, Winnie Waring, Stephen Beck, and Ed Strong.

Finally, I would like to thank my family and friends who gave me their encouragement and support.

Richard K. Squier
Princeton, New Jersey
17 January 1991

This work was supported in part by NSF Grant MIP-8912100, by U.S. Army Research Office - Durham Contract DAAL03-89-K-0074, by DARPA/ONR Contract N00014-88-K-0459, and by a Von Neumann Fellowship in Supercomputing.

Chapter 4 of this thesis was published previously in short form in *Proceedings of the International Conference on Application Specific Array Processors*, see [61].

Table of Contents

Chapter 1
Introduction

1.1. Introduction

The subject of this dissertation is finding an architecture for two-dimensional lattice-based computations that is verifiably correct, and the fastest and cheapest possible. The motivating problems for this work are large-scale scientific computations; the hardware context is that of application-specific processors attached to general-purpose systems. The results presented here are arrived at by mathematical study of tradeoffs between machine resources and throughput (Chapter 2), by analytic and numerical studies of the tradeoffs between cost of construction and throughput (Chapter 3), and by empirical development of a tradeoff between verifiability of implementation correctness and certain design factors (Chapter 4). While the conclusions are derived for a specific class of two-dimensional cellular automata, the so-called ''lattice gasses,'' they are applicable to a wide range of similar problems, including local iteration algorithms for differential equations.

Two areas of recent development have driven this work: theoretical results on cellular automata properties for complex system modeling, and technology for inexpensive custom VLSI. The theoretical understanding of the properties of certain simple cellular automata has made them interesting candidates for investigating complex systems by simulation study using these automata. There are two basic reasons why they are promising: (1) they avoid certain problems of instability and system definition encountered in the usual numerical approach to modeling complex systems, and (2) the cellular automata computing paradigm is inherently parallel with simple communication which suggests that large-scale parallel machines may be easier to build under this paradigm than others.

The rapid advance of VLSI technology in such directions as silicon compilers; cheaper, denser, and faster chip technology; and rapid prototyping suggests that, in the near future, building algorithm-specific adjunct hardware will become increasingly cost-effective. In general, simulation studies using cellular automata, such as lattice-gas simulations, require the simulation of automata with a very large number of cells for many iterations. This type of computation is generally beyond the capabilities of conventional general-purpose machines, and in the scales proposed, beyond the reach of today's super-computers. However, the developments in VLSI technology may make special-purpose hardware for the simulation of specific cellular automata practical.

It is becoming accepted that no single architecture is equally applicable to all algorithms. In the field of parallel computing, it is not uncommon to find that an algorithm which is efficient on one parallel architecture is inefficient on another. In the application-specific array community there is currently work proving various particular architectural organizations are optimal for the particular algorithm they implement. This dissertation answers some questions of optimality for the lattice-gas algorithm, and suggests directions for future work that might lead to a more general strategy for deriving an optimal architecture from an algorithm. The hope that studying the specific lattice-gas algorithm will teach us about implementing other algorithms affects our problem formulations, and colors the entire work in many ways.

The remainder of this chapter is organized as follows. Section 2 describes the lattice-gas automata; Section 3 gives a summary of results found in the thesis.

1.2. Lattice-Gas Automata

In cellular automata, values associated with nodes in an undirected graph are updated in synchronous steps according to a rule (the rule may, or may not, be uniform for every node) applied locally at each node. The edges of the graph define which nodes are ''neighbors'' of a node, and the update rule determines the new value of the node from the values of its neighbors and itself.

A lattice-gas automaton is a cellular automaton whose neighbor-defining graph is a lattice graph. (For a precise definition of a lattice graph, see [1].) The rule defining a lattice-gas automaton's evolution generally simulates particles traveling along the edges of the neighbor graph. Our version of a lattice gas is based on the HPP and FHP-III models as described in [2, 3] (see figure 1.1). The FHP models are of interest because they have been shown to correspond to the Navier-Stokes equations for fluid flow. The HPP models are interesting partly because of their modelling ability, and partly because the general form of their data dependency graphs conform to the data dependency graphs of a general class of iterated problems. These types of lattice-gas automata (FHP and HPP) consist of a two-dimensional lattice graph and a set of update rules for variables associated with each node in the lattice graph. The lattice is the integer grid, for the HPP gas, or the triangular lattice, for the FHP gas. The edges of the lattice graph connect nearest neighbors in the lattice (this can be extended by also connecting next-nearest neighbors and so forth), resulting in the two-dimensional grid

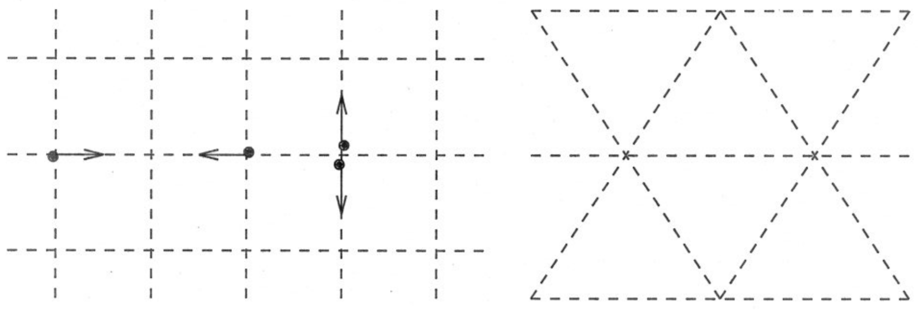


figure 1.1.

The lattice graphs for the HPP lattice gas (left) and the FHP lattice gas (right). Dashed lines represent undirected edges in the graph, and the intersections of the dashed lines represent nodes. On the left of the HPP example, two particles are shown about to collide at a node; on the right, two particles are exiting a node after a similar collision.

in the HPP case, and the triangular lattice graph in the FHP case. These graphs may be embedded on the surface of a torus, or in the plane. At each site each incident edge has an associated variable representing the presence or absence of a unit mass particle with unit velocity directed toward the site's neighbor along that edge. An additional dynamical variable can be used to encode the presence or absence of a unit mass particle with zero velocity positioned at the lattice site (called a ''rest-particle''). With this interpretation of the variables the update rule is designed so that, letting the edges have unit length, the lattice is populated with particles traveling along graph edges and colliding at lattice sites. Figure 1.1 shows a small section of an HPP lattice gas with several particles, and a small section of an FHP lattice graph without particles.

For the lattice gasses the automaton update rule table is called a collision rule set, the initial configuration of states determining a cell's new state is called a collision, the next state entry in the rule table is called the result of the collision, and the combination of a collision and its result is called a collision rule. A particular lattice gas is defined by specifying a collision rule set that gives the results of every possible collision. These rules can be thought of as rules about the action of particles (variables set to one) or equivalently as rules about the action of holes (variables set to zero), as they collide at lattice sites. In an HPP or FHP gas, particle collisions generally conserve momentum and mass, are symmetric with respect to rotation by integer multiples of the basic angle of the lattice, and are symmetric for time reversal; for the FHP-III gas, collisions are also symmetric with respect to hole/particle duality (complementation of the dynamical variables).

A data dependency graph for the evolution of an automaton is a layered graph, each layer being a replica of the set of nodes of the automaton's undirected lattice graph, there being as many layers as there are total steps in the automaton's evolution. Directed edges in the data dependency graph indicate which values were used to compute new values, so arcs go from layer to layer, indicating the state of the automaton at one step is derived from its state at a previous step.

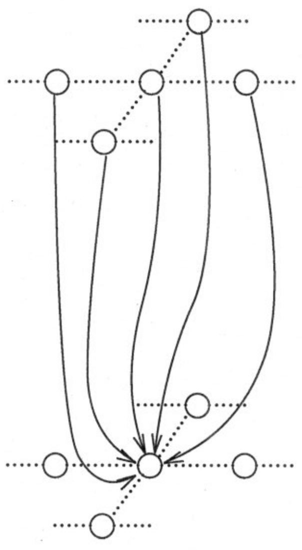


figure 1.2.

The dependency arcs in the data dependency graph for an update of a single in node in an HPP lattice. The dotted lines show the embedding of the HPP lattice graph in the layers of the data dependency graph.

Figure 1.2 shows the dependency arcs from the neighborhood of one node to the updated node in the next layer in a data dependency graph for an HPP simulation. Data dependency graphs for many other iterated computations are similar in that, even though they may not be layered in the strict sense, they may still have a structure similar to layering in that the there is some connected graph implicitly defined that can be used to define a layered subgraph. Since many of our results are based solely on the graph structure of a particular data dependency graph, the results apply to any computation whose data dependency graph contains a similar sub-graph.

1.3. The WSA Architecture

Since a considerable part of this work concerns the WSA architecture, we present here the logical organization of the computation used by that architecture.

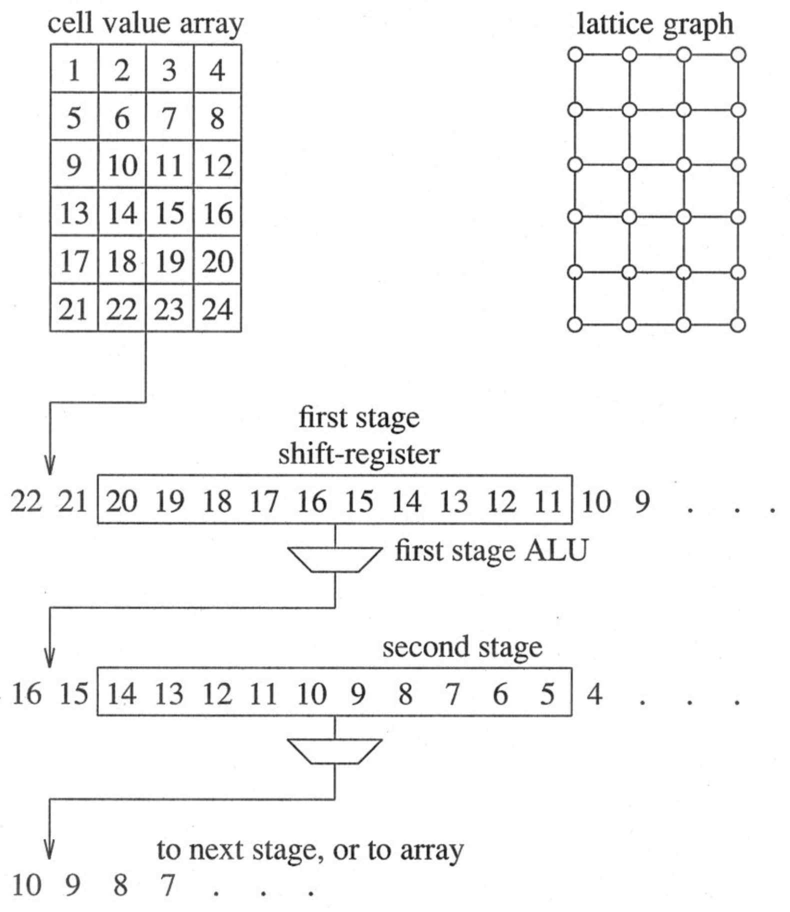


figure 1.3.

The logical organization of computation in the WSA architecture. At the top left is shown an array of site values corresponding to the HPP lattice graph shown at upper right. The lattice site values are passed to the pipeline stages below the array. Here, the first stage is producing updated values for sites 15 and 16, which are being passed on to the second stage. The first stage's shift-register contains site values for sites 11 through 20. The second stage is producing values for sites 9 and 10 which are updated by two automaton evolution steps from the values originally stored in the array.

The WSA architecture updates the state of a lattice-gas automaton by passing the state of the lattice through a pipeline. All the stages of the pipe are identical, and each stage produces lattice site values that are updated one generation from the stage's input values (see figure 1.3). The first stage of the pipe receives the lattice site values in raster scan order from the memory, and passes updated lattice site values in the same raster scan order to the following pipe stage. When the complete array of values has been scanned through the first stage, the entire lattice will have been updated one automaton evolution step by the first stage. As each stage is identical, the output from the second stage will be the lattice updated two steps, and if there are *s* stages in the the pipeline, the lattice values exiting the pipe and being returned to the array will be the values of the lattice updated *s* steps. The ''width'' of the architecture is determined by the number of lattice site values passed along the pipeline from one stage to the next in one global machine tick, or cycle. In figure 1.3, the width is 2 since each stage produces two site values simultaneously. We will go into further details of the WSA architecture, and the LGM-1 machine based on this architecture, in succeeding chapters when the discussion requires it.

1.4. Thesis Outline

Chapter 1 is this introduction. Chapter 2 develops an upper bound on throughput for machines computing problems with data dependency graphs based on the undirected graphs for the HPP and FHP lattice-gasses. Using these bounds, a particular architecture (WSA) is shown to be within a factor of approximately 6 of the bound for the HPP-like computations, and within a factor of about 4.5 of the bound for the FHP-like computations. This work extends previous work by making detailed definitions and refining the means of estimation. The result is increased insight into the nature of the constraints limiting throughput. Because Chapter 2 is rather lengthy, we give a brief summary below.

Outline of Chapter 2

Formal Basis for Input/Output Costs

A pebbling game is an analytical tool for modelling resource usage by a computation. The game is played on the associated data dependency graph, the pebbles representing machine storage registers. A pebble placed on a node is interpreted as indicating the value associated with the node is stored in the register associated with the pebble. This section introduces a new variant of the red-blue pebbling game, and shows that it models any computation of a parallel machine with fixed resources of input/output (I/O) bandwidth and storage. This section defines precise characterizations of the state of the pebbling game as it progresses, and proves, for this new pebbling game, two strengthened theorems similar to ones previously shown for the original red-blue pebbling game: the Partitioning Theorem, and the Dominator Theorem. These two theorems provide the basis for making I/O cost estimates using the new pebbling game by providing a characterization of sets of nodes in the data dependency graph of equal I/O cost.

Bounding the Size of the Estimators

This section proves two essential theorems: the Dependency Symmetrization Theorem, and the Dependency Collapsing Theorem. When making size estimates of the maximum size of the node sets mentioned in the first section, these two theorems taken together allow the assumption that the node sets have a canonical shape. Using previous work on discrete isoperimetric inequalities for the discrete torus, the exact form of these maximum sized sets is found for the HPP-like computations. Using previous work for continuous isoperimetric extremal sets, the use of the Wulff Crystal is introduced to solve the discrete isoperimetric problem for the FHP-like computations.

Nearly Optimal Architecture

Using the bounds on set sizes mentioned in the previous section, throughput bounds are derived in this section, and compared with the throughput of the WSA architecture for HPP- and FHP-like computations, showing that this architecture is nearly optimal.

Chapter 3 is an analytic and numerical study of the least-cost configuration for a WSA-type machine when problem scaling is required. The use of overlap-save method is described, and the efficiency of the WSA architecture using this method is derived. A linear cost function is derived for a completely scalable WSA-type machine. A numerical search is used to find the least cost machine configuration as the problem size is scaled under the constant-time assumption. A slightly super-linear speedup is shown over a moderate range of problem sizes. The least-cost machine parameters are described.

Chapter 4 introduces a data-embedded, specification-based testing technique for FHP lattice gasses. The tests consist of limit cycles in the cellular automaton. Their error detection coverage is shown empirically to be good. Their use in software, hardware, and system debugging is described, as well as their use as runtime simulation error detectors. A trade-off between the number of ''frames'' in the design of the cellular-automaton simulating machine and its amenability to this type of testing is discussed.

Chapter 2
Upper Bounding Throughput and A Nearly Optimal Architecture

2.1. Introduction

Ever since von Neumann and Ulam invented cellular automata [4], computing under this paradigm has been looked on with optimism because it appears to facilitate a high degree of parallelism. If cellular automata simulations can be effectively parallelized, methods which can be formulated as cellular automata computations have an advantage over those that cannot be similarly parallelized. Particular examples of such computational methods are the cellular automata known as lattice gasses, which are important candidates as substitutes for other, less easily parallelized computational methods used in such problem domains as fluid dynamics. Unfortunately, cellular automata simulations have as yet been unable to fulfill completely the hopes held for them, even though there have been some machines built specifically for their simulation, for two principal reasons. The first, which we will not address here, is that the theoretical understanding of complex systems has not matured to the point where such systems can be modeled by cellular automata with quantitatively predictable results. The second reason for the slow progress of cellular computing is that sufficiently powerful hardware has not been available.

The need for hardware with very high throughput for cellular automata simulation is mostly a consequence of the sizes of the simulated automata. For instance, it has become clear that applying lattice-gas simulation to interesting problems requires the use of automata with a very large number of cells [60], and therefore its effectiveness is dependent upon employing inexpensive, highly-parallel, special-purpose hardware. Because of the simplicity of the cellular computing paradigm, one would expect that simulators for cellular automata, and especially for lattice-gasses, should be able to use the inexpensive logical functionality provided by VLSI technology to achieve cost-effective performance. However, hardware with sufficient power to simulate large cellular automata is, as yet, still not cost-effective. The principal shortcoming, as we will show below, is that it is not possible to devote the majority of the processor's VLSI real-estate to usable logical functionality. The aim of this chapter is to identify and quantify the factors limiting the effectiveness of hardware resources in lattice-gas simulators.

The goal in designing special purpose hardware for simulating lattice-gas automata —assuming there is no computational shortcut to determining the automaton's future state— is to maximize the throughput for the automaton simulation given the architectural resource constraints. The most appealing idea for a lattice-gas simulating machine, or other cellular automata simulator, is a machine with a single processor for every cell of the automaton. In such a machine communication is strictly local: processors communicate only with processors assigned to neighboring cells of the automaton. This aspect of cellular automata, local communication, has been singled out as one of the fundamental advantages that cellular automata offer [5]. However, because the lattices often must be large for these simulations to achieve interesting behavior, and because the present technology limits our ability to interconnect and synchronize such large physical arrays, the usual approach to special purpose hardware has been to scan the lattice data through some sort of processor that updates the sites' states. That is, the current state of the automaton is held in some memory device and portions of the automaton state are passed to some hardware that updates the cells until a new state has been computed for every cell of the automaton.

Up to now, almost all special-purpose cellular automata machines built have used a scanning approach, and are similar in their basic logical organization of communication and processing. For a fairly comprehensive list of examples from CELLSCAN (1960) to Cytocomputer (1989), see [6]. There has also been a multitude of activity in this area in the systolic array literature, see [7] for examples, and see [8] for a description of the RAP1 machine which was built specifically with simulating lattice-gas automata in mind. Some machines have been designed to closely approximate the one-processor-per-cell concept, such as the Connection Machine [9], and the general cellular automaton simulating machine, CAM [5]. However, because it is overly restrictive in practice to add more hardware whenever one has an automaton simulation problem larger than the current machine can accommodate, these types of machines must resort to a scanning strategy when the problem size becomes large enough. Therefore, we will focus our attention on the class of special-purpose cellular automata simulators of the scanning type. The particular architecture we will concentrate on, the LGM-1 architecture [10], is one devised specifically for simulating lattice-gas automata, and is a scanning type of architecture employing custom VLSI chips for the update processors.

Let us make the discussion concrete by looking more closely at the LGM-1 machine. Figure 2.1 shows the top level organization of the machine (we will describe it in more detail later). The lattice-site state values are stored in the host machine, in

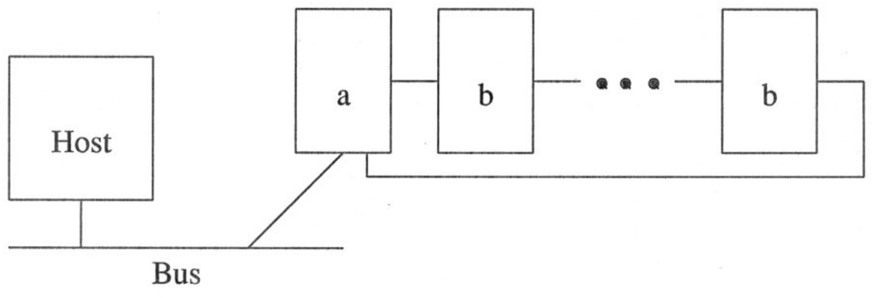


figure 2.1.

The LGM-1 machine organization. (a) is a bus interface board, and (b) represents processors boards containing ten custom VLSI chips each.

this case a Sun 3/160C†, and passed to the custom hardware over the VME bus. The updated site values are passed back over the bus to the host. The custom chips are arranged in a linear pipeline so that the output of one chip feeds the input of the next chip; the two end chips communicate with the host via bus interface logic. Repeatedly scanning the lattice array produces further global updates of the automaton state. Every custom chip in the processor updates the automaton state by one generation. The LGM-1 machine is extensible so that adding more chips increases the throughput: an n-chip machine has nearly n-times the throughput of a one-chip configuration.

Table 2.1 shows the measured performance of the LGM-1 machine and its host. The performance was measured for three different software C language drivers running on the host: *run_ca*, *cpu.SFM*, and *cpu.LOCAL*. Each of these drivers was also measured in an optimized form: *INLINE* indicates that all procedure calls were eliminated by putting the procedure code inline, *O2* indicates the code was compiled using an optimizing compiler, and *UNR* indicates that the code is 256-fold loop unrolled. The *run_ca.MON* code was compiled with a profiler. The *run_ca* code is the driver for the LGM-1 pipeline hardware. This code does all the host data handling duties for the custom boards and controls board operation via the VME bus: the code runs on the host and writes data and control words to the VME bus and on to the custom hardware; the results are collected by reading the output register of the custom hardware via the same route. Each cycle of the pipeline requires two 16-bit writes and one 16-bit read. This code is responsible for configuring the lattice data into slices 256 sites wide that are

† Sun is a trademark of Sun Micro Systems, Incorporated.

code	site updates	effective interface bandwidth	
		data rate	total bus traffic
	(M sites/sec)	(M sites/sec)	(M bits/sec)
run_ca.MON	0.0941	0.188	2.26
run_ca.INLINE.O2	0.201	0.402	4.82
cpu.SFM	0.214	0.429	5.15
cpu.SFMUNR.O2	0.578	1.16	13.9
cpu.LOCAL	0.0	0.684	8.21
cpu.LOCUNR.O2	0.0	1.43	17.1
run_ca.MON profile			
function	%	msec/call	total time
ca_pipe	48%	130	260
build_strip	26%	68	136
unpack_strip	25%	67	134

table 2.1.

LGM-1 measured computation and bandwidth performance (M $\equiv$ 10^6). *LGM-1 is configured as a one-stage pipeline. The bus traffic figures for the* LOCAL *and* LOCUNR.O2 *codes are projections for comparison purposes only.*

passed consecutively to the custom hardware, and rebuilding the lattice as the results return from the hardware. As the profile above shows, these slicing (*build_strip*) and rebuilding (*unpack_strip*) functions account for about one-half the total work done by the host. For comparison's sake, results are shown for the two *cpu* codes, which consist of only the five core instructions from the *run_ca* main loop. These codes do no computation and no data rearranging: for the *cpu.LOCAL* code the reads and writes are always to the same locations locally on the host; for the *cpu.SFM* code communication is between local variables and the custom hardware registers.

From table 2.1 we see that the host can provide at most 0.402 M sites/sec bandwidth to the custom boards when the host is also required to manipulate the storage location of the data (*run_ca.INLINE.O2*). Even when the host's only job is to step through memory and pass data back and forth to the custom boards, the peak rate is still only 1.16 M sites/sec (*cpu.SFMUNR.O*2), which is nearly the peak rate at which it can read and write it's own memory (*cpu.LOCUNR.O*2). We will assume the maximum data channel capacity between the host and the custom boards is 1.16 M sites/sec.

Consider the following facts about LGM-1.

(1) The custom chips can run reliably at at least 7 Mhz. Each chip contains storage for 518 lattice sites, and two programmed logic arrays (PLA's) that perform the update operations. At every tick of the global clock, each custom chip (i) reads two lattice site values from its input pins, (ii) produces two updated lattice site values, and (iii) writes two lattice site values to its output pins.

(2) The host can deliver at most 1.16 M sites/sec in bandwidth to the custom hardware: one two-site read, one two-site write, and one global clock pulse at a time.

(3) Using the minimum local storage per PLA (seven lattice sites), the area of the chip could hold 10 PLA's with their local data.

Looking at (3) it seems plausible that one could build, using the same technology, custom chips with ten PLA's per chip running at 7 Mhz and get $2 \cdot 10 \cdot (\text{ 7 Mhz }) = 140$ M site-updates/sec per chip. In contrast, the actual measured computation rate is about one three-hundredth of this at 0.508 M site-updates/sec. Even without adding processors to the custom chips, as LGM-1 is presently constructed, clocking the chips at 7 Mhz leaves them idle for more than 10 local clock ticks for every global clock signal received via the bus. Is it possible that by using the local storage more effectively and reordering the computation and data access pattern we could use these idle periods to compute ten times more updates before sending the site values back to the host? If we packed ten PLA's onto each chip, and reorganized in just the right way the data sent to the chips, could we get 140 M site-updates/sec per chip performance? The general question posed by these questions is, how effectively does LGM-1 use its resources?

In this paper we develop a measure of optimality for scanning type architectures with respect to two resource constraints, main storage bandwidth and local storage capacity, that tells quantitatively how well an architecture makes use of its resources for the lattice-gas simulation problem. That is, we will answer whether or not the throughput of LGM-1 could have been considerably improved by some clever computational strategy. In particular, we show that the linear pipeline, as defined by the

LGM-1 architecture, approaches a bounding throughput value to within a small constant. This suggests that this architecture nearly achieves, or perhaps does achieve, the maximum throughput possible given the constraints on memory bandwidth and local storage regardless of the speed, computational power, number, or intercommunication network of the processors in any parallel machine computing the lattice computation.

Background

Because the structures of the data dependency graphs for cellular automata computations are independent of the input data, the data dependency graph is a particularly useful characterization of a computation for developing insight into the inherent limitations of the computation. This is done by modeling and analyzing the effects of machine resources on the computation of the nodes of the data dependency graph without regard to the values the nodes represent. The resources we have identified above can be divided into three categories: communication, storage, and computation. Here we review some of the methods used in analyzing data dependency graphs, along with some related work specifically dealing with I/O communication requirements.

Perhaps the first factor to consider in analyzing the effect of resources on the computation of a cellular automaton simulation, is the number of processors. Assuming processors are the only resource, Theorem 1 of the software analysis of Eager, Zahorjan, and Lazowska [11], leads to two conclusions. One is that, in simulating a cellular automaton with K cells, on average no more than K processors can be kept busy simultaneously. The other conclusion is that, if storage and communication are unlimited, then n processors can be kept simultaneously busy without delays, provided n is less than K. That is, if a simulation requires time t using one processor, then a machine with n processors can complete the same simulation in time t/n (with the above restriction on n). Referring to our earlier look at LGM-1, this suggests that either LGM-1 can be sped up linearly by using more processors more intelligently, or resources other than processing power are the limiting factors.

The effects of communication resources have been studied in several different ways. The effect of hierarchical communication has been studied by using variations of the general idea of extending the PRAM model by charging $\log(i)$ for accessing the data item at the i^{th} address. See Aggarwal, Alpern, Chandra, and Snir [12] and Aggarwal, Chandra and Snir [13]. Instead of separating communication into hierarchies by the above approach, communication can be divided into just two components: communication local to the machine, and I/O. This leads to the consideration of the effect of I/O complexity. For example, Aggarwal and Vitter [14] use a counting argument on the number of permutations possible using M units of internal storage and assuming I/O communication is in blocks of size B, to determine a bound on the minimum I/O required for sorting and related problems. In an approach that is similar to pebbling, Papadimitriou and Ullman [15] develop tradeoffs between the total I/O and the time in a parallel processing environment. Other work on communication complexity is exemplified by the recent paper by Lovász and Saks [16] which addresses the communication required between processors jointly sharing the computation of some function.

The effects of limited storage locally available to the processing machine has been studied by using various pebbling games. For instance, the articles by Lengauer and Tarjan [17, 18] give storage-space/time tradeoffs for several families of data dependency graphs. See the article by Pippenger [19] for a general summary of traditional pebbling analysis.

The combined importance of communication and local storage has been recognized for a long time, and, as was pointed out by H.T. Kung in [20], is especially important for high-speed special-purpose hardware. In [21] he and Hong devised a pebbling game that models the effects of these resources on computational power. The red-blue pebble game described there models the computation and I/O steps in a sequential computation. They used it to get space-I/O tradeoffs for several problems, and to get upper bounds on speed-up of a computation of these problems using a sequential machine. The red-blue pebble game they describe was extended by Savage and Vitter [22] to the parallel-red and the block-red-blue pebble games, which model parallel computation without I/O, and with block-parallel I/O respectively.

In [23] Kugelmass, Squier, and Steiglitz introduced a further refinement of the red-blue pebble game that allows arbitrary computation and I/O parallelism. Based on this work Lopresti and Nodine [24] developed a lower bound on the the I/O requirements for simulating the ''Life'' cellular automaton [25], and applied this bound to several computational schemes for employing an n by n processor array.

Chapter Outline

The goal of this chapter is to find an upper bound on throughput for computing the evolution of a lattice-gas automaton, which we can use to compare with the performance of LGM-1 (or any other machine) to determine whether the machine's resources have been used to maximum potential. As we mentioned above, limiting factors are local storage and I/O costs, and the pebbling games address both these constraints. We will analyze the data dependency graphs of lattice-gas simulations using pebbling arguments to develop the required bounds on throughput. There are several issues that we address in this chapter that are essential to making performance comparisons. The first issue concerns the bounds established in previous work. In [21] and [23] the bounds are asymptotic in the amount of local processor storage. However, there are two problems with using these estimates: (1) although the bounds are valid within a finite range of values for the number of red pebbles used, the bounding arguments are not valid in the asymptotic limit, and (2) the valid portion of the resulting bounding curve is very loose. Thus, the primary task taken up here is to establish a sufficiently tight non-asymptotic bound using a variant of the red-blue pebble game.

The second issue we address is that making tighter bounds requires establishing a formal basis for these estimates. Previous work [24] attempting to establish tighter non-asymptotic bounds relied on two premises that were not established: (a) that the arrangement of r red pebbles in a data dependency graph that results in the largest amount of pebbling without I/O consists of placing all the pebbles in a single layer of the graph, and (b) that optimal arrangement within the layer consists of packing the vertices into a ball, the set of vertices with distance less than some parameter d from a specified vertex. The tasks taken up here then are the following:

(1) Establish the formal basis for making size estimates of sets of data dependency graph nodes defined by pebbling. See section 2.2.

(2) Solve the problem of extremal pebble sets, and derive the bounds on I/O costs. See section 2.3

(3) Apply the I/O bounds to throughput comparisons. See section 2.4

Implicitly, there is another issue that is addressed here, at least partially. The shapes of sets of data dependency graph nodes defined by pebbling game optimization, define the computational strategy of least I/O cost. So, we hope to find not only the numerical value of the least I/O cost for a machine with given fixed machine resources, but also discover a method of determining the optimal computational strategy.

2.2. The Formal Basis for Input/Output Bounds

2.2.1. The Parallel Red-Blue Pebble Game

In this section we define the $parallel-red-blue$ pebble game. It models any computation that can be performed by a machine which has arbitrary parallel I/O capabilities to an external memory, and internally is equivalent to a CRCW PRAM†. Before we discuss the necessity of defining a new pebbling game, we first need to review the definition of the original red-blue pebble game [21].

The red-blue pebble game is played on a directed acyclic graph with uniformly bounded indegree according the following rules (see figure 2.2).

i) a pebble of any color may be removed from a vertex at any time.

ii) a red pebble may be placed on any vertex that has a blue pebble.

iii) a blue pebble may be placed on any vertex that has a red pebble.

iv) if all immediate predecessors of a vertex v are red pebbled, v may be red pebbled.

The ''inputs'' are those vertices which have no predecessors, and the ''outputs'' are those which have no successors. A vertex that is blue-pebbled represents the associated value's presence in main memory. A red-pebbled vertex represents presence in processor (chip) memory. Rules (ii) and (iii) represent I/O, and Rule (iv) represents the computation of a new value. Generally, the game is played with a fixed number of red pebbles and an infinite supply of blue pebbles. The goal of the game is to blue-pebble the outputs given a starting configuration in which the inputs are blue-pebbled and the rest of the vertices are free of pebbles.

The need to add parallelism to the red-blue game is a consequence of the fact that it is an open question whether or not the red-blue game is sufficient for modelling I/O in general parallel computations. One approach to investing it with parallel properties without defining a new game consists of considering a block of moves as occurring in a single ''time step''. This allows a certain form of parallelism, and is the extension used by Savage and Vitter [22] in the block-red-blue game: the actual play of the game is not altered, but rather the counting of moves is redefined. However, in the general case, it is easy to find a simple example in which the number of I/O steps can be

† Such a machine model consists of an arbitrary number of processors communicating via a shared memory. This model is often referred to as a CRCW PRAM: concurrent-read concurrent-write parallel random access machine [26].

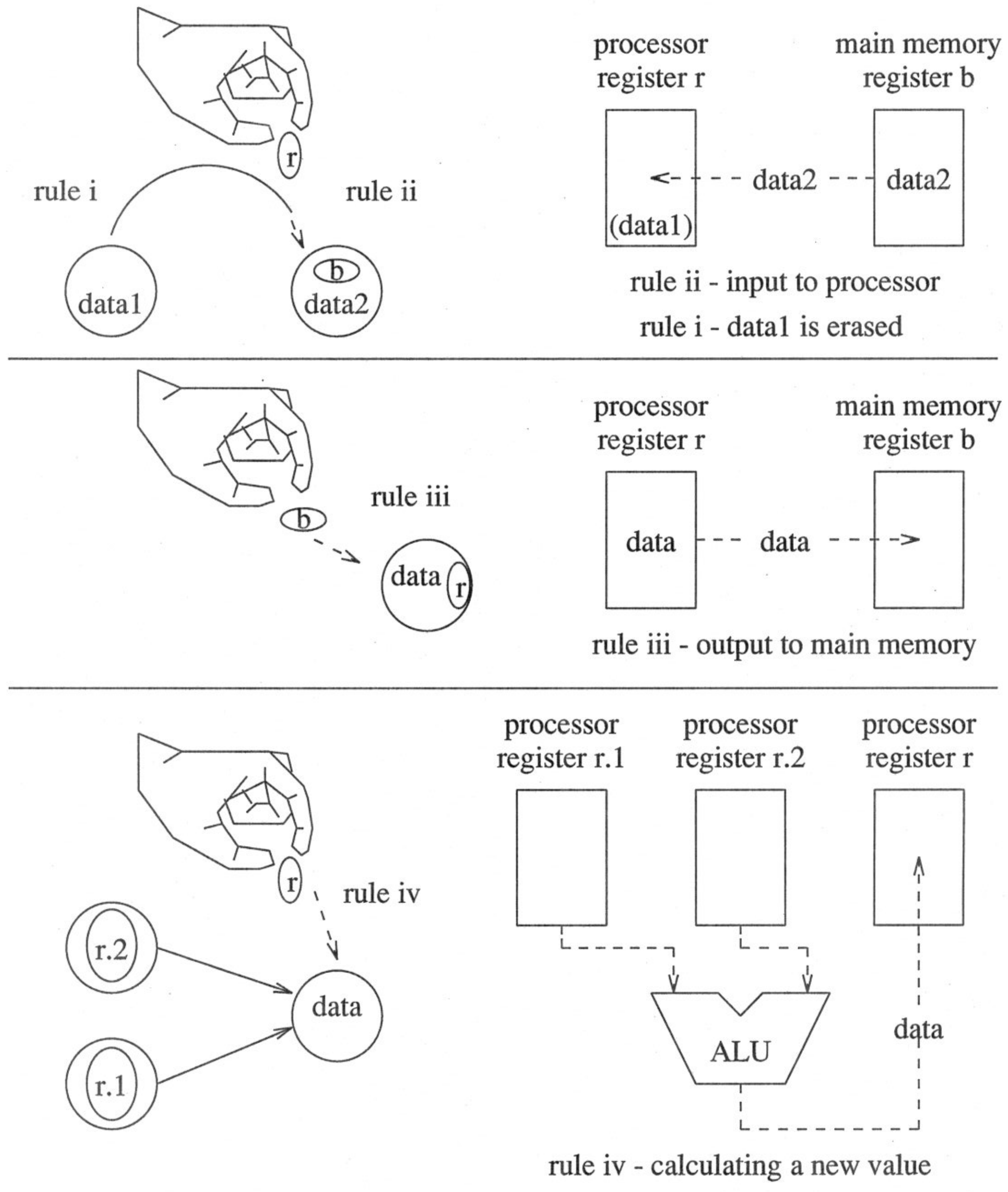


figure 2.2.

Interpretation of the rules of the standard red-blue pebble game. Rules i and ii are illustrated in the top diagram, Rule iii in the middle diagram, and Rule iv in the bottom diagram. Respectively, the interpretations are, register deallocation, write to processor register, write to main memory register, and write ALU result to local register. Ellipses labelled "r.1," "r.2," and "r" represent red pebbles, and those marked "b" represent blue pebbles.

reduced by allowing the Rule iv red pebbling moves to occur in parallel. The problem with the standard red-blue pebble game is that lifting a pebble to move it to a node that is being calculated may destroy the precondition for Rule iv red pebbling another node.

The parallelism we want to model in the red-blue game is to allow any number of pebble moves to occur simultaneously, so long as their preconditions allow a move in the non-parallel game, yet we also want to maintain a pebbling game that moves a single pebble at a time. The reason we want to maintain a linear ordering of pebble placement is that the method of bounding I/O cost depends on dividing any pebbling of a data dependency graph into pieces of equal I/O cost in such a way that the division induces a partition of the nodes of the graph into pieces whose sizes can be easily estimated. The size estimates on these pieces are derived from a bound on the maximum amount of pebbling possible given an arbitrary placement of red pebbles, but without the use of blue pebbles. Consequently, we must be sure that the partitioning does not violate the data dependencies; otherwise, an element of the partition might contain more nodes than the method of estimation allows, invalidating the estimate. If the pebbling moves are linearly ordered without data dependency violations, then the pebbling can be divided into contiguous pieces that result in a valid partitioning of the nodes. The red-blue pebble game can be so divided because it is a strictly sequential game: a single pebble is moved on the data dependency graph according to the rules stated above, and and the resulting configuration determines the applicable rules for the next move. Since we want to use the same techniques for bounding I/O cost in the parallel case as was used for the red-blue game, we want the parallel game to be a sequential game as well.

Definition: the rules of the $parallel-red-blue$ pebble game:
The game is similar to the red-blue pebble game with the addition of a new pebble (pink, for ''place holder''), and a modified Rule iv:

i) a pebble of any color may be removed from a vertex at any time.

ii) a red pebble may be placed on any vertex that has a blue pebble.

iii) a blue pebble may be placed on any vertex that has a red pebble.

iv) if all predecessors of a vertex v are pink pebbled, then v may be red pebbled.

v) the game consists of cyclic repetition of three consecutive phases: write-phase, calculate-phase, read-phase.

vi) the write-phase consists of only Rule i and iii moves (output).

vii) the calculate-phase comprises the following moves

a) pink pebbles are placed on every node containing red pebbles

b) red pebbles are moved according to Rules i and iv.

c) at the end of the phase all pink pebbles are removed.

viii) the read-phase consists of only Rule i and ii moves (input).

□

A complete cycle represents a single parallel step in the computation. We use the following terminology: placing a red pebble on a node that contains no red pebbles is a *calculation*. The node pebbled is called the *dependent* node, and the nodes with arcs ending at the dependent node are called the *supporting* nodes. Usually, the game is played with a restricted number of red pebbles.

It is easy to see the parallel-red-blue pebble game allows the parallelism we want. Consider a computation which proceeds by doing many steps in parallel in real time. Decompose the computation into pieces which occur simultaneously designated C_i; we say the complete computation C consists of their concatenation: $C = C_1 \cdot C_2 \cdot \cdots C_k$. Consider the pebble moves within a single parallel move C_i: memory reads, memory writes, and register to register calculations. Since all these actions take place simultaneously, we can order them arbitrarily because they are not interdependent. Thus, we may order them exactly as the phases are ordered in the parallel-red-blue game. Within the phases there are no interdependencies, and the individual moves may be arbitrarily ordered. Consequently, there is a pebbling that models the steps of the computation.

We next show that every pebbling represents a valid CRCW computation. Assume that every pebble move is valid, so far. The first phase of a cycle is the write-phase: some nodes with red pebbles on them get blue pebbles. The red pebbles were placed in an earlier cycle, so they are not a concern. A blue pebble that is moved onto a node containing a red pebble might have been sitting on another node previously; however, this simply represents a memory register overwrite of previously stored data, in the case that the blue pebble was not moved previously during this cycle, or it represents the concurrent writing of more than one register to the same memory location, in the case the blue pebble was moved in this cycle. Neither case conflicts with the model of computation. See figure 2.3.

The second phase is the calculate-phase: some nodes with all their supporting nodes pink-pebbled get one or more red pebbles. The sources for any calculation in

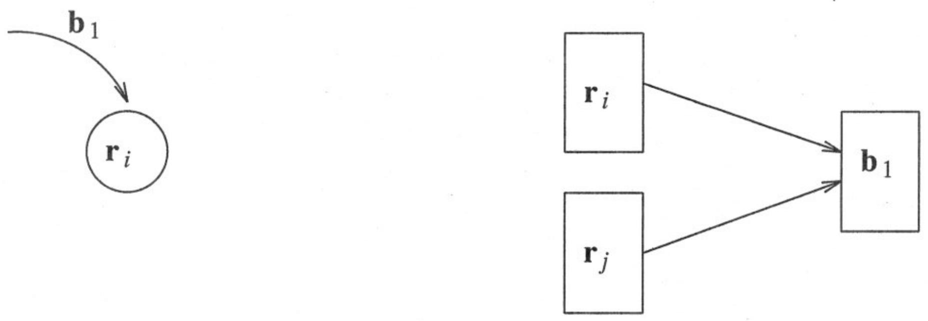


figure 2.3.

Left: A blue pebble, $\mathbf{b}_1$*, is placed on a node of the data dependency graph that already has a red pebble* $\mathbf{r}_i$ *on it. Right: the hardware interpretation is that the data held in register* $\mathbf{r}_i$ *is written to the main memory register* $\mathbf{b}_1$*. If pebble* $\mathbf{b}_1$ *is moved to another such node, say one containing pebble* $\mathbf{r}_j$*, the interpretation is that the two registers make concurrent writes to the same memory location; in this case the result is that the contents of* $\mathbf{r}_j$ *end up in memory because the* $\mathbf{b}_1$ *ends up on the same node as* $\mathbf{r}_j$*.*

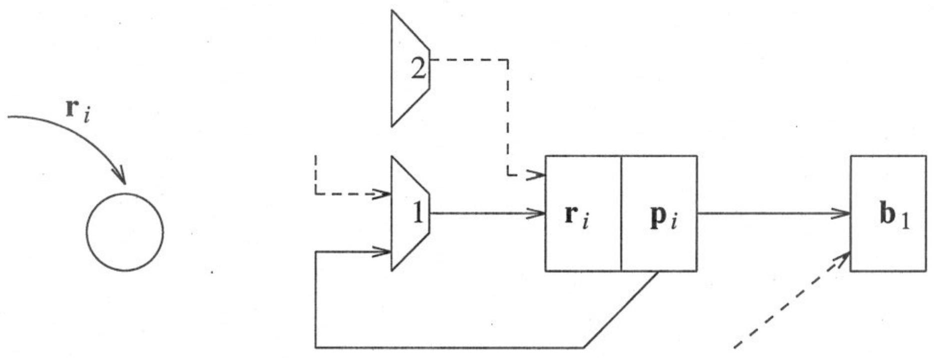


figure 2.4.

Left: a red pebble $\mathbf{r}_i$ *is placed on a node by using the calculation rule (the node may or may not have pebbles on it already). Right: the possible interactions are summarized. The output side of a master-slave register* $\mathbf{p}_i$ *feeds into an ALU while the input side* $\mathbf{r}_i$ *receives the result of the calculation. The red pebble* $\mathbf{r}_i$ *may have been used in another calculation during this calculation-phase (indicated by dashed lines from ALU 2) which is interpreted as a concurrent write to register* $\mathbf{r}_i$*. The previous write-phase of the cycle also allows* $\mathbf{p}_i$ *to concurrently fan-out its data to a memory location (that is, the pebble* $\mathbf{r}_i$ *may have been moved from a node that was blue-pebbled in the write-phase).*

this cycle, the supporting nodes, were red pebbled in a previous cycle (as opposed to having red pebbles during this cycle), so the input to the calculation is valid. There are two possible sources of conflict. The red pebble placed on the dependent node could have been on a node which was newly pebbled either blue or pink during this cycle. In either case this represents simultaneously reading and writing the register associated with the red pebble. This is within the CRCW model, and is commonly implemented in practice. If the red pebble was used previously in this cycle to receive the results of a calculation, this is again a concurrent write to the same register; the ''last write wins.'' See figure 2.4.

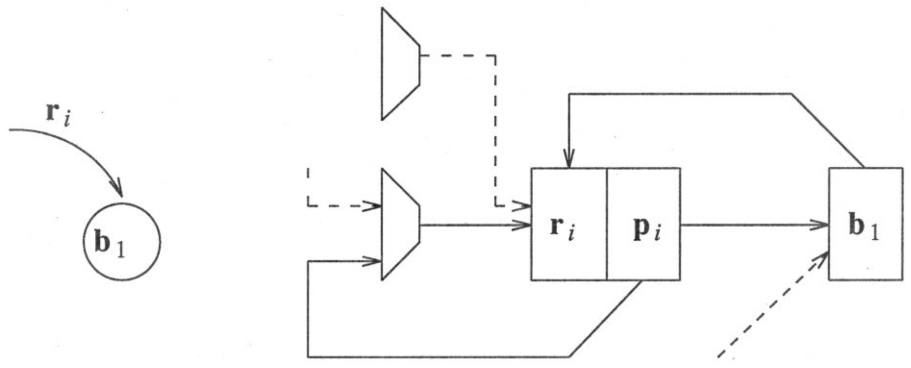


figure 2.5.

Left: a red pebble $\mathbf{r}_i$ is placed on a node holding a blue pebble $\mathbf{b}_1$. Right: summarizing the result shows another concurrent write to register r_i (possibly from multiple memory locations).

The last phase is the read-phase: some blue pebbled nodes get red pebbles. There are again two possible conflicts: the red or blue pebbles may have been moved previously during this cycle. The blue pebble may have been used to receive data in the write-phase. Again this is concurrent reading and writing of a register. The red pebble may have been used as a data source for writing to memory, as a data source for a calculation, or as a destination for a calculation during this cycle. Again this represents concurrent read/write and again ''last write wins.'' See figure 2.5.

2.2.2. Dependency Graph Partitioning and Partition Element Estimators

This section sets forth the foundations of making input/output cost estimates (the ''Partitioning Theorem'', Theorem 1, and the ''Dominator Theorem'', Theorem 2). We use the general strategy introduced by Hong and Kung in [21] to get an estimate of the total input/output cost of an optimal pebbling using r red pebbles. Their strategy is as follows.

(1) Establish the existence of a partitioning of the nodes of a data-dependency graph G for any pebbling P of that graph using r red pebbles. The partition has the property that each element of the partition accounts for k I/O moves of P, where k is a

fixed constant parameter.

(2) Find a bound β on the maximum number of elements in a member set of the partition.

   (a) Identify a convenient containing set for each element of the partition. The containing set is characterized by a ''dominator'' set of known size less than or equal to $(r + k)$.

   (b) Establish an upper bound $\beta(r, k)$ on the size of sets dominated by sets of size $(r + k)$.

(3) Estimate the number of sets in the partition as $h \geq \frac{|G|}{\beta}$. This gives a lower bound on the I/O cost as $h \cdot k$.

In the next section we address steps (1) and (2a). The method we use for (1) replaces the arguments based on paths in the data dependency graph used in [21] with careful definitions of vertex sets in pebbling games. The results are then simple consequences of the definitions, and allow precise identification of the sets for which we later develop size estimates. The method we use for (2a) significantly improves the estimate of maximum partition size so that the estimate is tighter.

We will need to develop some terminology which will allow us to precisely define the sets for which we want to develop size estimates. The appearance of the result may seem unnecessarily complicated; however, the need for such precision is demonstrated when subtle errors appear in arguments that do not use carefully defined terminology. For instance, in [14] the authors attempt to prove a result found in [21] concerning a lower bound on the total I/O of an FFT computation. The proof fails because of the mistaken identification of the sets of nodes red pebbled during I/O operations under different I/O conventions.

The Partitioning Theorem

This section develops notation for defining sets of vertices that arise implicitly in the play of the red-blue and parallel-red-blue pebble games. We begin with the definition of an I/O-division of a pebbling game, and conclude with a description of the induced partition of the nodes of the data dependency graph. In the following let a pebbling $P = (p_1, p_2, \ldots, p_n)$, where each of the p_i is a single pebbling move, be a pebbling of a computation graph $G = (X, A)$, where X is the set of vertices of G and A is the set of directed edges in G. A *sub-pebbling* of P is any sub-sequence of consecutive pebbling moves of P. Let $C_{I/O}$ be the number of p_i in P that are I/O moves.

Let k be a positive integer not greater than $C_{I/O}$, and define h as

$$h = \left\lceil \frac{C_{I/O}}{k} \right\rceil, \text{ when } C_{I/O} \neq 0, \text{ and otherwise } h = 1.$$

Definition: A sequence of non-negative integers σ is a $k - I/O - division$ of a pebbling if it satisfies the following:

(i) $\sigma = (\sigma_0, \sigma_1, \ldots, \sigma_h)$ is a monotonically increasing sequence in the range $[0, n]$ with $\sigma_0 = 0$, and $\sigma_h = n$.

(ii) σ divides a pebbling P into h pieces $(P_1, P_2, \ldots, P_h)$ in the following sense:

Each of the pieces is a $sub-pebbling$ of P: $P_i \equiv (p_{\sigma_{i-1}+1}, p_{\sigma_{i-1}+2}, \ldots, p_{\sigma_i})$ for $1 \le i \le h$.

Exactly k of the p_j contained in P_i are I/O moves, except that P_h may contain any integer number of I/O moves in the range $[0, k]$. (For instance, if there are k or fewer I/O moves in P, then $\sigma = (0, n)$, and $P_h = P_1 = P$.)

□

Definition: The *Red Set at step i*, R_i, is the sub-set of X containing the vertices which have red pebbles:

$$R_i = \{x \mid x \text{ has a red pebble after } p_i\}.$$

□

Note that $R_i - R_{i-1}$ identifies a vertex —if one exists— that was red-pebbled by pebbling move p_i. A pebbling begins with $R_0 = \varnothing$.

Definition: The *Blue Set at step i*, B_i, is defined analogously to R_i.

Definition: The *Computed Set at step i*, Ψ_i, is the sub-set of X that has been red-pebbled at or before step i, or are inputs:

$$\Psi_i = \bigcup_{k=1}^{i} R_k + \Psi_0 \text{ where}$$

$$\Psi_0 \equiv \{x \mid x \text{ is an input vertex}\}.$$

□

We interpret this set as the set of vertices that have been calculated up to the end of pebbling step i. (See figure 2.6.)

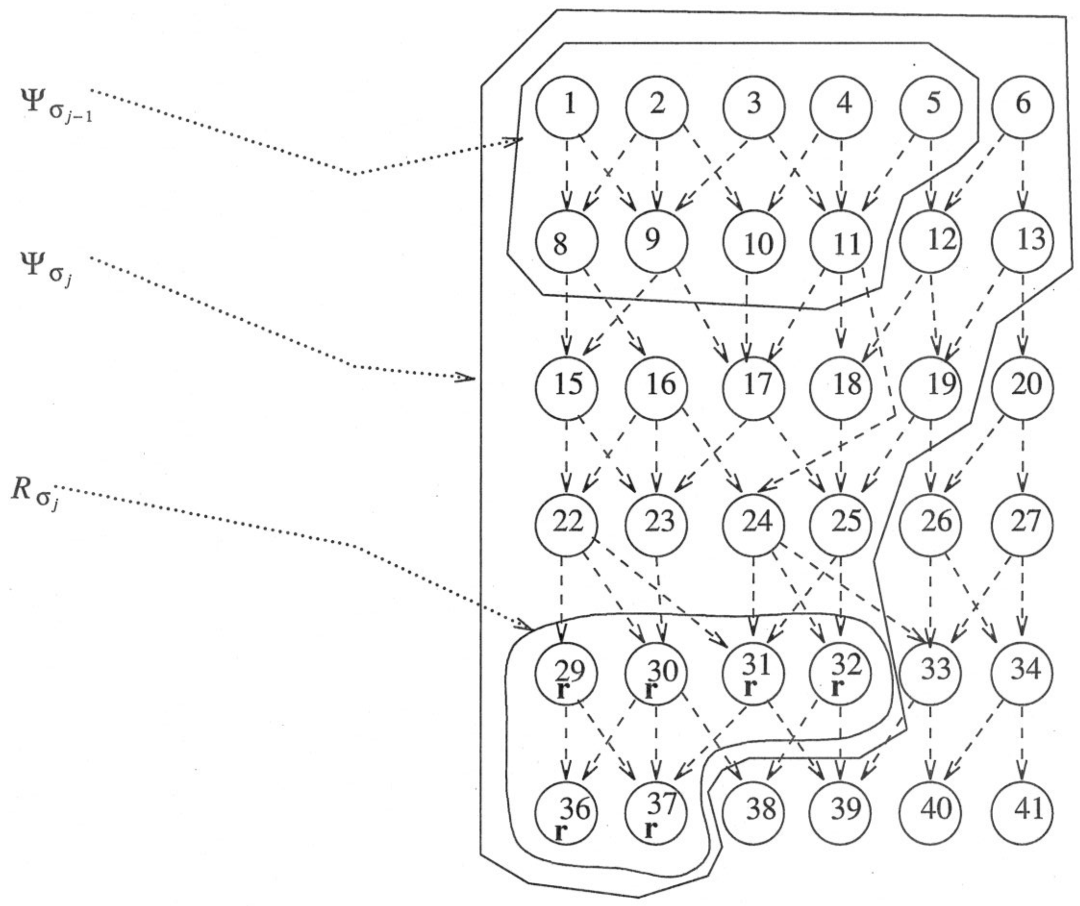


figure 2.6.

A pebbling game at the end of pebble step p_{σ_j}. *(Note that nodes 7, 14, 21, and so on are not shown.)* R_{σ_j} *is the red-set at the end of move* p_{σ_j}. *The nodes in the region labelled* Ψ_{σ_j} *were red pebbled by some pebbling move* p_i *where* $1 \le i \le \sigma_j$. *Also,* Ψ_{σ_j} *indicates that the nodes* $\{20, 26, 27, 33, 34, 38-41\}$ *were not red pebbled by any of these pebbling moves. The kernel is* $\hat{P}_j = \{12, 13, 15-19, 22-25, 29-32, 36, 37\}$.

Definition: The *Kernel* of a sub-pebbling P_i is the set $\hat{P}_i$ of vertices which get red-pebbled for the first time by some move in P_i:

$$\hat{P}_i = \Psi_{\sigma_i} - \Psi_{\sigma_{i-1}} .$$

□

Note that Ψ_0, the set of inputs, is not contained in any kernel.

Definition: The *Red Closure* of a sub-pebbling P_i is the set $\overline{P}_i^R$ of vertices that have been red-pebbled by some move in P_i:

$$\overline{P}_i^R = \bigcup_{j=(\sigma_{i-1}+1)}^{\sigma_i} (R_j - R_{j-1}) \ .$$

□

In general, $\hat{P}_i \neq \overline{P}_i^R$ since the red closure includes vertices which are red pebbled and already contain blue pebbles (input from main memory), and vertices which had been previously red-pebbled and were red-pebbled again during P_i (recalculation). The *Blue Closure* is defined analogously (see figure 2.7).

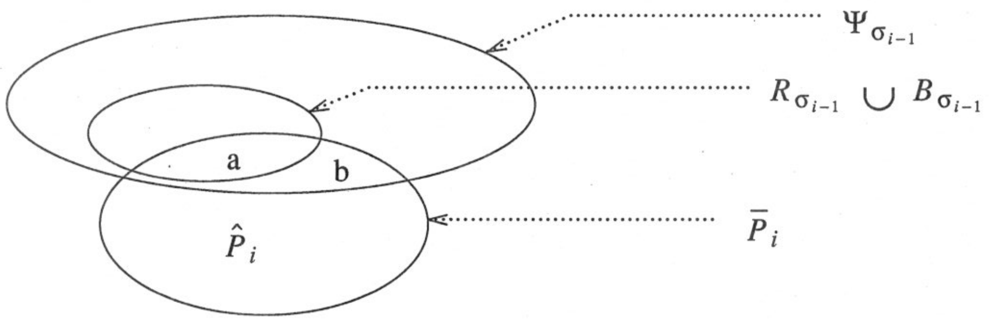


figure 2.7.

A Venn diagram of the sets defining the closure of a sub-pebbling. The region marked "a" indicates nodes that had red or blue pebbles on them at the start of sub-pebbling P_i, and were pebbled again in P_i (recalculation or I/O or both). The region marked "b" indicates nodes that had been calculated by previous sub-pebblings and were recalculated and possibly used for I/O in P_i.

Definition: The *Closure* of a sub-pebbling P_i is simply the set of vertices which were either red or blue pebbled during P_i:

$$\overline{P}_i = \overline{P}_i^B \cup \overline{P}_i^R \ .$$

□

We now state a partitioning theorem similar to the partitioning theorem in [21]. This theorem simply states that the kernels defined above partition the nodes of the data dependency graph.

Theorem 1: The collection of kernels of the sub-pebblings defined by a k-I/O-division, $V = \{\hat{P}_1, \hat{P}_2, \ldots, \hat{P}_h\}$, is a partition of the non-input vertices of the computation graph $G = (X, A)$.

proof:

i) We assume throughout this chapter that the data dependency graphs do not contain useless nodes: nodes without incoming or outgoing arcs. Consequently, every vertex $x \in X$ is red pebbled at least once, and therefore every vertex must belong to at least one red-set R_i:

$$\begin{aligned}
\bigcup_{i=1}^{h} \hat{P}_i &= \bigcup_{i=1}^{h} \left(\Psi_{\sigma_i} - \Psi_{\sigma_{i-1}} \right) \\
&= \bigcup_{i=1}^{h} \left(\bigcup_{j=1}^{\sigma_i} R_j - \bigcup_{j=1}^{\sigma_{i-1}} R_j - \Psi_0 \right) \\
&= \bigcup_{i=1}^{h} \left(\bigcup_{j=\sigma_{i-1}}^{\sigma_i} R_j \right) - \Psi_0 \\
&= \bigcup_{j=1}^{n} R_j - \Psi_0 \\
&= X - \Psi_0 .
\end{aligned}$$

ii) By definition the $\hat{P}_i$ are all disjoint.

□

The Dominator Theorem

We now turn our attention from the sets defined by a pebbling, and define sets which are implied by the dependencies in $G = (X, A)$.

Definition: The *Support Neighborhood* $N(M)$ of a set $M \subseteq X$ is the set of nodes which can be said to "support" the calculation of the nodes in M (see figure 2.8):

$$N(M) = \{ y \in X \mid (y, m) \text{ is an arc in } A \text{ and } m \in M \} .$$

$N^k(M)$ is the support neighborhood k levels back in the graph:

$$N^k(M) = N\left[N^{k-1}(M) \right] , \text{ where}$$

$$N^0(M) \equiv M .$$

The *Support* $\overline{N}(M)$ of a set is the union of all nodes necessary to calculate the set M:

$$\overline{N}(M) = \bigcup_{j=1}^{\infty} N^j(M) .$$

□

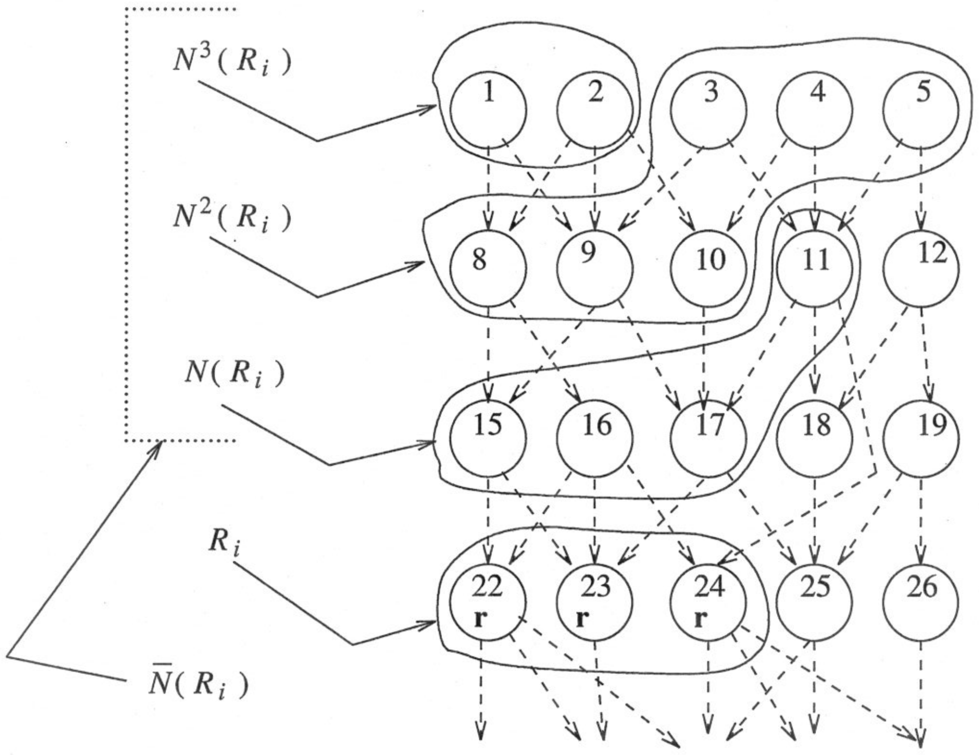


figure 2.8.

The support of the set R_i = {22, 23, 24}. The set {1-5, 8-11, 15-17} constitute the support of R_i: every node in this set must be pebbled before R_i can be completely pebbled.

Definition: The *Dependent Neighborhood* $D(M)$ of a set $M \subseteq X$ is the set of vertices which could be calculated by the next pebbling step if M was entirely red-pebbled but nodes outside of M were un-pebbled:

$$D(M) = \{x \mid N(x) \subseteq M\} - M .$$

The *Dependency* $\overline{D}(M)$ of set M is the set containing all the vertices for which knowledge of M is sufficient for their computation:

$$\overline{D}(M) = \bigcup_{j=1}^{\infty} D^j(M) \quad \text{where}$$

$D^k(M)$ is the k^{th} recursive iteration of $D(M)$, taking the union of previous neighborhoods to describe the next:

$$D^k(M) = D(M \cup \Theta_{k-1}) , \text{ where}$$

$$\Theta_n = \bigcup_{j=1}^{n} D^j(M) \quad , \text{ and}$$

$$\Theta_0 \equiv \varnothing \ .$$

□

Note that $M \cap D^k(M) = \varnothing$ for all k. See figure 2.9.

As an aside, note that, given a set of nodes in a data dependency graph, it is intuitive that nodes that are not in its dependency cannot become a part of its dependency by addition of nodes and arcs to the graph. Establishing this allows the bounds on the sizes of dependencies to be

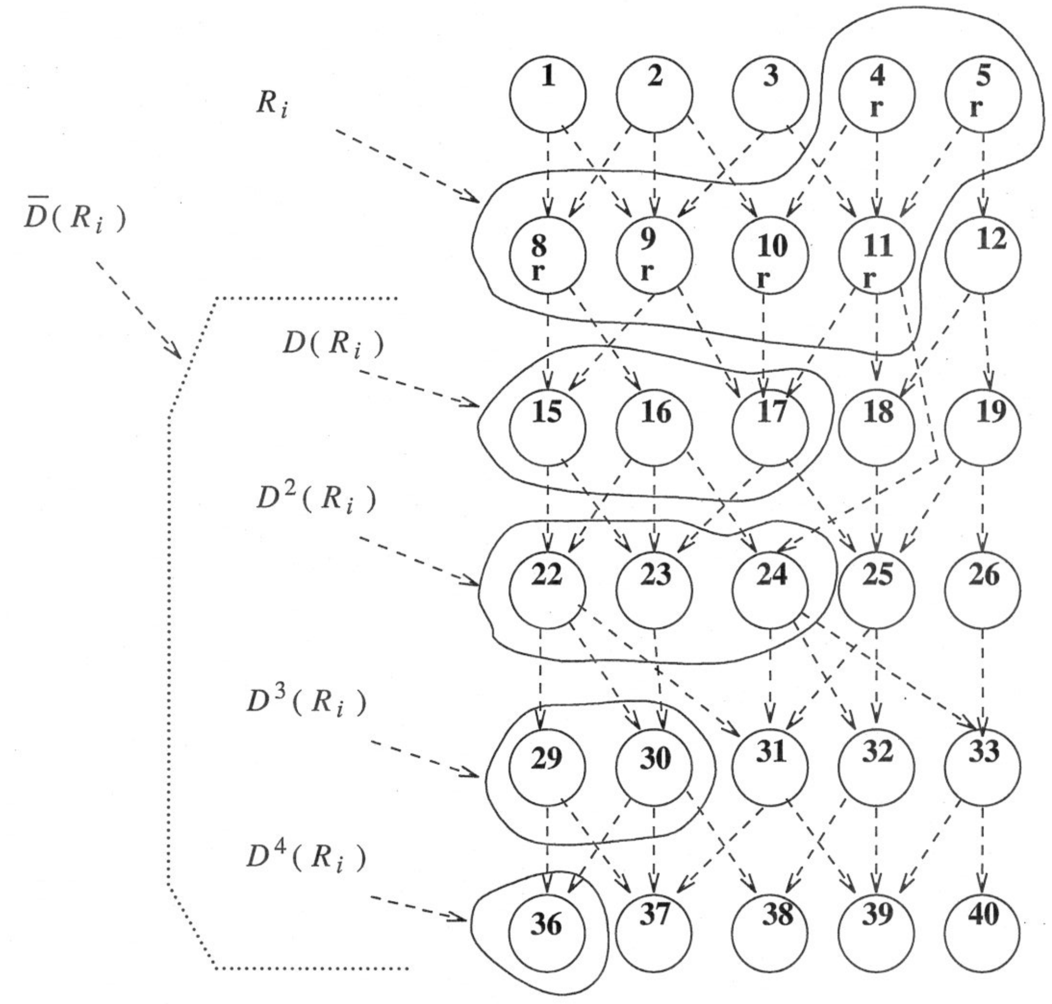


figure 2.9.

The dependency of the set R_i = {4, 5, 8-10}. The set of nodes {15-17, 22-24, 29, 30, 36} is the dependency of R_i: knowledge of the values in the nodes of R_i is sufficient to calculate all the nodes in its dependency.

applied to any super-graph. The statement of this proposition and its proof can be found in the appendix.

The estimates on the number of sets in a partition mentioned at the beginning of section 2.2.2 come from estimating the size of a convenient super-set: the dependency of some of the nodes containing pebbles before P_i begins. These pebbled nodes are called the *dominator* of P_i.

Definition: A set d_M is a *Dominator* of a set M, if

$$M \subseteq \overline{D}(\ d_M\ )\ \ .$$

□

Definition: The *Kernel Dominator* , d_i, of the sub-pebbling P_i is

$$d_i \ = \ R_{\sigma_{i-1}} \cup (\ B_{\sigma_{i-1}} \cap \overline{P}_i^{\ R}\ )\ .$$

□

We will show below that d_i is in fact a dominator of $\hat{P}_i$. First we will show that the closure of P_i is contained in the union of d_i and the dependency of d_i.

Lemma 1:

$\overline{P}_i \subseteq (\ \overline{D}(\ d_i\ ) \cup d_i\ ).$

proof:

We use a straightforward induction to establish the above. In the following let P_i be a sub-pebbling with

$$P_i \ = \ (p_{\sigma_{i-1}+1}\,,\, p_{\sigma_{i-1}+2}\,,\ \cdots\,,\, p_{\sigma_i})\ .$$

For convenience, let us write

$$P_i = (\ p_1\,,\, p_2\ ,\, \ldots\, ,\ p_t\ )\ ,$$

where it is understood p_c represents $p_{\sigma_{i-1}+c}$, and that $t \equiv \sigma_i - \sigma_{i-1}$. Let Q_j be a sub-pebbling formed from P_i by taking the first j elements,

$$Q_j = (\ p_1\ ,\, p_2\ ,\, \ldots\, ,\ p_j\ ) \ \text{ for } 1 \le j \le t\ \ .$$

The induction will be over j and we will show that the condition above is satisfied by $\overline{Q}_j$. Then, since $\overline{P}_i = \overline{Q}_t$ we will have the result. Since the Rule i pebbling moves and any move of a pink pebble do not affect the definitions of the various node subsets, we will assume the moves in P_i consist only of moves placing a red or blue pebble on a node.

(basis):

$Q_1 = (p_1)$. Let p_1 pebble node x, then $\overline{Q}_1 = \{ x \}$. There are three possible cases for p_1:

[1] p_1 is a calculation. Then

$N(x) \in R_{\sigma_{i-1}}$, and by definition

$x \in D^1(R_{\sigma_{i-1}}) \subseteq \overline{D}(d_i)$.

[2] p_1 is an input operation (adding red to blue on x). Then

$x \in (B_{\sigma_{i-1}} \cap \overline{Q}_1^R) \subseteq d_i$, by

definition of $\overline{P}_i^R$ and the definition of d_i.

[3] p_1 is an output operation (adding blue to red on x). Then

$x \in R_{\sigma_{i-1}} \subseteq d_i$.

(induction):

Assume $\overline{Q}_k \subseteq (d_i \cup \overline{D}(d_i))$ for $1 \le k < j$. Now $\overline{Q}_j = \{ x \} \cup \overline{Q}_{j-1}$, and we need only determine whether $\{ x \}$ is contained in $(d_i \cup \overline{D}(d_i))$.

[1] p_j is a calculation. Then

$N(x) \subseteq (\overline{Q}_{j-1}^R \cup R_{\sigma_{i-1}})$, which implies that

$N(x) \subseteq (\overline{D}(d_i) \cup d_i)$ by the inductive hypothesis and

the definition ofd_i, and consequently

$x \in \overline{D}(d_i)$.

[2] p_j is an input move. Then either

$x \in B_{\sigma_{i-1}}$, in which case $x \in (B_{\sigma_{i-1}} \cap \overline{Q}_j^R) \subseteq d_i$, or

$x \in \overline{Q}_{j-1}^B$, in which case $x \in (R_{\sigma_{i-1}} \cup \overline{Q}_{j-1}^R) \subseteq (d_i \cup \overline{D}(d_i))$.

The first instance follows from the definition of d_i; and the second from the definition of d_i, the fact that $\overline{Q}_{j-1}^R \subseteq \overline{Q}_{j-1}$, and the inductive hypothesis.

[3] p_j is an output move. Then

$x \in (R_{\sigma_{i-1}} \cup \overline{Q}_{j-1}^R) \subseteq (d_i \cup \overline{D}(d_i))$, which follows

again from the definition of d_i, the fact that $\overline{Q}_{j-1}^R \subseteq \overline{Q}_{j-1}$, and the inductive hypothesis.

□

Theorem 2: The set d_i is a dominator for $\hat{P}_i$.

proof: We want to show that $\hat{P}_i \subseteq \overline{D}(d_i)$. Using the previous lemma and the definition of a kernel gives us

$$\hat{P}_i \subseteq (d_i \cup \overline{D}(d_i)) \ .$$

We need only to establish that $\hat{P}_i \cap d_i = \varnothing$, which follows immediately from

$$\hat{P}_i \cap (R_{\sigma_{i-1}} \cup B_{\sigma_{i-1}}) = \varnothing \quad \text{and}$$

the definition of d_i.

□

2.3. The I/O Bound from the Maximum Kernel Size

A data dependency graph can be pebbled in many ways. The optimal pebbling, from an I/O cost standpoint, makes the fewest I/O pebble moves. While we cannot find the optimal pebbling strategy for an arbitrary lattice-graph computation, we can at least lower bound its I/O cost. The bound comes from an upper bound on the size of a sub-pebbling kernel. That is, since almost every sub-pebbling in a k-I/O-division contains exactly k I/O moves, and the kernels partition the nodes of the data dependency graph, dividing the number of nodes by the maximum size of a kernel leads to the desired bound on I/O cost.

Let P be a pebbling of a computation graph $G = (X, A)$ with input vertex set $I^G \subseteq X$ where the number of non-input vertices in G is $z = | X - I^G |$. For any k a k-I/O-division divides P into $h(k)$ pieces, $\{ P_1 , \ldots , P_{h(k)} \}$. The I/O cost, $C_{I/O}$, of pebbling P is given by

$$C_{I/O} = (h - 1) k + \varepsilon(k) \ , \quad \varepsilon(k) \in [0, k] \text{ where}$$

the last sub-pebbling has $\varepsilon(k)$ I/O moves. Suppose we have a lower bound $\hat{h}$ for h, then the I/O cost is bounded below:

$$C_{I/O} \geq (\hat{h}(k) - 1) k \ .$$

Bounding h can be done by finding some upper limit β on the size of a sub-pebbling kernel, $\hat{P}_i$. Then, because Theorem 1 tells us that $\{\hat{P}_i\}$ partitions $X - I^G$,

$$z = | X - I^G | = \sum_{i=1}^{h} | \hat{P}_i | \leq h \beta \ ,$$

and the desired bound for h is

$$\hat{h} = \frac{z}{\beta} .$$

The bound for the I/O cost of P becomes

$$C_{I/O} \geq \left\lceil \frac{z}{\beta} - 1 \right\rceil k .$$

By Theorem 2, $\hat{P}_i \subseteq \overline{D}(d_i)$, and the size of each kernel is bounded above by $|\overline{D}(d_i)|$. A suitable choice for β then is $\beta = \max|\overline{D}(d_i)|$ where the maximum is taken over all pebblings using at most r red pebbles. In general, β is a function, $\beta = \beta(r, k)$, of the number of red pebbles used by P, $r = \max_i |R_i|$, and k, the number of I/O moves per element of the k-I/O-division. The rest of this section develops bounds on β

In particular, the following subsections present a method of estimating the maximum size of the dependency of a set. Section 2.3.1 establishes the the necessary basis for assuming the dependency takes a general shape, the ''Symmetrization Theorem'', Theorem 3, and the ''One Level Theorem'', Theorem 4. Section 2.3.2 finds the particular shape and size for the discrete torus, and Section 2.3.3 does the same for the triangular lattice.

2.3.1. Bounding the Size of the Dependency of a Set

The previous section established the basis for making I/O estimates based on pebblings. In particular, a suitable estimate comes from bounding the maximum size of the dependencies of a special collection of sets, the kernel dominators. The bound on the maximum size of such a dependency is derived from (1) bounding the size of the dominator by some value m, and (2) deriving a tight isoperimetric inequality for $|\overline{D}(M)|$ for sets M with $|M| = m$. This isoperimetric problem is

(ISO-1)
Given a computation graph G and $m \in Z_+$, find a function f such that

$$f(m) \geq |\overline{D}(M)|$$

for any subset M of the vertices of G where $|M| = m$. (The symbol Z_+ represents the non-negative integers $= \{0, 1, 2, ,... \}$.)

□

A simple estimate can be made for part (1), and we leave it for later. Part (2) requires finding the shape of at least one extremal set $\overline{D}(M)$ for each value of m, and that problem is addressed in two parts: (a) we show that for every extremal set $\overline{D}(M)$ there is another extremal set which is a symmetrized version of $\overline{D}(M)$, and (b), we show that for any symmetrized extremal set there is another extremal set with the vertices of M compacted together into a special form. This special form will allow us to find an f such that $f(m) = |\overline{D}(M)|$ for any extremal set for (ISO-1) with $|M| = m$.

Although the results in the previous section apply to general data dependency graphs, in order to get these results we will restrict our attention in (a) to graphs that represent ''lattice-graph computations'' with the ''nested'' property, and in (b) add the further restriction to computations on the discrete torus. The types of data dependency graphs we will define as ''lattice-graph computations'' are layered directed graphs. In cellular automata, and particularly lattice-gas automata, values are associated with lattice points in an undirected lattice graph (see [1] for the precise definition of lattice graphs) and these values are updated in synchronous steps. Similar conditions also hold for many other iterated computations whose computation graphs may not be layered in the strict sense, but still have a structure similar to layering in that there is some connected undirected graph implicitly defined that can be used to define a layered subgraph. Theorem 2 in the previous section allows us to apply results for a graph to any of its supergraphs as well; consequently, we need only consider strictly layered graphs.

Dependency Set Symmetrization

This section will show that an extremal $\overline{D}(M)$ can be transformed to another extremal set whose layers are extremal sets in the undirected graph defining the data dependencies. We will rely heavily on recent results by Bollobás and Leader [27, 28] for extremal sets in undirected graphs, and our notation will attempt to conform to theirs whenever possible. In the following let $L = (V, E)$ be an undirected graph with vertex set V and edge set E.

Definition: The *Distance* $d(x, y)$ between a pair of vertices in an undirected graph $G = (V, E)$ is the number of edges in the shortest path connecting x and y. A vertex is connected to itself and $d(x, x) \equiv 0$.

□

Definition: The *Neighborhood,* ΛM, of a vertex subset $M \subseteq V$ is the set of vertices at most unit distance from M

$$\Lambda M = \{ v \in V \mid d(v, m) \leq 1 , m \in M \} \text{ , and}$$

$$\Lambda^k \equiv \Lambda(\Lambda^{k-1}) .$$

Definition: The *External Perimeter,* $\partial_{ex}M$, of M is the set of vertices at unit distance from M :

$$\partial_{ex}M = \Lambda M - M .$$

The *Internal Perimeter,* $\partial_{in}M$, is the subset of M adjacent to M's external perimeter:

$$\partial_{in}M = \Lambda M^c - M^c \text{ , where}$$

M^c is the set complement of M with respect to the set of vertices V: $M^c = V - M$.

The *Interior,* ∇M , is the subset of M which has no neighbors outside of M :

$$\nabla M = M - \partial_{in}M .$$

□

See figure 2.10. Equivalently, one can say that a vertex x is in the interior of M if $\Lambda x \subseteq M$.

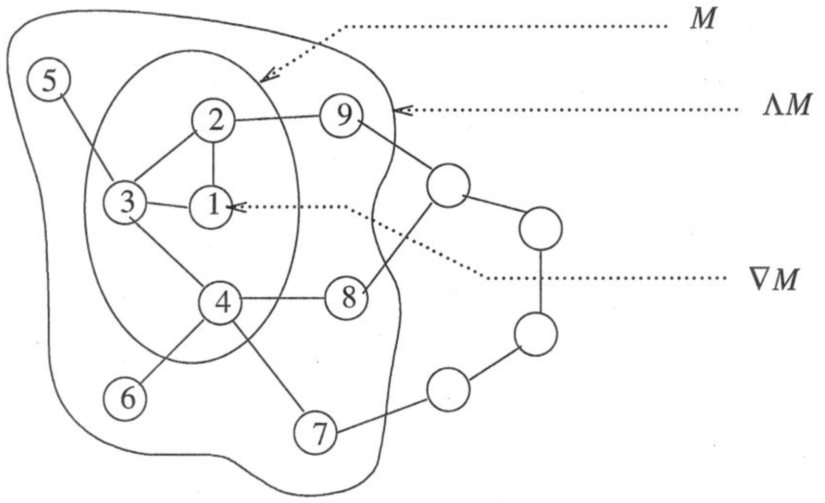


figure 2.10.

Sets defined in an undirected graph by a subset of the vertices $M = \{1, 2, 3, 4\}$. *The interior is* $\nabla M = \{1\}$, *the internal perimeter is* $\partial_{in}M = \{2, 3, 4\}$, *the external perimeter is* $\partial_{ex}M = \{5, 6, 7, 8, 9\}$, *and the neighborhood is* $\Lambda M = \{1-9\}$.

To clarify the relationships between these set operators, note first that, generally Λ is not a left inverse of ∇. That is, $\Lambda\nabla M$ may be a proper subset of M. Another way of saying the same thing is to say that $\partial_{ex} \nabla M$ may be a proper subset of $\partial_{in} M$.

Let $G = (X, A)$, be a data dependency graph for some computation. Let the inputs of G, those vertices in X with in-degree zero, be I^G. Recall that $D^i(M)$ is the subset of X that is the i^{th} iteration of the dependency operator D operating on M. Thus, $D^i(I^G)$ is the set of vertices at the i^{th} level of a breadth-first search from the input vertices of G.

Definition: The i^{th} *Layer* of G is

$$G_i = D^i(I^G) \quad \text{, where}$$

we define $G_0 \equiv I^G$. We say that G has k layers if k is the least i such that $D^i(I^G) = \varnothing$.

□

Definition: The *Sections* S_i of a set S, where S is a subset of the vertices of a computation G, are the subsets of S contained in the layers of G:

$$S_i \equiv S \cap G_i \, .$$

□

We are now ready to define a type of layered data dependency graph we will call a lattice-graph computation. The idea behind the definition is simply a formalization of the structure of the data dependency graph of a lattice-gas simulation. In the data dependency graph of a lattice-gas simulation, each layer can be augmented by undirected edges to form an undirected component identical to the lattice-graph defining the lattice-gas automaton. These undirected edges define the arcs from layer to layer: the undirected neighbors of a node in one layer form the directed support neighborhood of the corresponding node in the next layer down. The following two definitions state the above formally.

Definition: A *Layer Stencil Graph* of a data dependency graph G, whose non-empty layers have equal numbers of nodes, is any undirected graph $L = (V^L, E^L)$, with vertices V_L and edges E^L, such that the number of vertices in V_L is equal to the number of vertices in any non-empty G_i.

Given a fixed layer stencil graph L and a fixed set of functions $\{\phi_i\}$ such that ϕ_i maps G_i to V^L bijectively, the i^{th} *Layer Graph* of G is the undirected graph $L_i = (G_i, E_i)$, with vertices G_i and edges E_i, such that ϕ_i is a graph isomorphism between L and L_i.

□

Assuming a fixed layer stencil graph L and fixed mapping $\{\phi_i\}$ we can use the following notation for nodes and their neighborhoods. If x is a node in the layer stencil graph L, $x \in V^L$, we write x_i for the vertex in L_i that maps to x; if s is a subset of vertices in V^L we write s_i for the corresponding vertices in L_i; Λx_i is taken to be the undirected neighborhood of x_i in layer graph L_i: $\Lambda x_i = (\Lambda x)_i$.

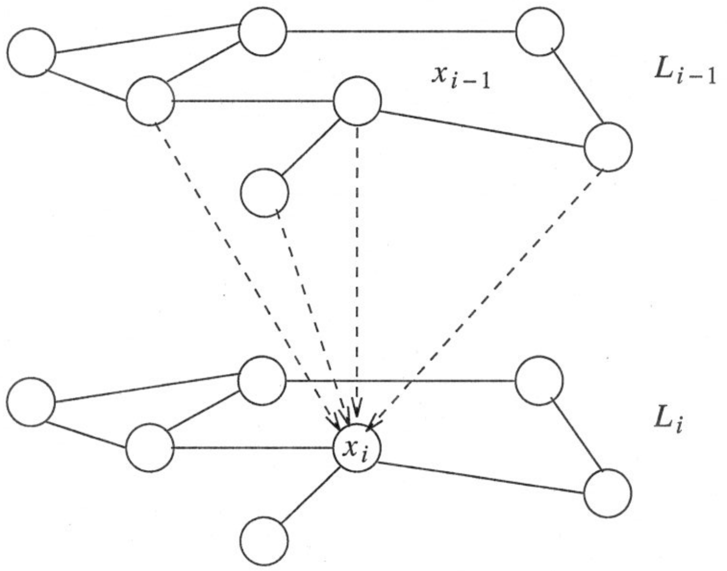


figure 2.11.

The mapping between layers in a lattice-graph computation. The undirected graph L *defines the data dependencies between every consecutive pair of layers in the data dependency graph. Dashed lines represent data dependency arcs.*

We now give the definition of a lattice-graph computation. See figure 2.11. Recall that if y is a vertex in a data dependency graph, then y's support neighborhood $N(y)$ is the set of vertices in the graph with arcs going to y.

Definition: A computation graph G is a *Lattice-Graph Computation* if we can find some layer stencil graph and corresponding set of layer graphs L_i such that the following holds,

$$N(x_i) = \Lambda x_{(i-1)} ,$$

for every x_i and for every i greater than zero.

□

We will want to speak of the image of a section in a different layer. So, if S_i is a subset of the vertices of layer G_i, we write $(S_i)_j$ for the set of vertices in layer G_j that are identified with the S_i via the layer-graph L.

Before we go further we want to emphasize a simple property of dependencies in lattice-graph computations. Because all the data dependency arcs go between layers and the neighborhoods of vertices map uniformly between layers, nothing can be computed in one layer that is not part of the interior of the set of known or computed information in the previous layer. The following lemma formalizes this statement.

Lemma 2: If G is a lattice-graph computation, M is a vertex subset of G, and $S = M \cup \overline{D}(M)$, then the i^{th} section of the dependency of M lies in the image of the interior of S_{i-1}:

$$\overline{D}(M)_i \subseteq (\nabla S_{i-1})_i .$$

Proof: If $\overline{D}(M)_i$ is empty the result is immediate, so suppose it has at least one member x_i. Suppose, in contradiction to the hypothesis, that

$$x_i \notin (\nabla S_{i-1})_i \text{ , or equivalently,}$$

$$x_{i-1} \notin \nabla S_{i-1} .$$

By definition then, the neighborhood of x_{i-1} contains some element not in S_{i-1}, that is

$$\Lambda x_{i-1} \not\subseteq S_{i-1} . \qquad (1)$$

Since G is a lattice-graph computation,

$$N(x_i) = \Lambda x_{i-1} , \qquad (2)$$

and consequently the support neighborhood of x_i is not contained in M or anything computable from M. That is, substituting (2) into (1) and using the definition of S_{i-1} we have,

$$N(x_i) \not\subseteq M_{i-1} \cup (\overline{D}(M))_{i-1} ,$$

or

$$N(x_i) \not\subseteq M \cup \overline{D}(M) ,$$

contradicting the assumption that $x_i \in \overline{D}(M)$.

□

The above proof works identically if the hypothesis is changed so that, instead of assuming that

$$S = M \cup \overline{D}(M) ,$$

one only assumes that

$$S_{i-1} = (M \cup \overline{D}(M))_{i-1} .$$

This is the form of Lemma 2 we will use in the proof of Theorem 3. An obvious, but nevertheless useful, corollary lets us assume the ''shadow'' of the previous section is always in the dependency.

Corollary: If $x_i \in (\nabla S_{i-1})_i$ and x_i is not in the i^{th} section of M, then

$$x_i \in \overline{D}(M) .$$

The corollary follows from the observation that $\Lambda x_{i-1} \subseteq S_{i-1}$.

□

We now describe the ''nested'' property for lattice-graph computations. This property allows us to assume that extremal sets in the layer-graph form a sequence of sets that are ordered by set inclusion. Recall that a vertex subset S in an undirected graph consists of its interior points, ∇S, and its interior perimeter $\partial_{in} S$.

Definition: A lattice-graph computation G with layer-graph L has the *Nested Property* if

(1) there exists a sequence $\{B(x)\}$ of extremal sets for the undirected graph L for the following isometric problem on L:

(ISO-2)
Given x a non-negative integer, find $B(x)$ a vertex subset in L such that

$$|B(x)| = x , \text{ and}$$
$$|\nabla B(x)| \geq |\nabla S|$$

for any S with $|S| = x$. (Such a set $B(x)$ will be called a ∂_{in}-*extremal* set because it has a minimum internal perimeter.)

(2) the sequence $\{B(x)\}$ has the property that the interiors of the extremal sets form a linear order under set inclusion:

$$\nabla B(x) \subseteq \nabla B(x+1) .$$

The sequence $\{B(x)\}$ is called a *nested extremal sequence* for L.

□

In general, the sequence $\{B(x)\}$ is assumed fixed in any discussion of a nested, lattice-graph computation G.

With our data dependency graphs restricted to lattice-graph computations with the nested property, we can proceed to show that for any set M we can find a ‘‘symmetrical’’ set whose dependency is at least as large as $\overline{D}(M)$. A ‘‘symmetrical’’ set has every section in the form of a nested extremal set.

Definition: A vertex subset S of a nested, lattice-graph computation G whose layer-graph L has a nested extremal sequence $\{B(x)\}$, is *Section-symmetric* if every section S_i of S is an element of the nested extremal sequence $\{B(x)\}$.

A vertex subset S' is a *Section-symmetrization* of set S if S' is section-symmetric and each section of S' has the same number of vertices as the corresponding section of S, that is:

$$\begin{aligned} |S'_i| &= |S_i| = x \text{ , and} \\ S'_i &= B(x) \text{ .} \end{aligned}$$

□

We are now ready to take the first step in identifying a class of (ISO-2) extremal sets in lattice-graph computations that have a simple formula for the size of their dependencies. The following theorem establishes that any vertex subset of a nested, lattice-graph computation can be transformed into a set that generates a section-symmetric dependency that is at least as large as the dependency of the original set. This class is extremal, but does not yield a simple formula for the size of the dependencies. The proof uses the notion that, in any section of the dependency set, the elements of the extremal set always lie in the exterior perimeter or beyond.

Theorem 3: If S' is a section-symmetrization of S, where

$$S = M \cup \overline{D}(M)$$

and M is any vertex subset of a nested, lattice-graph computation G, then there exists a vertex subset M' of G such that

$$\begin{aligned} S' &= M' \cup \overline{D}(M') \text{ , and} \\ |M'| &\le |M| \text{ .} \end{aligned}$$

Proof: In the following let layer index i_0 be the least i for which $M_i \neq \varnothing$. Assign vertices of S' to M' as follows:

$$M'_i = \begin{cases} S'_i & \text{, if } S'_{i-1} = \varnothing \\ S'_i - (\nabla S'_{i-1})_i, & \text{if } S'_{i-1} \neq \varnothing \end{cases}$$

We proceed by first establishing that this assignment of vertices to M' results in $S' = M' \cup \overline{D}(M')$. This is done by induction on the layer index i with the hypothesis:

$$S'_i = M'_i \cup \overline{D}(M')_i . \tag{1}$$

(basis)

Since $M'_i = \varnothing$ for all $i < i_0$, then $\overline{D}(M')_i = \varnothing$ for all $i < i_0$. In particular, when $i = i_0$,

$$M'_{i-1} \cup \overline{D}(M')_{i-1} = \varnothing ,$$

from which it follows that

$$\overline{D}(M')_i = \varnothing .$$

By the definition of S' we know that $S'_i = \varnothing$ for all $i < i_0$. Consequently, again setting $i = i_0$, the definition of M' gives us

$$S'_i = M'_i = M'_i + \overline{D}(M')_i .$$

So, (1) holds for $i \leq i_0$.

(induction)

Suppose (1) holds for i. By Lemma 2

$$\overline{D}(M')_{i+1} \subseteq (\nabla S'_i)_{i+1} .$$

Since M'_{i+1} and $(\nabla S'_i)_{i+1}$ are disjoint by the definition of M', the corollary to Lemma 2 gives us

$$(\nabla S'_i)_{i+1} \subseteq \overline{D}(M')_{i+1} ,$$

that is,

$$\overline{D}(M')_{i+1} = (\nabla S'_i)_{i+1} . \tag{2}$$

Writing S'_{i+1} as

$$S'_{i+1} = (S'_{i+1} - (\nabla S'_i)_{i+1}) + (\nabla S'_i)_{i+1} , \text{ and}$$

substituting for the two terms on the right using the definition of M'_{i+1} and (2) we get

$$= M'_{i+1} + \overline{D}(M')_{i+1}$$

which is the desired result for the $(i+1)$ section of S'.

Summing (1) over *i* gives the first part of the hypothesis. It remains to be shown that

$$|M'| \quad \leq \quad |M| \ .$$

Lemma (2) applied to *S* tells us that

$$\bar{D}(M)_{i+1} \quad \subseteq \quad (\nabla S_i)_{i+1} \ ,$$

and, consequently, it follows that

$$|\bar{D}(M)_{i+1}| \quad \leq \quad |(\nabla S_i)_{i+1}| \quad = \quad |\nabla S_i| \ .$$

Because the sets in $\{B(x)\}$, the extremal sequence for *G*'s layer graph, are extremal, and because $S_i' \equiv B(|S_i|)$ we have

$$|\nabla S_i| \quad \leq \quad |\nabla S_i'| \quad = \quad |(\nabla S_i')_{i+1}| \ .$$

In the previous induction it was shown that

$$(\nabla S_i')_{i+1} \quad = \quad \bar{D}(M')_{i+1} \ .$$

Combining the previous three statements results in

$$\begin{aligned} |\bar{D}(M)_{i+1}| \quad \leq \quad |(\nabla S_i)_{i+1}| \quad &= \quad |\nabla S_i| \\ &\leq \quad |\nabla S_i'| \\ &= \quad |(\nabla S_i')_{i+1}| \\ &= \quad |\bar{D}(M')_{i+1}| \ . \end{aligned}$$

That is,

$$|\bar{D}(M)_j| \quad \leq \quad |\bar{D}(M')_j| \ , \tag{3}$$

for all *j*.

Because *M* and $\bar{D}(M)$ are disjoint for any set *M*, and because

$$S \quad = \quad M \quad \cup \quad \bar{D}(M) \ ,$$

we have

$$|S_i| \quad = \quad |M_i| \quad + \quad |\bar{D}(M)_i| \ .$$

Similarly, since (1) holds for S', we have

$$|S_i'| \quad = \quad |M_i'| \quad + \quad |\bar{D}(M')_i| \ .$$

From the definition of S' we know that $|S_i| = |S_i'|$, which gives us

$$| M_i | \quad + \quad | \overline{D}(M)_i | \quad = \quad | M'_i | \quad + \quad | \overline{D}(M')_i | \ , \text{ and}$$

applying (3) to this identity we have

$$| M_i | \quad \geq \quad | M'_i | \ . \tag{4}$$

Summing (4) over i gives the desired result. Also, taking (3) and (4) together and summing shows that M' is (ISO-1)-extremal.

□

We have thus far established a canonical form for (ISO-1)-extremal sets M in a nested, lattice-graph computation. Unfortunately, the shape of the set is still not well enough defined to allow an easy estimate of the size of its dependency. A natural and intuitive idea is that all the elements of an (ISO-1)-extremal set can be moved to a single layer. In that case, the size of the dependency can be found easily since it is the sum of powers of the interior operator ∇ operating on the set M. The next section shows that for two specific layer-graphs this can be done.

Moving the Supporting Set into a Single Layer

Theorem 3 showed that if M is an (ISO-1)-extremal set in a nested lattice-graph computation G with layer-graph L, then the set $S = M \cup \overline{D}(M)$ could be assumed to be in a form where every section S_i is a ∂_{in}-extremal set for L. That is, an (ISO-1)-extremal set M has every section in the shape of an annulus between two ∂_{in}-extremal sets. In this section we will show a sufficient condition on ∂_{in}-extremal sets for L which, when satisfied, allows us to assume S can be further transformed so that the support set M lies in a single layer in a ∂_{in}-extremal form. This will allow us to find a convenient expression for the value of the size of the maximum dependency of any set M.

We will proceed by supposing that S has more than one section containing elements of M. Taking the last section S_i that contains elements of M, we move M_i up to the next layer G_{i-1} (see figure 2.12). The result gives a condition on the extremal function $| \nabla B(x) |$ sufficient to assure the transformed set M' has at least as large a dependency as M. Iterating the process brings all the elements of M to the top level. Thus, if the condition on $| \nabla B(x) |$ is satisfied, all the elements of M can be moved to the same layer containing S_0 and the result has the maximum dependency possible for a support set of size $| M |$.

Since we will assume that the sections of S are in the form of ∂_{in}-extremal sets $\{ B(x) \}$, we only need to consider the sizes of sets. Consequently, we will find it convenient to use the notation in table 2.2. The entries there are self-explanatory except

Notation		
m_i	$\equiv$	$\mid M_i \mid$
d_i	$\equiv$	$\mid \overline{D}(M)_i \mid$
$\nabla^i(x)$	$\equiv$	$\mid \nabla^i B(x) \mid$
$\Lambda(x)$	$\equiv$	$\mid \Lambda B(x) \mid$
$V(m)$	$\equiv$	$\mid M \cup \overline{D}(M) \mid$

table 2.2.

for the definition of the ''volume'' of a set of a given size, $V(m)$. If we suppose M is a standard sphere, that is, M is a subset of a single layer G_i and $M = B(m)$ with $m = \mid M \mid$, then $V(m)$ can be written as,

$$V(m) \quad = \quad \sum_{i=0} \nabla^i(m) \quad = \quad \sum_{i=0} \mid \nabla^i B(m) \mid \; .$$

We will not use the notation $V(m)$ in any situation where M is not a ∂_{in}-extremal set, so this equation can serve as an alternate definition of $V(\,)$.

We now give a sufficient condition for assuming that any (ISO-1)-extremal set M is equivalent to a standard sphere of the same size.

Theorem 4: Suppose G is a nested lattice-graph computation with layer graph L whose nested ∂_{in}-extremal sequence is $\{B(x)\}$, and let M be any (ISO-1)-extremal set in G. If the following condition holds for the function $\nabla(x)$ defined by the nested sequence $\{B(x)\}$,

Given integers a, b, and c, with a and b positive and c non-negative, such that,

$$b \geq \nabla(c) \quad \text{, and}$$

$$a - b \geq c - \nabla(c) \; ,$$

then

$$\nabla(a) - \nabla(b) \geq \nabla(c) - \nabla^2(c) \; .$$

then there exists a set M' with $|M'| = |M|$ which is also (ISO-1)-extremal, M' lies in a single layer of G, and $M' = B(|M'|)$.

Proof: We will work backwards by assuming M' is (ISO-1)-extremal and discovering a sufficient condition. Since G is a nested lattice-graph computation, we can assume that the sections of $S = M \cup \overline{D}(M)$ are elements of $\{B(x)\}$ by Theorem 3. Suppose there are at least two sections of S containing elements of M. Let S_i be the section with greatest index containing elements of M, and let S_j be the section from $S - S_i$ with greatest index containing elements of M (see figure 2.12).

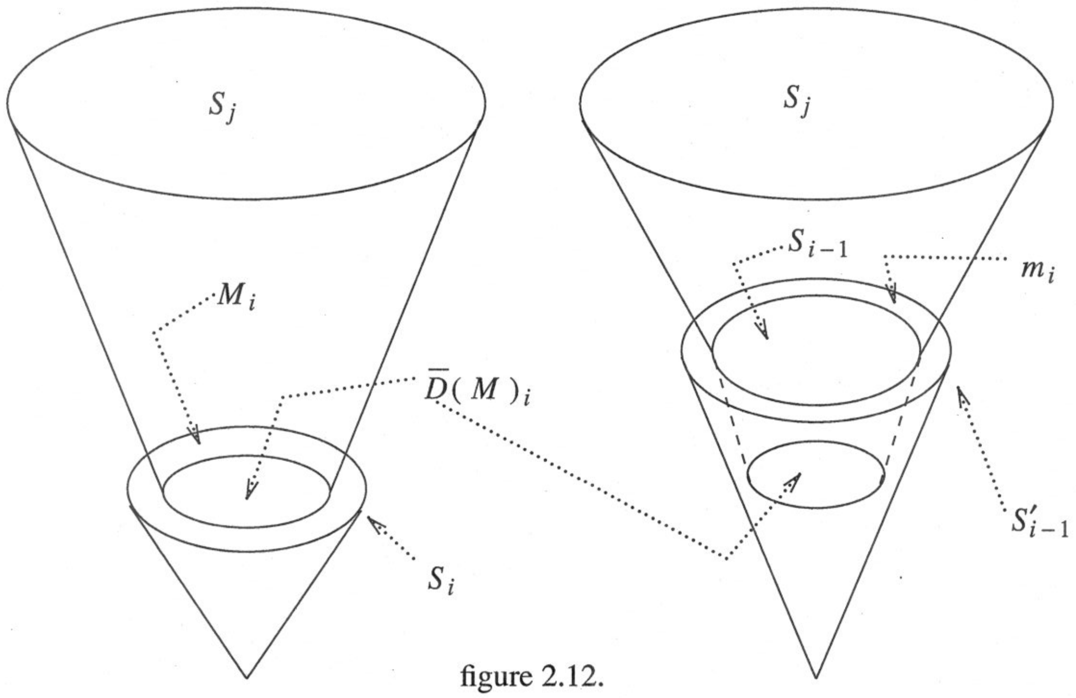


figure 2.12.

Transforming the last section of M. The left figure shows the lowest and second lowest sections containing elements of M (the sections of S with the largest indices containing elements of M), with their dependencies. The right figure shows the effect of moving the m_i vertices of M_i up from section S_i and adding them to the perimeter of section S_{i-1}. If this new support M' is also (ISO)-extremal, then the new dependency must be at least as large as M's. The only sections that change under the transformation are the sections at $i-1$ and below. Thus, the "volume" of M''s lower cone minus its top (S'_{i-1}) must be at least as large as the "volume" of the cone at S_i minus M_i.

Let M' be formed from M by removing M_i and adding m_i points to S_{i-1}: that is, let

$$S'_{i-1} = B(|S_{i-1}| + m_i),$$

by setting

$$M'_{i-1} = B(|S_{i-1}| + m_i) - \overline{D}(M)_{i-1}.$$

Clearly, $|M'| = |M|$ since $M_{i-1} = S_{i-1} - \overline{D}(M)_{i-1}$. If the resulting dependency is at least as large as $|\overline{D}(M)|$, then M' is also (ISO-1)-extremal.

Since M and M' are identical in any section above S_{i-1}, in comparing the dependencies of M and M', we only need to consider the parts below the $(i-1)^{th}$ section. That is, we want to compare the old dependency in the lower section of S with the new dependency in the lower section of S'. The first quantity is the volume of the cone formed from S_i but excluding M_i,

$$\left| \bigcup_{t \geq i} \overline{D}(M)_t \right| = V(|S_i|) - |M_i| \tag{1}$$

$$= V(m_i + d_i) - m_i \ .$$

Similarly, the second quantity is the volume of the cone formed by S'_{i-1} but excluding S'_{i-1},

$$\left| \bigcup_{t \geq i} \overline{D}(M')_t \right| = V(|S'_{i-1}|) - |S'_{i-1}| \ . \tag{2}$$

Since we know that M is extremal, we have $\overline{D}(M)_i = (\nabla S_{i-1})_i$, which gives us $(\Lambda \overline{D}(M)_i)_{i-1} \subseteq S_{i-1}$, which we can write equivalently as $\Lambda(d_i) \leq |S_{i-1}|$. Now, because $\nabla(x)$ is monotonic[†], we can substitute for the right-hand side of (2) to get,

$$\left| \bigcup_{t \geq i} \overline{D}(M')_t \right| \geq V(m_i + \Lambda(d_i)) - (m_i + \Lambda(d_i)) \ . \tag{3}$$

Combining (1) and (3) we get

$$V(m_i + \Lambda(d_i)) - V(m_i + d_i) \geq \Lambda(d_i) \ .$$

Because m_i and d_i are independent of each other, and because $\nabla \Lambda d_i = d_i$, we can rewrite this last inequality as,

$$V(x + y) - V(x + \nabla(y)) \geq y \ . \tag{4}$$

Inequality (4) gives a condition which, when satisfied, shows that M' has at least as large a dependency as M. However, we can simplify this further. Let $t_{(x+y)}$ be the

† Since $\nabla()$ is monotonic, and $V(m) = \sum \nabla(m)$, then $V(x) - x \geq V(y) - y$ whenever $x \geq y$. Setting $x = |S_{i-1}| + m_i$ and setting $y = \Lambda d_i + m_i$ gives the inequality on the right-hand side of (3).

greatest t such that $\nabla^t(x + y) > 0$, and similarly define $t_{(x+\nabla y)}$ and t_y. Now, rewriting (4) by replacing $V()$ with a summation in terms of ∇, and replacing y by the sum $(y - \nabla(y)) + (\nabla(y) - \nabla^2(y)) + \cdots$, gives us,

$$\sum_{t=0}^{t_{(x+y)}} \nabla^t(x + y) - \sum_{t=0}^{t_{(x+\nabla y)}} \nabla^t(x + \nabla y) \geq \sum_{t=0}^{t_y} (\nabla^t(y) - \nabla^{t+1}(y)) .$$

Since $t_y \leq t_{(x+\nabla y)} \leq t_{(x+y)}$, we can take the summation over all terms simultaneously, giving,

$$\sum_{t=0}^{t_{(x+y)}} \left[\nabla^t(x + y) - \nabla^t(x + \nabla y) \geq (\nabla^t(y) - \nabla^{t+1}(y)) \right] . \quad (6)$$

Suppose we decide to establish this last condition by induction on the terms of the sum. Since $\nabla^0(x) = x$, the basis case is immediate:

$$(x + y) - (x + \nabla y) \geq y - \nabla y .$$

The inductive step would then be satisfied if

$$\nabla^t(x + y) - \nabla^t(x + \nabla y) \geq (\nabla^t(y) - \nabla^{t+1}(y)) \text{ , implies} \quad (7)$$
$$\nabla^{t+1}(x + y) - \nabla^{t+1}(x + \nabla y) \geq (\nabla^{t+1}(y) - \nabla^{t+2}(y)) .$$

Substituting a for $(x + y)$, substituting b for $(x + \nabla y)$, and substituting c for y, we can rewrite this last statement as,

$$a - b \geq c - \nabla(c) \Rightarrow \nabla(a) - \nabla(b) \geq \nabla(c) - \nabla^2(c) . \, (8)$$

Because of the monotonicity of $\nabla()$, it follows from (7) that we only need to verify this last inequality when $a \geq c$ and $b \geq \nabla(c)$. Of course, if the left-hand side of (8) holds, then it follows from $b \geq \nabla(c)$ that $a \geq c$. We then have the condition stated in the hypothesis.

□

2.3.2. Size of the Dependency for The Discrete Torus

Here we show that any lattice-graph computation whose layer graph is a two-dimensional discrete torus is nested and we show a nested extremal sequence satisfying the condition stated in Theorem 4. Once that has been established we will give the maximum size of a dependency as a function of the size of the ∂_{in}-extremal set. We will restrict the sizes of the (ISO-1)-extremal sets to be less than a fixed fraction of the number of nodes in a single layer. This restriction limits the generality of the results in this section, but suffices for the application of these results to the architecture problem.

We will use some techniques developed by Bollobás and Leader [27] as they apply to the results of Wang and Wang [29, 30]. The main result we use comes from [28] which can be used to give, among other things, the shapes of ∂_{in}-extremal sets in the discrete torus.

The discrete torus is a *d*-dimensional grid graph wrapped around a *d*-dimensional toroid.

Definition [28]: the *d-dimensional mod-k Discrete Torus* Z_k^d is the undirected graph on the finite lattice of the (mod *k*) *d*-dimensional euclidian integers $(Z/kZ)^d$ for $k \geq 2$ and $d \geq 1$ formed by connecting vertices *x* and *y* if and only if *x* and *y* differ in exactly one component by ± 1:

$$Z_k^d \quad = \quad (V, E) \text{ , where}$$

$$V \quad = \quad \{ x \in Z^d \mid 0 \leq x_i < k \text{ , for } 1 \leq i \leq d \} \text{ , and}$$

$$E \quad = \quad \left\{ \{ x, y \} \mid x_i = y_i \pm 1 \text{ for some } i \text{ , and } x_j = y_j \text{ for } j \neq i \right\} .$$

The graph for the *d*-dimensional infinite ($k \rightarrow \infty$) torus, or equivalently the *d*-dimensional infinite grid, is written Z^d.

□

A Nested Extremal Sequence for the Two-Dimensional Discrete Torus

We first want to show that if G is a lattice-graph computation with layer-graph $L = Z_k^d$, then G is nested as well. That is, we wish to show that there exists a sequence $\{ B(x) \}$ of ∂_{in}-extremal sets for Z_k^n such that $\nabla B(x) \subseteq \nabla B(x+1)$. Actually, it is not necessary to confirm that the $B(x)$ are nested extremal sets for x greater than the number of points in any section of the union of a dependency and its supporting set. In the application of these results to bounding throughput we will make the assumption that the size of the set whose dependency we want to estimate is small compared with the number of nodes in the layer graph. Consequently, in the following we will assume any sets in the discrete torus we have interest in have a maximum diameter less than $k/2$. Thus, we can think of the torus as an infinite grid, or a discrete torus, depending on which is more convenient.

The nested extremal sequence we will use is a simple rearrangement of the sets Wang and Wang showed were extremal for the infinite grid graph [30]. In that paper they called their sets ''standard spheres of size *v*,'' *v* referring to the number of vertices in the set. The sequence of subsets of Z^n that we propose to show is a nested extremal sequence we will also call a sequence of standard spheres of size v. It should be noted that the isoperimetric problem addressed in [28] and [29] is not the same as the ∂_{in}-minization problem. The problem addressed in those two papers concerns minimizing ∂_{ex} for a set of given size. However, since this problem is very closely related to ∂_{in}-minimization, attention to a few details is sufficient to apply their results. In particular, the authors in [29] give a ∂_{ex}-extremal nested sequence for Z^d. It is easy to see that if S is a ∂_{ex}-extremal set, then ΛS is ∂_{in}-extremal. If we could show that every $B(x)$ in Wang and Wang's extremal sequence was equal to $\Lambda B(j)$ for some $j < x$, then we would know that the sequence was also ∂_{in}-extremal. Unfortunately, this is not true. We therefore need to prove that those elements in the sequence which are not the neighborhood of a previous element, that is, not Λ of a previous element in the sequence, are also ∂_{in}-extremal. Unfortunately, the nature of Wang and Wang's sequence makes this somewhat more difficult than starting from scratch with a new sequence of sets and proving that this new sequence is ∂_{in}-extremal. Also, this new sequence is somewhat more amenable to summing the sizes of its elements. Consequently, we now introduce our version of standard spheres in Z^2, and then proceed to show that they form a nested ∂_{in}-extremal sequence (see figure 2.13).

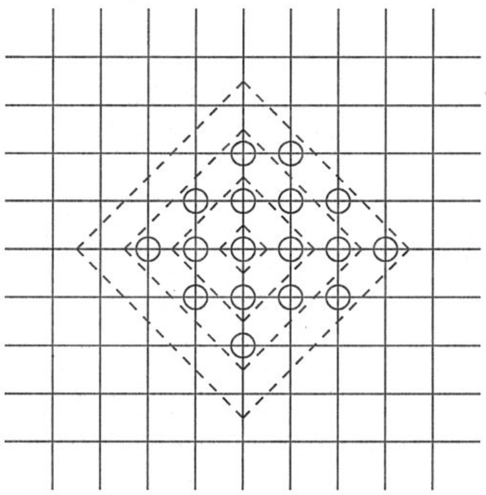


figure 2.13.

A standard sphere. The circled vertices indicate the vertices in the standard sphere $B(v_2 + 4)$. The vertices between adjacent concentric dashed diamonds constitute a shell. Shown are the shells $Shell_i$ for $i = 0, 1, 2, 3$. $B(v_2 + 4)$ has four vertices in its partially filled outer shell, $Shell_3$.

Definition: the *Standard Spheres of size* x in the undirected graph Z^2 are the sets $\{B(x)\}$ defined by the following. Let

$$Shell_r \quad = \quad \{x \in Z^2 \mid d(x, \mathbf{0}) \ = \ r\} \quad \text{, where}$$

$\mathbf{0}$ is the origin (0, 0) of Z^2 and the distance $d()$ is graph distance. Let

$$v_r \quad = \quad \left| \bigcup_{i=0}^{r} Shell_i \right| \quad .$$

Note that $v_{r+1} = v_r + 4(r+1)$. We then define

$$B(0) \quad = \quad \varnothing \quad \text{, and}$$

$$B(v_r) \quad = \quad \bigcup_{i=0}^{r} Shell_i \quad .$$

Define the *Spiral Order* on the elements of $Shell_r$ by setting $x_1 \ = \ (1, r-1)$ and setting x_{i+1} to be the element in $Shell_r$ that a half line with endpoint at $\mathbf{0}$ sweeping clockwise intersects next after x_i. Then, if $1 \leq z < 4(r+1)$,

$$B(v_r + z) \quad = \quad B(v_r) \cup \{ \text{ the first } z \text{ elements in the spiral order of } Shell_{r+1} \} \quad .$$

□

We now show that the standard spheres form a nested ∂_{in}-extremal sequence.

Theorem 5: Let $\{B(x)\}$ be the sequence of standard spheres in Z^2. This sequence is nested, that is, for every $x \geq 0$,

$$\nabla B(x) \quad \subseteq \quad \nabla B(x+1) \quad \text{, and}$$

each element of the sequence is ∂_{in}-extremal. That is, if A is any subset of vertices in Z^2 with

$$|A| \quad = \quad x \quad \text{, then}$$

$$|\nabla A| \quad \leq \quad |B(x)| \quad .$$

Proof: The proof will proceed by proving a series of simple lemmas. Let $\{S(x)\}$ be any sequence of ∂_{in}-extremal sets for Z^2, and let $f(x) \ = \ |\nabla S(x)|$. We will show that the standard spheres are extremal at $x = v_r$, that is, we will show that $\nabla B(v_r) \ = \ f(v_r)$. Then, we will show that the standard spheres at $(v_r + j(r+1) + 1)$ are extremal for $0 \leq j \leq 3$, from which it will follow that the standard spheres are extremal for all x except at $x = v_r + j(r+1)$ for $1 \leq j \leq 3$. Finally, we will handle this remaining case. We begin with a general observation on the function $f(x)$.

Definition: A *Loose Vertex* v is a vertex in any subset of vertices M of an undirected graph that is not connected to the interior of M. That is,

$$v \in \partial_{in} M \text{ , and}$$

$$v \notin \Lambda \nabla M \text{ .}$$

□

Lemma 3: (No-Jumps Lemma) If $\{S(x)\}$ is a sequence of ∂_{in}-extremal sets for Z^2, then $f(x+1) \leq f(x) + 1$, that is,

$$|\nabla S(x+1)| \leq |\nabla S(x)| + 1 \text{ .}$$

Proof: Suppose by contradiction that,

$$|\nabla S(x+1)| > |\nabla S(x)| + 1 \text{ .}$$

First, if $S(x+1)$ has any loose vertices, then $\nabla S(x+1) = \nabla S(x)$, since if we remove the loose vertex the interior of $S(x+1)$ is not affected and the resulting set has total size x which is the same as the size of $S(x)$. That is, since $S(x)$ is extremal, $\nabla S(x)$ is at least as large as this new set's interior. On the other hand, since any set with an interior at least as large as $\nabla S(x)$ could be formed by adding arbitrary vertices to $S(x)$, the interior of $S(x+1)$, which is also the interior of the new set, is at least as large as $\nabla S(x)$.

So $S(x+1)$ has no loose vertices. Consequently, every $v \in \partial_{in} S(x+1)$ has at least one neighbor in $\nabla S(x+1)$, which is to say that $S(x+1) = \Lambda \nabla S(x+1)$ and $\partial_{in} S(x+1) = \partial_{ex} \nabla S(x+1)$. Suppose such a v has only one neighbor in $\nabla S(x+1)$. Then removing v decreases the interior of the resulting set to $|\nabla S(x+1)| - 1$. Since the resulting set has size x, this contradicts that $S(x)$ is extremal. So every $v \in \partial_{in} S(x+1)$ has at least two neighbors in $\nabla S(x+1)$. We will next show that this last statement is a contradiction, that is, there must be some $v \in \partial_{in} S(x+1)$ with at most one neighbor in $\nabla S(x+1)$.

Let $v = (v_1, v_2)$ be the vertex in $S(x+1)$ whose component sum $(v_1 + v_2)$ is at least as large as the component sum for any other element in $S(x+1)$ and whose second component, v_2, is at least as large as the second component of any other element's. The vertex v is then the furthest "upper right" element in $S(x+1)$. Two of v's neighbors, $v + (1, 0)$ and $v + (0, 1)$, cannot be in $S(x+1)$ because v was chosen to have the largest component sum. The vertex $v + (-1, 1)$ cannot be in $S(x+1)$ because v's second component is maximum, and consequently, v's neighbor $v + (-1, 0)$ cannot be in $\nabla S(x+1)$. Since we have accounted for three of v's neighbors, v has at most one neighbor in $\nabla S(x+1)$.

□

Lemma 4: The standard sphere $B(x)$ is ∂_{in}-extremal for $x = v_r$.

Proof: Corollary 6 to Theorem 4 in [28] can be rewritten as follows:

$$|A| \quad < \quad |B(v_r)| \quad \Rightarrow \quad |\nabla A| \quad < \quad |\nabla B(v_r)| \quad .$$

Consequently, $f(v_r - 1) \ < \ |\nabla B(v_r)|$. That is, $f(v_r - 1) \ \leq \ |\nabla B(v_r)| - 1$. Because of the No-Jumps Lemma (Lemma 3) $f(v_r) \ \leq \ f(v_r - 1) + 1$. Combining these two statements we have $f(v_r) \ \leq \ |\nabla B(v_r)|$, which states that $B(v_r)$ is ∂_{in}-extremal.

□

At this point we want to show that the standard spheres $B(x)$ are extremal for $x = v_r + j(r+1) + y$ with suitable restrictions on j and y. The proof will be a sort of induction on j. That is, as can be seen in figure 2.14, the function $g(x) = |\nabla B(x)|$ is naturally broken into sections at $x = v_r + j(r+1)$ for $j = 0, 1, 2, 3, 4$. In that figure, the circles indicate the value of $g(x)$ at the points $x = v_r + j(r+1) + 1$. The next lemma will show that if g is optimal at $x - 2$ (the points marked with triangles) or $x - 1$ (the squares), then g is optimal at x (the circled points). Once this is established, since $g(v_r)$ is already known to be optimal, we can conclude that the first circled point is optimal. By Lemma 3 (No-Jumps Lemma), it then follows that every point up to and including the next triangle marked point is also optimal, and induction on j shows that g is optimal at all x except at the points $x = v_r + j(r+1)$. These last remaining points will require a separate treatment.

Figure 2.15 shows the situation at one of the knees of the function $g(x)$. Recall that if $\{A(x)\}$ is any sequence of extremal sets, then the optimal values $|\nabla A(x)|$ are written $f(x)$. Suppose in figure 2.15 we know that the triangle marked point is optimal, that is, $g(x) = f(x)$. By the No-Jumps Lemma (Lemma 3), $f()$ must lie on or below the dotted line of slope one from $g(x)$. Suppose $f(x+1)$ does lie on the line, then any extremal set $A(x+1)$ cannot have a loose vertex since otherwise $f(x)$ is not optimal by the same reasoning used in the proof of the No-Jumps Lemma. If $f(x+2)$ lies on this line, then because there are no jumps in $f()$, and because $g(x)$ is optimal, $f(x+1)$ must also lie on the line and neither $A(x+1)$ nor $A(x+2)$ can have loose vertices. Stating that A has no loose vertices is equivalent to saying $A = \Lambda\nabla A$, which is the premise of our next lemma.

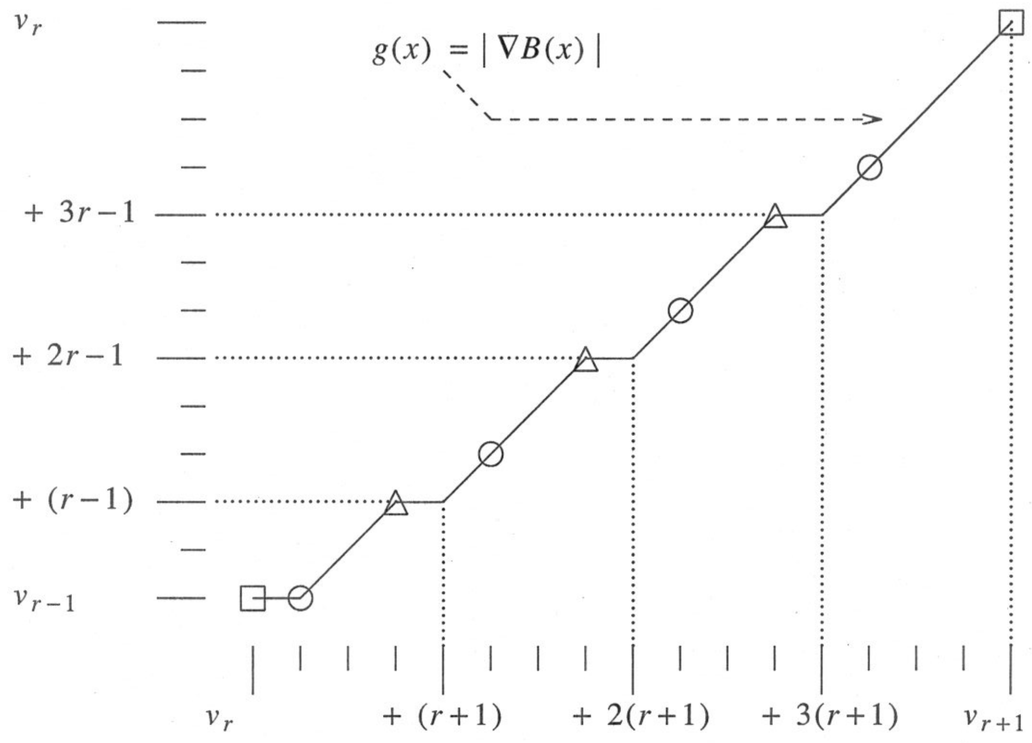


figure 2.14.

The graph of the function $g(x) = | \nabla B(x) |$ for $v_r \le x \le v_{r+1}$ with $r > 0$. Squares mark values that are known (by the previous lemma) to be the best possible. If a circled point is shown to be an extremal value, then every point on $g(x)$ from that circled point to the next point marked with a triangle is also an extremal value.

Lemma 5: If A is any subset of vertices in Z^d with

$$|A| = v_r + j(r+1) + 1 \text{ , and}$$

$$A = \Lambda \nabla A \text{ , then}$$

$$| \nabla A | < v_{r-1} + jr + 1 \ .$$

The proof is very short, but we will use a result from [28], so we will delay the proof of this lemma until after introducing some of their definitions and the required result. For our purposes, the main point behind these definitions is to define a continuous approximation to a discrete standard sphere.

First, one needs a generalization of an indicator function for points in a subset: if a universal set is of size z then a subset can be identified with a z-component vector with each component equal to 0 or 1 depending on whether the associated element is in the subset or not. This vector can also be identified with a $\{0, 1\}$-valued function. This

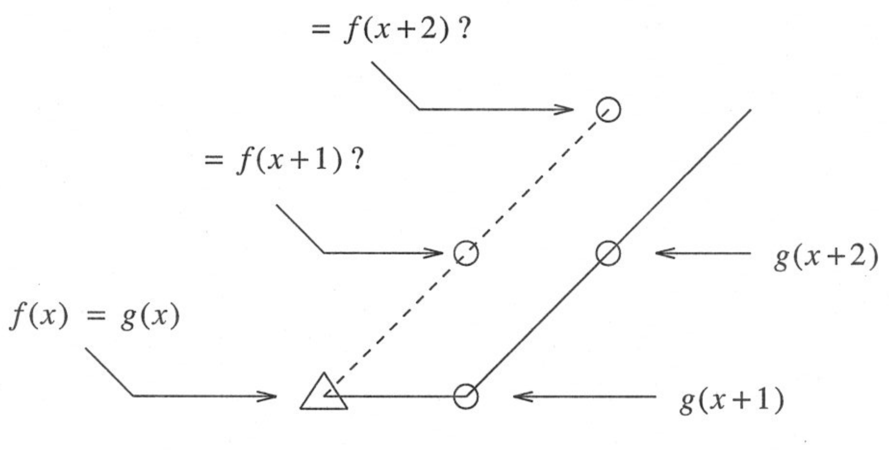


figure 2.15.

An enlarged portion of the graph of $g(x)$ at a knee.

function can be generalized by allowing points to map to values intermediate between 0 and 1.

Definition: [27] A *Fractional System f* is a function from points in a set S to the closed interval of the real numbers [0, 1]:

$$f : S \rightarrow [0,\ 1] \ .$$

For our purposes, we can think of S as either Z^d or Z_k^d. If f identifies a subset A, that is, each element of the domain maps to one if the element is in the subset and otherwise maps to zero, then we can ignore the difference between f and the subset itself and write A for f interchangeably. Thus, if we write A where a fractional system is expected, there should not be any confusion. Standard spheres are treated differently since one wants to smooth out the ''rough outside.''

A *Fractional Hamming Ball* (or ball) $b(x)$ is a fractional system of the form

$$b(x) \quad = \quad \begin{cases} 1 & \text{if } d(x,\, \mathbf{0}) \ < \ r \\ \alpha & \text{if } d(x,\, \mathbf{0}) \ = \ r \\ 0 & \text{if } d(x,\, \mathbf{0}) \ > \ r \end{cases}$$

$$\text{where} \quad 0 \le r \le \frac{k}{2} \ , \qquad \alpha \in [0,\ 1] \ , \text{ and}$$

$d()$ is edge distance. A fractional Hamming ball smooths out the partially filled outer shell of any standard sphere and distributes it over the entire outer shell. That is, suppose one has the standard sphere $B(v_{r-1} + x)$ for some x in the interval $[1,\ 4r]$. Then there is a fractional Hamming ball which assigns each element in $\nabla B(v_{r-1} + x)$ the value 1, assigns each element in $\partial_{ex}\nabla B()$ the value $x/4r$, and assigns all other points the value 0. In fact, given any set A of size a, there is a standard sphere of size a, and also a corresponding ball.

The *Weight* $w(f)$ or $|f|$ of a fractional system f is the sum of f over its domain:

$$|f| \equiv w(f) = \sum_{x \in \text{domain}(f)} f(x)$$

For a fractional system that defines a subset A, its weight is simply $|A|$. A ball of weight $a = |A|$, which we will write b^A or b^a, essentially converts A into a ''smoothed'' standard sphere with the same total weight as the number of the vertices in A. The advantage of balls is that the measurement of neighborhoods is simplified. That is, define any vertex with non-zero value to be inside ball b, and all others outside. Then the neighborhood of b is Λb with each vertex in b getting the value one, vertices

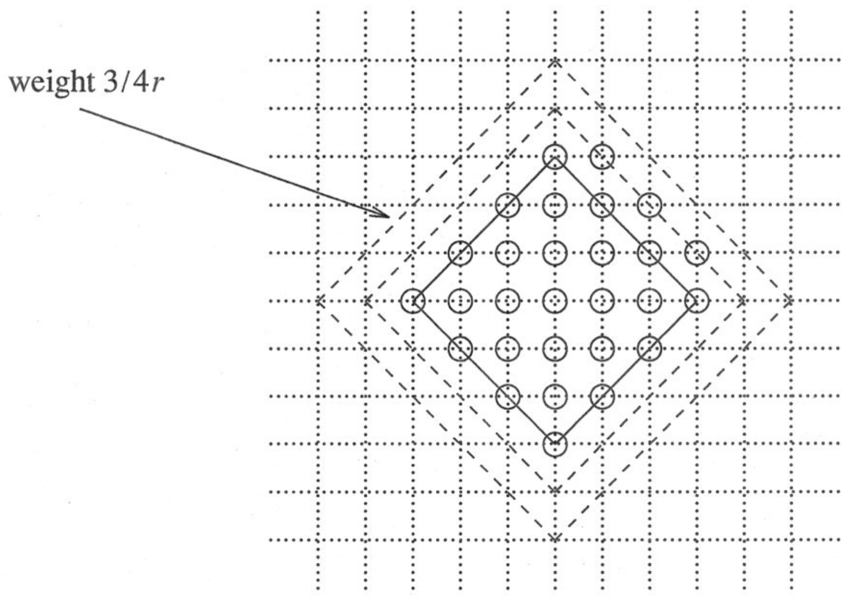


figure 2.16.

The neighborhood of a ball. The circled vertices represent the standard sphere $B(v_3 + 3)$. The corresponding ball is b^{v_3+3} and includes all the vertices on or inside the inner dashed diamond. For b^{v_3+3}, all the vertices inside or on the solid diamond have weight 1, and all vertices on the inner dashed diamond have weight $\frac{3}{4r}$, where $r = 4$. The neighborhood of the ball Λb^{v_3+3} includes all vertices inside or on the outer dashed diamond. Every vertex in Λb^{v_3+3} has weight 1 except those on the outer dashed diamond which get the weights previously assigned to vertices on the ball's exterior: $3/4r$.

in $\partial_{ex} b$ getting the value α, and all other vertices remain at value zero. We then will augment the notation for Λ to include this new meaning of neighborhood for fractional systems.

The *Neighborhood of a Fractional System f* is the fractional system

$$\Lambda f(x) = \begin{cases} 1 & \text{if } f(x) \neq 0 \\ \max\,\{f(y) : d(x, y) = 1\} & \text{if } f(x) = 0 \end{cases}$$

For a subset A, this definition is identified with the previous meaning of ΛA: the vertices in ΛA all have value one. Thus, the weight of ΛA is $|\Lambda A|$. For a ball the result of taking its neighborhood is that the outer shell vertices get value one, and the values that were on the outer shell get assigned to the vertices in the next larger shell. So, for the ball $b^{v_{r-1} + x}$ corresponding to the standard sphere $B(v_{r-1} + x)$, the vertices in $B(v_r)$ get the value one, while those in $Shell_{r+1}$ get the value $x/4r$. See figure 2.16.

□

We now can state Theorem 4 of Bollobás and Leader.

Theorem 6 [28]: Let $k \geq 2$ be even, and let f be a fractional system on Z_k^d of weight v. Then $w(\Lambda f) \geq w(\Lambda b^v)$.

In terms of our present problem this can be rewritten as follows.

If A is a subset of Z_k^d with interior ∇A, then

$$|\Lambda \nabla A| \geq |\Lambda b^{\nabla A}| .$$

As was mentioned earlier, for our purposes, the set A will always be small enough that the graph can be either Z^d or Z_k^d.

□

Now that these preliminaries are out of the way, we can give the proof of Lemma 5. Essentially, Theorem 6 does all the work for us.

Proof: (Lemma 5) The proof will be by contradiction. That is, suppose $A = \Lambda \nabla A$,

$$|A| = v_r + j(r+1) + 1 \text{ , and}$$

$$|\nabla A| \geq v_{r-1} + jr + 1 .$$

Consider a ball of weight $|\nabla A|$. The vertices in the outer shell of $b^{\nabla A}$ have weight $(jr+1)/4r$, which is also the weight of the vertices in the outer shell of $\Lambda b^{\nabla A}$. Since there are $4(r+1)$ of these vertices in $Shell_{(r+1)}$, we have the weight of $\Lambda b^{\nabla A}$ as,

$$|b^{\nabla A}| = v_r + 4(r+1)\frac{jr+1}{4r}$$

$$= \quad v_r + j(r+1) + 1 + \frac{1}{r} .$$

However, this contradicts Theorem 6 since

$$\begin{aligned} |\Lambda\nabla A| &= |A| \\ &= v_r + j(r+1) + 1 \\ &< |b^{\nabla A}| . \end{aligned}$$

□

Now that Lemma 5 has been established, it follows from the induction we described earlier that $g(x) = f(x)$ at all x except $x = v_r + j(r+1)$.

The last lemma needed for Theorem 5 tells us that the standard spheres at $x = v_r + j(r+1)$ are extremal. To prove this, we will use the sequence of ∂_{ex}-extremal sets described by Wang and Wang [29] to show that any set with an interior larger than $\nabla B(x)$ must contain more than x points.

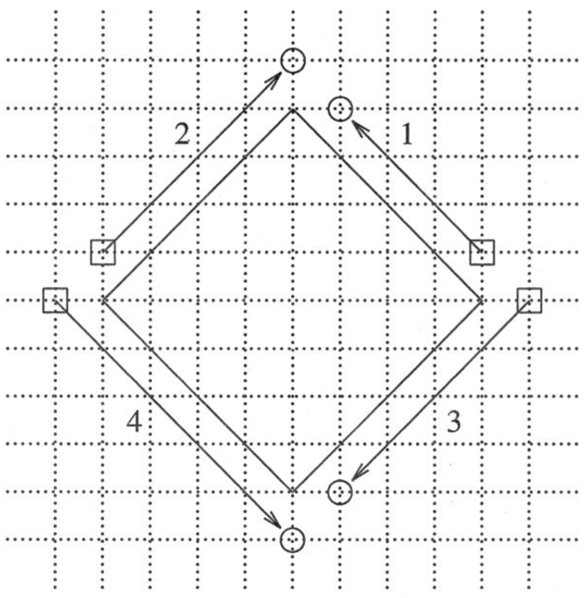


figure 2.17.

The order of choosing points from $Shell_{r+1}$ *for the* ∂_{ex}*-extremal sets* $W(x)$*. The inner diamond represents* $B(v_r)$*, which is, in the case shown above,* $B(v_4)$*. The outer shell (* $Shell_5$ *) is filled by travelling along each of the arrowed lines, marked 1 through 4, in turn, starting at a square-marked vertex and ending at a circled vertex. The arrowed line 1 covers r vertices, lines 2 and 3 cover* $r+1$ *vertices each, and line 4 covers* $r+2$*.*

In two-dimensions the Wang and Wang sequence of ∂_{ex}-extremal sets, $\{ W(x) \}$, is described by the following. Let each $q = (q_1, q_2) \in Z^2$ be associated with the 4-tuple (i_2, i_1, q_1, q_2), where $i_j = 1$ when q_j is strictly positive and equals zero otherwise. Write x as $x = v_r + z$ for some r and $0 \leq z < 4(r+1)$. Then $W(x)$ consists of all the points in $B(v_r)$ together with the z points in $Shell_{r+1}$ whose associated 4-tuples are the first z tuples in lexicographic order of the tuples associated with $Shell_{r+1}$. See figure 2.17.

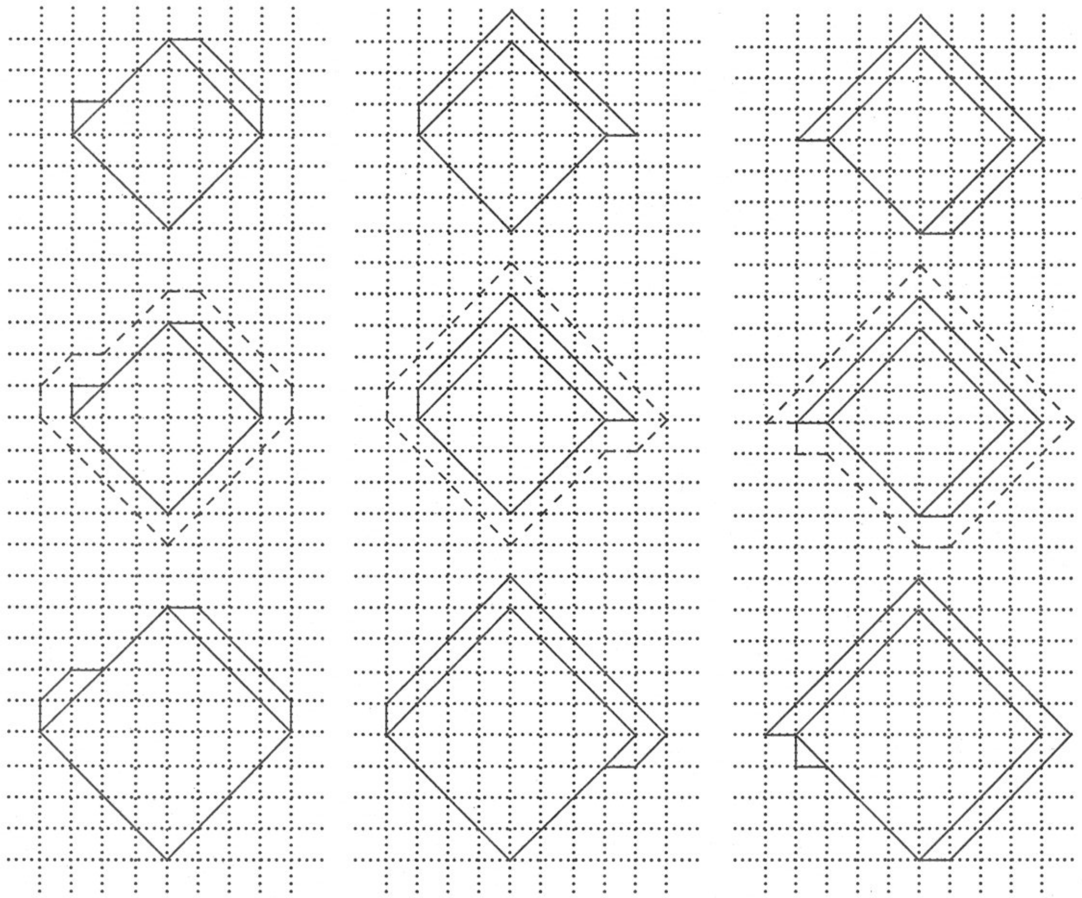


figure 2.18.

The neighborhoods of the ∂_{ex}-extremal sets $W(x)$ at $x = v_{r-1} + jr$. The top figure in each of the columns (a), (b), and (c), represents $W(x)$ for $j = 1, 2, 3$, respectively. The bottom figures represent $\Lambda W(x)$. For $r > 1$, it is easy to see that $\Lambda W(x)$ is $W(v_r + j(r+1))$ with one additional vertex. The case $r = 1$ can be checked easily and gives the same result.

Because $|\nabla B(v_r + j(r+1))| = v_{r-1} + jr - 1$, we are interested in the minimum size of a set whose interior is one larger than this value. In figure 2.18, the minimum size set is shown as the neighborhood of the ∂_{ex}-extremal set $W(v_{r-1} + jr)$. It is simple to check that $\Lambda W(v_{r-1} + jr) = v_r + j(r+1) + 1$ for $j = 1, 2, 3$ and $r \geq 1$. This is all that is necessary to establish the lemma.

Lemma 6: the standard sphere $B(v_r + j(r+1))$ is ∂_{in}-extremal for $1 \leq j \leq 3$.

Proof: Suppose, in contradiction, that $B(x)$ is not ∂_{in}-extremal, where $x = v_r + j(r+1))$. Let $S(x)$ be a ∂_{in}-extremal set of size x. Because of the No-Jumps Lemma (Lemma 3), we must have,

$$\begin{aligned} |\nabla S(v_r + j(r+1))| &= |\nabla B(\, v_r + j(r+1) - 1\,)| + 1 \\ &= (\, v_{r-1} + jr - 1\,) + 1 \; . \end{aligned}$$

Because the sets in the sequence $\{ W(y) \}$ are ∂_{ex}-extremal, we have,

$$|\Lambda W(\, v_{r-1} + jr\,)| \leq v_r + j(r+1) \; ,$$

since

$$\begin{aligned} |\Lambda \nabla S(v_r + j(r+1))| &= |S(v_r + j(r+1))| \\ &= v_r + j(r+1) \; . \end{aligned}$$

However, in the previous paragraph we showed that,

$$|\Lambda W(\, v_{r-1} + jr\,)| = v_r + j(r+1) + 1 \; , \text{and}$$

therefore, the set $S(x)$ cannot exist.

□

The condition on the $\nabla()$ function given by Theorem 4 is easily verified for the layer graph Z^2.

Theorem 7: If $\nabla(x) = |\nabla B(x)|$ for the standard spheres in Z^2, and given non-negative integers a, b and c, and given that,

$$b \geq \nabla(c) \quad \text{and} \quad a - b \geq c - \nabla(c) \; ,$$

then

$$\nabla(a) - \nabla(b) \geq \nabla(c) - \nabla^2(c) \; .$$

Proof: We will prove a slightly simpler form of the hypothesis which replaces the inequality in the first line with equality:

$$a - b = c - \nabla(c) \quad \Rightarrow \quad \nabla(a) - \nabla(b) \geq \nabla(c) - \nabla^2(c) \; . \quad \text{(i)}$$

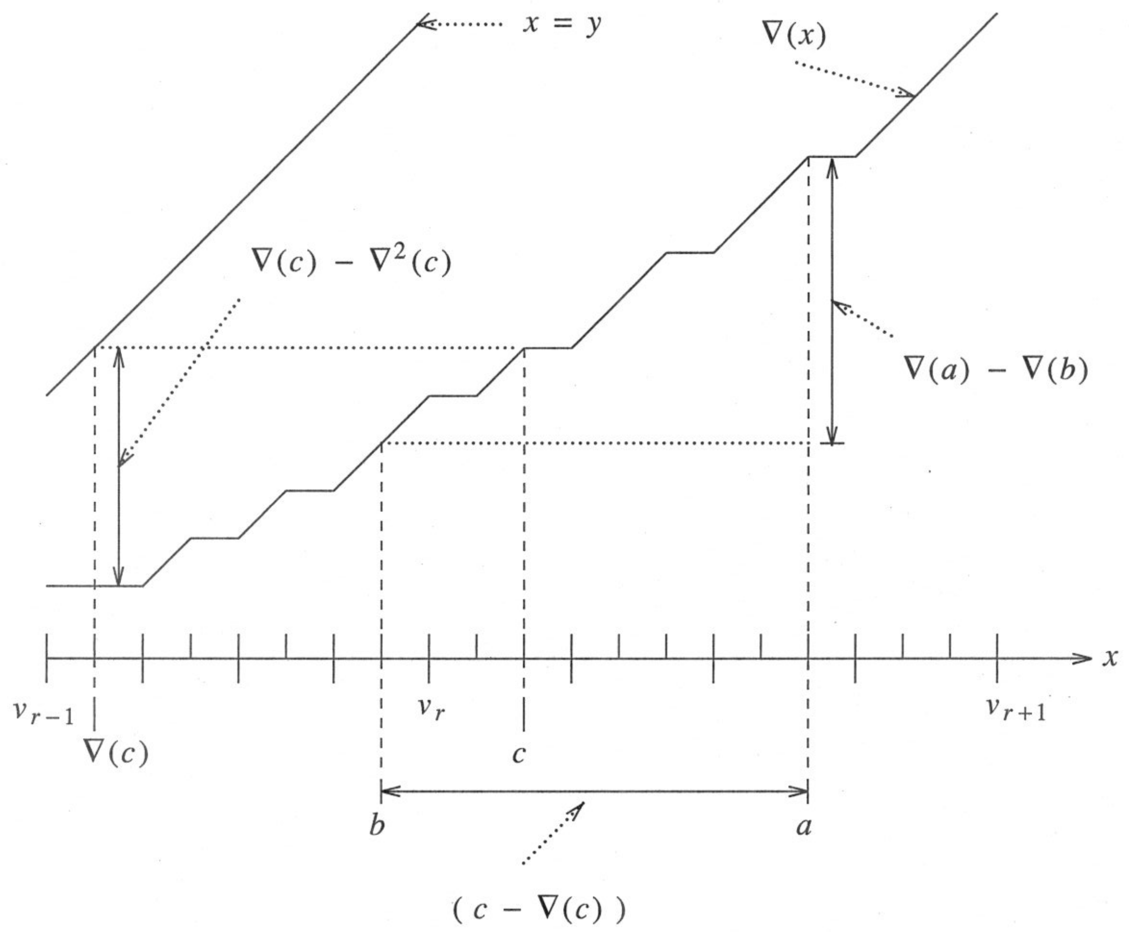


figure 2.19.

The graph of $\nabla(x)$ *from* $x = v_{r-1}$ *to* v_{r+1} *(here* $v_r = v_2$*). With the distance between a and b held constant at* $(c - \nabla(c))$ *and starting with* $b = \nabla(c)$*, proposition (i) says that sliding b to the right never causes the distance* $(\nabla(a) - \nabla(b))$ *to become less than* $(\nabla(c) - \nabla^2(c))$*.*

It will be obvious from the method of proof that making a larger than $b + (c - \nabla(c))$ can only strengthen the argument, and the proposition therefore follows in its more general form. Proposition (i) holds trivially when $\nabla(c) = 0$ because of the monotonicity of $\nabla()$, so we assume in what follows that $c \geq 5$ (the least x for which $\nabla(x) > 0$).

Figure 2.19 illustrates the proposition in a general context. The question is then, if b starts at $\nabla(c)$ and slides to the right while a maintains a distance of $c - \nabla(c)$ from b, does the distance marked $\nabla(a) - \nabla(b)$ ever decrease to less than its starting value of $\nabla(c) - \nabla^2(c)$?

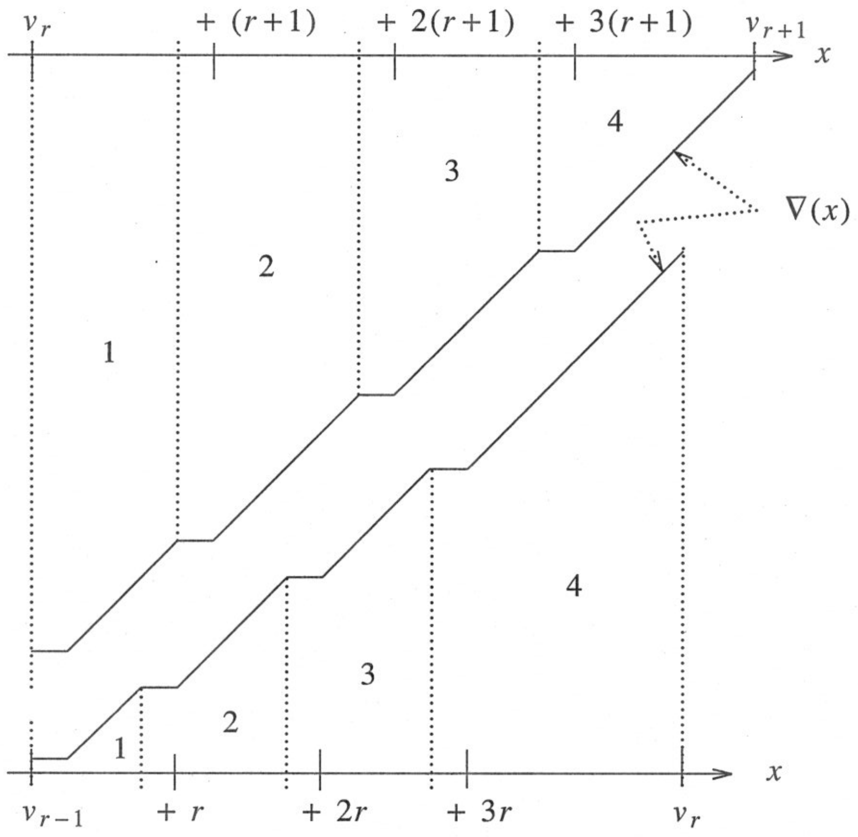


figure 2.20.

The phases of two adjacent periods of $\nabla(x)$. *The lower curve can be thought of as a copy of the upper curve with a shifted origin. In this case the lower curve would have its origin shifted by* $(v_r - v_{r-1}) = 4r$, *that is, the lower curve is the graph of* $\nabla(x - (v_r - v_{r-1}))$. *Phase j of period r of the curve* $\nabla(x)$ *includes all the points from* $\nabla(v_r + \phi(r, j))$ *up to but not including* $\nabla(v_r + \phi(r, j+1))$.

Let us refer to the portion of the x axis from v_r to v_{r+1} as the r^{th} *Period* of $\nabla(x)$, these are the values associated with filling $Shell_{r+1}$. Within each period the graph of $\nabla(x)$ can be broken into four *Phases* at $v_r + \phi(r, j)$ for $j = 0, 1, 2, 3$, where

$$\phi(r, j) \quad = \quad \begin{cases} 0\,, & \text{for } j = 0 \\ j(r+1) - 1\,, & \text{for } 0 < j \leq 3 \end{cases} .$$

The i^{th} phase of the r^{th} period of $\nabla(x)$ is the half-open interval $[\, v_r + \phi(r, i),\, v_r + \phi(r, i+1)\,)$. Figure 2.20 shows the general structure of the graph of $\nabla(x)$ for two adjacent periods with the phases numbered and indicated by dotted lines.

We can produce a graph similar to figure 2.20 by graphing $g_1 = \nabla(x)$ and a shifted version of this curve, $g_2 = \nabla(x - (c - \nabla(c)))$, above the same axis. In fact, figure 2.20 is a representation of such a graph of these two curves when $c = v_r$ and $\nabla(c) = v_{r-1}$. The two curves produced in this way show the difference $(\nabla(c) - \nabla^2(c))$ as the vertical distance between the two curves at $x = c$. Also, the vertical distance between the two curves at $x = c + y$ is equal to $(\nabla(a) - \nabla(b))$ when $a = c + y$, that is, when b has moved y to right of its starting point at $\nabla(c)$.

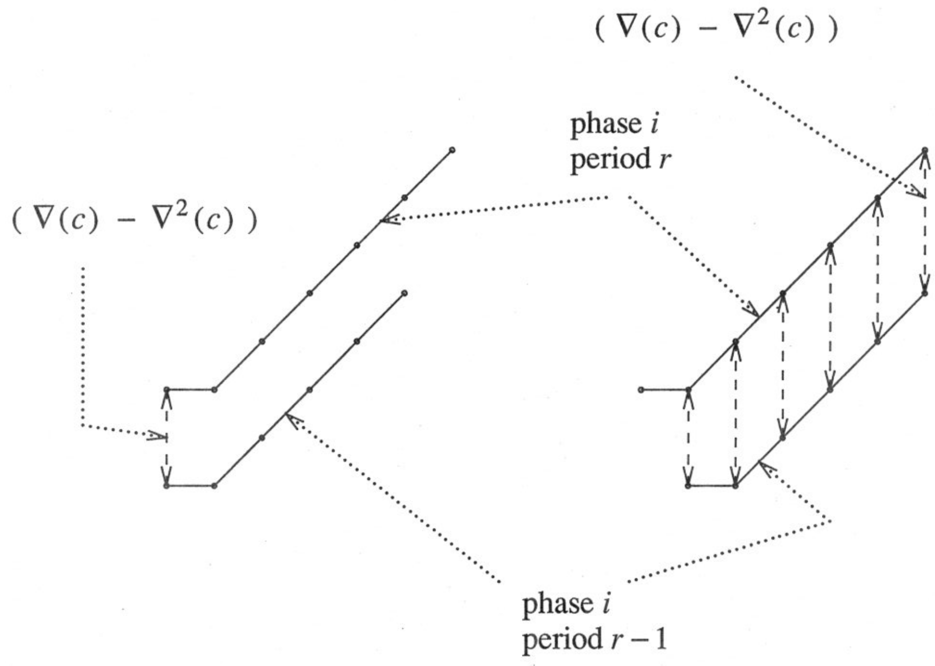


figure 2.21.

The two possible alignments of the curves $\nabla(x)$ and $\nabla(x - (c - \nabla(c)))$ at phase i when c is assumed to lie somewhere in phase i of period r. The dashed, arrowed lines represent the vertical distance between the curves at $x = c$ for all possible choices of c in phase i.

It is easy to verify that if c is in phase i of period r, then $\nabla(c)$ is in phase i of period $(r-1)$. Figure 2.21 shows the two possible orientations of the two curves at phase i. The left portion of that figure shows the orientation that results when c is at the beginning of phase i. If c is not at the beginning of the phase, then the orientation is as shown in the right portion of that figure. The dashed lines show the distances representing $(\nabla(c) - \nabla^2(c))$ for the various possible values of c.

It is clear that, given any starting position at $x = c$, the vertical distance between the curves at x either remains the same or increases by one as x moves right to the end of phase i in period r. Now, the distance between the curves, as x moves right beyond the end of phase i, remains the same or increases by one whenever x coincides with a horizontal segment in the lower curve, that is, at the initial segment of a phase. On the other hand, the distance remains the same or decreases by one when x coincides with the initial segment of a phase of the upper curve. In any case other than these two, the distance remains the same. Because the horizontal span of phase j in period $t-1$ is one less than the horizontal span of phase j in period t, it follows that, as x moves to the right from the end of phase i in period r, that m initial segments in the lower curve will always be reached at least as soon as m initial segments in the upper curve for any $m > 0$. Consequently, $\nabla(a) - \nabla(b) \geq \nabla(c) - \nabla^2(c)$.

□

2.3.3. Standard Spheres for the Triangular Lattice

Part of the general program presented previously for the layer graph Z^2 can be carried out in a somewhat simpler way for other layer graphs. Using the same language as was used to define the standard spheres in Z^2, we define shells and standard spheres in the triangular lattice. In this case, as in the case for Z^2, the standard spheres with ''full'' shells are defined by $B(v_r) = \{x \mid d(x, 0) \leq r\}$, where $v_r = |\{x \mid d(x, 0) \leq r\}|$. The principal work done in this section will be to make use of the ''Wulff Construction'' for continuous extremal sets to show that the standard spheres $B(v_r)$ for the triangular lattice are ∂_{in}-extremal. Although, as we will demonstrate below, the specific standard spheres $B(x)$ are extremal when $x = v_r$, we will not prove this for general values of x, that is, we will not show here that the standard spheres $\{B(x)\}$ form an extremal sequence. However, is should be clear that the approach used for Z^2 could, with the necessary attention given to details, be applied here. Technically, the demonstration of the existence of a nested, extremal sequence is required in order to apply Theorems 3 and 4 and thereby make pebbling set size estimates and apply the I/O cost analysis to throughput bounds. However, since it is intuitively apparent that, if the standard spheres $B(v_r)$ are extremal it is only a matter of details to establish the intermediate spheres are also, we will assume that the our standard spheres do form an extremal sequence for the triangular lattice when we come to analyzing the performance of LGM-1.

The Wulff Crystal

The Wulff construction in its general form is presented in [31] (see [32] for an introduction to geometric measure theory). We will only paraphrase it here in sufficient detail to allow us to handle the present problem. We will also present in abbreviated form the Wulff theorem which addresses the optimality of the result of a Wulff construction, a Wulff crystal. The Wulff theorem states that the result of the Wulff construction for a given function *f*, is a set of a given ''mass'' (its size by some measure) that is minimized for the integral of *f* over its surface. For instance, in three dimensions, when the integrand is 1 identically and the mass is the usual volume, the integral is simply the surface area of the set, and the Wulff crystal gives the set that solves the isoperimetric problem of least surface area for a fixed volume, a sphere.

In two dimensions, the ∂_{in}-extremal sets are the discrete analog of least-perimeter continuous sets. In the discrete case, the perimeter of a set S is measured by the number of lattice points in $\partial_{in} S$, and the mass is the number of points in ∇S. In two-dimensions, we can make the connection between the Wulff crystals and our discrete sets by assigning a tile of unit area to each point in ∇S, and connecting the points in $\partial_{in} S$ with a piecewise linear closed curve C. The value of *f* at a point on C at which the tangent to C has a given orientation, will be the mean density of lattice points per unit distance along a line of that orientation. That is, taking a line of the given orientation, count the least number of vertices per unit length along the line whose removal partitions the lattice into two pieces so that one piece lies on one side of the line and the other piece lies on the other side of the line. Integrating *f* around C gives the number of lattice points in $\partial_{in} S$. The area inside the curve C will approximate the number of lattice points in ∇S. The Wulff Crystal for *f* will give the shape of a ∂_{in}-extremal set in the triangular lattice.

Wulff's Construction and Theorem in their general forms are stated for *d*-dimensional real space and for more general measures; however, we will only state the construction and theorem for the plane R^2, using the usual measures on the plane and restricting the the range of subsets of the plane to which the theorem applies. Further, we find it more convenient to state the integrand *f* as a function of the unit vectors and integrate *f* of the outward normal to C.

Wulff's Construction [abbreviated]: Let *f* be a function from the unit vectors in R^2 to the reals. For each unit vector *x*, form the vector $xf(x)$. At the point $xf(x)$, construct the line normal to *x*: $l(a) = ay + xf(x)$ where $x \cdot y = 0$ (that is, ''$\cdot$'' is the inner product, so *x* and *y* are orthogonal). For each such line *l*, discard the open half space not

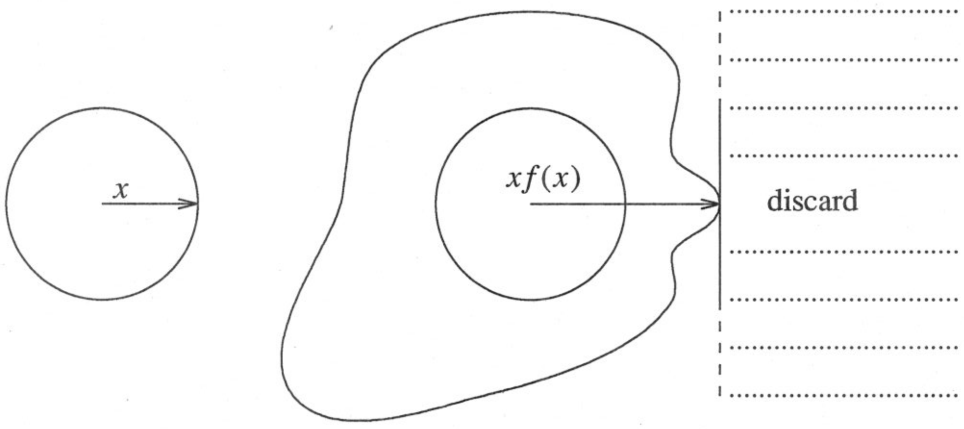


figure 2.22.

The Wulff Construction. The unit vector x (x is shown on left lying on the unit circle), are scaled by $f(x)$ to get the vector $xf(x)$ (the curve shown on right beyond the unit circle is the scaled transformation of the unit circle). The half-space defined by the line normal to $xf(x)$ is discarded for every x (the half-plane indicated by the dotted lines is discarded). The remaining set is the Wulff Crystal for f.

containing the origin. The union of the remaining half spaces defines the Wulff Crystal for f, W_f. See figure 2.22.

□

Wulff's Theorem is also abbreviated and stated only for R^2.

Wulff's Theorem [abbreviated]: Let S be any region of R^2 bounded by a simple closed piecewise C^1 curve ∂S whose outward unit normal to the tangent of ∂S exists at almost all points s on ∂S and is $n(s)$. Define $\mathbf{F}(S)$ as,

$$\mathbf{F}(S) \quad = \quad \int_{\partial S} f(n(s)) \; ds \ .$$

Also, define the mass of S as $\mathbf{M}(x)$,

$$\mathbf{M}(S) \quad = \quad \int_S dxdy \ .$$

Then,

$$\mathbf{F}(W_f) \quad \leq \quad \mathbf{F}(S) \ ,$$

for every S such that

$$\mathbf{M}(W_f) \quad = \quad \mathbf{M}(S) \ .$$

□

Since the measures used in the integration are uniform on R^2, the theorem applies to any scaled version of W_f as well. The above theorem states that the Wulff crystal for f minimizes the "length" of the perimeter for a given area, where the length is generalized to mean the integral of any function which is only dependent on the orientation of the tangent to the boundary.

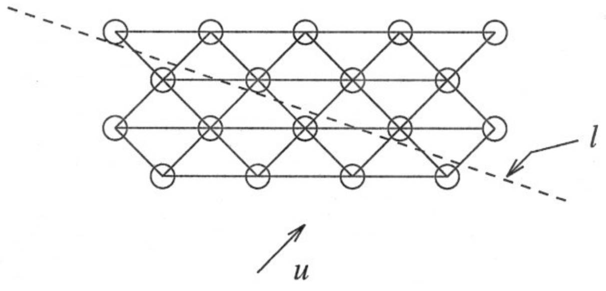


figure 2.23.

An arbitrarily oriented line in the triangular lattice with unit edge length d_0. Separating the lattice along l requires removing one vertex for each edge parallel to u that l intersects.

We want to find a function f that, when integrated around the curve C defined by the vertices in $\partial_{in} S$ for some set S, gives the number of vertices in $\partial_{in} S$. The function f in this case is found by counting, per unit distance, the number of lattice points required to separate the lattice in two along a line with normal x. In figure 2.23 an arbitrary line l is drawn through a portion of the infinite triangular lattice graph whose edge lengths are d_0. As can be seen in that figure, it is sufficient to measure the distance along l between lines parallel to the lattice-generating unit vector u passing through the lattice points. We place the separating line l so that it coincides with a lattice point to make the calculation more convenient (see figure 2.24). We then want to find the distance along l between the lattice point and the nearest line through a vertex parallel to u. If the line l lies at an angle θ to the angle bisector b in figure 2.24, then the distance is $d_0\sqrt{3}/(2\cos(\theta))$, and thus f for the normal to such a line l is equal to 1 over this quantity. Now that we know what the function f is, we can proceed to find its Wulff Crystal.

The upper diagram in figure 2.25 shows a unit vector normal to line l, $normal(l) = x_\theta$. (There are two such normals for each line l. We assume the following arguments apply to both.) The lower left diagram in figure 2.25 shows the result of scaling x_θ by $f(x_\theta) = 2\cos(\theta)/(d_0\sqrt{3})$. A little trigonometry will show that for θ in the range $[0, \pi/6]$ the line normal to x_θ at $x_\theta f(x_\theta)$ intersects $normal(b)$ at the point

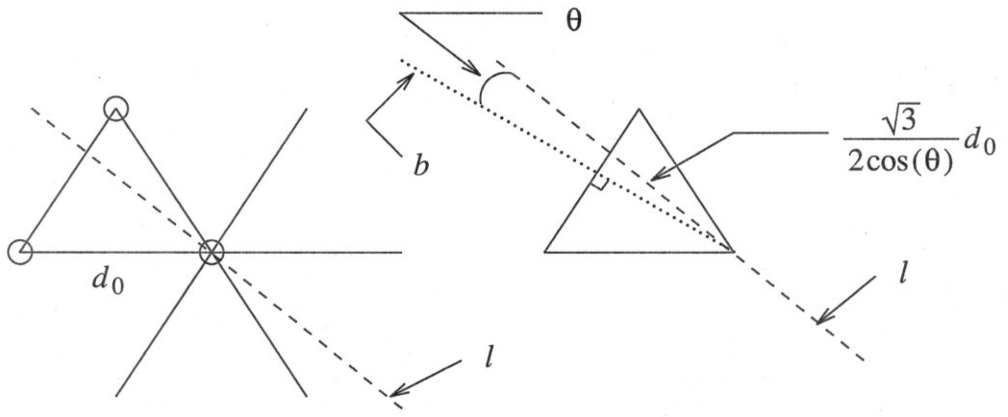


figure 2.24.

The distance along line l between lines parallel to u. The left diagram shows the line l as a dashed line passing through a vertex (circled). On the right, the angle bisector b makes an angle θ *with l. The distance, when the graph edges have length* d_0, *is* $d_0\sqrt{3}/(2\cos(\theta)$.

marked $a = x_0 f(x_0)$. Since in this range of values for θ, $f(x_\theta)$ is minimum at $\theta = \pi/6$ ($f(x_{\pi/6}) = 1/d_0$), the Wulff Crystal W_f is defined by the lines normal to the unit vectors $x_{\pi/6}$. The result is as shown in the lower right of figure 2.25.

Extremal Standard Spheres for the Triangular Lattice

Now that we have a Wulff Crystal for the function f defined in the previous section, we can use it to prove that the standard spheres $B(v_r) = \{x \mid d(x, 0) \le r\}$ are ∂_{in}-extremal. The method will be to compare the area of a Wulff Crystal with the area of a region with the same perimeter integral. The region will be the area inside a curve drawn through the vertices of the interior perimeter of a set of vertices. See figure 2.26. Suppose $B(v_r)$ is not ∂_{in}-extremal. That is, suppose there exists some set S which is the same size as $B(v_r)$, but has a larger interior. We first find a planar region defined by set S which we can compare with a Wulff Crystal.

Scaling the triangular lattice so that the plane is tiled by unit area hexagons centered at the lattice points (see figure 2.27, the unit edge length becomes $d_0 = \sqrt{2}3^{-1/4}$), construct a curve C by connecting the vertices in $\partial_{in}S$ in the following way. Suppose the interior of S, ∇S, forms a connected component. (If ∇S is disconnected, each connected component can be treated separately.) Consider the perimeter C_1 of the region defined by the union of the hexagons associated with vertices in ∇S. Clearly, this is a

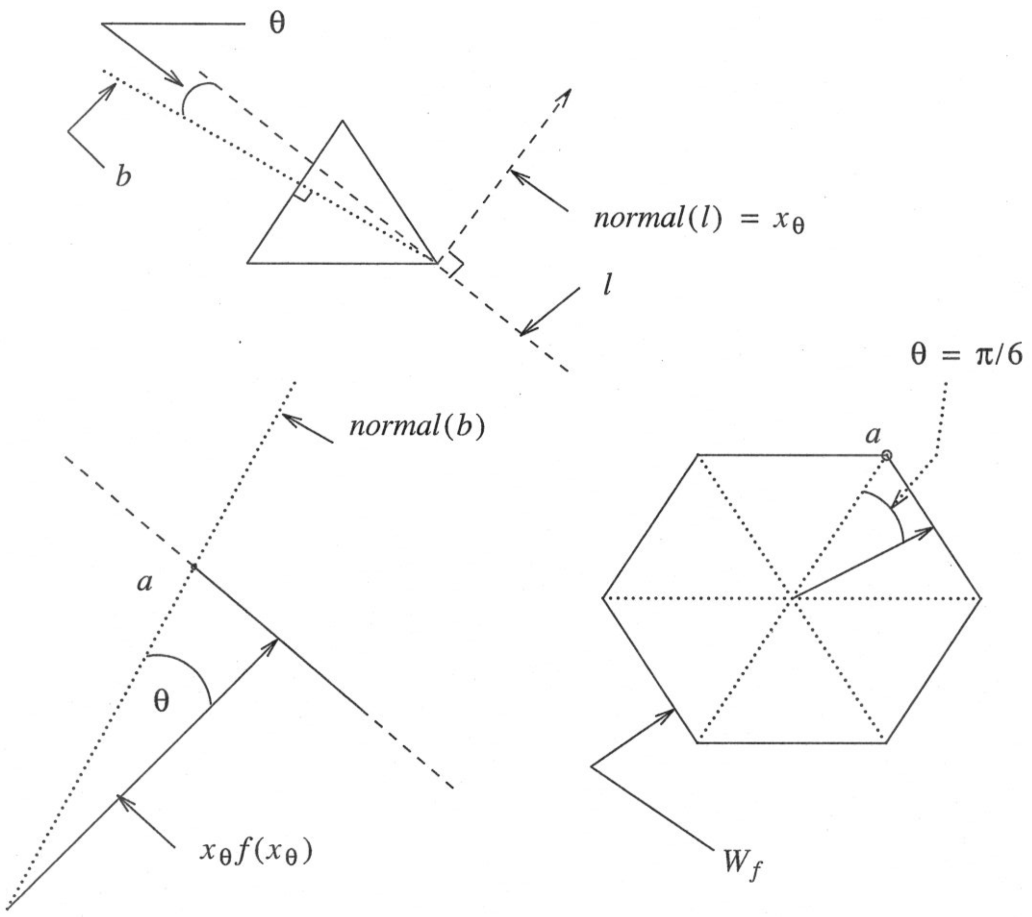


figure 2.25.

Constructing the Wulff Crystal for the function f defined above. The line l is pushed away from the origin along both its normals $\pm x_\theta$ *(only one is shown) for a distance* $f(x_\theta)$*. The top diagram shows the unit normal to l,* x_θ*. The lower left shows l pushed away from the origin. The point a is common to all such ''pushed'' lines for* θ *in the range* $[0, \pm\pi/6]$*. The resulting crystal is shown at lower right.*

simple closed curve. (If ∇S contains ''holes,'' then the function $|\nabla S(x)|$ for the extremal sets $S(x)$ of size x has jumps, violating the No-Jumps Lemma.) Let us call the edges with one end in ∇S and one end in $\partial_{in} S$ the *hair* of ∇S. Curve C_1 intersects every hair of ∇S at its midpoint. Produce curve C_2 from C_1 by deforming every section of C_1 lying between between adjacent hair intersections into straight line segments. Now form curve C by sliding every hair intersection point until it coincides with a vertex in $\partial_{in} S$. Curve C is a simple closed curve. Vertices adjacent along C are neighbors in the lattice graph because every line segment in C_2 either contracted to a point or is coincident with an edge. This last follows from the fact that adjacent hairs

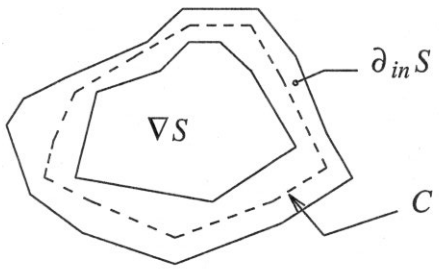


figure 2.26.

The curve C through the vertices of $\partial_{in} S$ defines a region whose are can be used to compare with the area of a Wulff Crystal whose perimeter integral is identical to that around C.

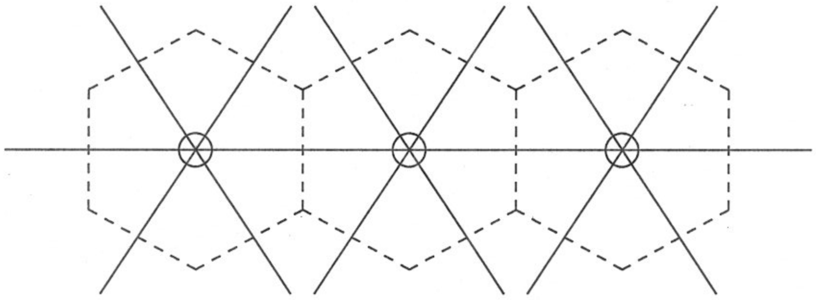


figure 2.27.

The tiling of the plane with hexagons. Edge lengths are scaled so that the area of each hexagon is 1.

share a common vertex, either in ∇S or in $\partial_{in} S$. In referring to the region of the plane bounded by the curve C, we will write $\hat{S}$.

We will use the following notation when referring to the areas associated with sets of vertices and regions in the plane. If A refers to a Wulff Crystal or the region $\hat{S}$, let

$$|| A || \equiv \mathbf{M}(A) .$$

If A is the interior of a set of vertices, say $A = \nabla S$, then let $|| \nabla S ||$ be the area of the union of the unit area hexagons associated with vertices in ∇S,

$$|| \nabla S || \equiv \sum_{x \in \nabla S} 1 .$$

Finally, if $A = \partial_{in} S$, let $|| \partial_{in} S ||$ be the area inside $\hat{S}$ associated with vertices in $\partial_{in} S$,

$$|| \partial_{in} S || \equiv || \hat{S} || - || \nabla S || .$$

The region $\hat{S}$ is a region of the plane to which Wulff's Theorem applies: we can compare its area with that of a Wulff Crystal for f whose perimeter integral is the same as the perimeter integral for $\hat{S}$. That is, for any j there is a scaled Wulff Crystal for f, $W(j)$, such that the integral of f around its perimeter is j: $j = \mathbf{F}(W(j))$. It follows from Wulff's Theorem and from the fact that the area of $W(j)$ is a strictly increasing function of j, that if, for some set N to which Wulff's Theorem applies, $\mathbf{F}(N) = j = \mathbf{F}(W(j))$, then $\| N \| \leq \| W(j) \|$. So, setting

$$j = \mathbf{F}(\hat{S}) ,$$

we must have

$$\| \hat{S} \| \leq \| W(j) \| . \tag{1}$$

We will show that this inequality is a contradiction, and thus $B(v_r)$ is extremal.

First, we can rewrite the left-hand side of (1) to get

$$\| \nabla S \| + \| \partial_{in} S \| \leq \| W(j) \| . \tag{2}$$

Now, we have, by supposition, that S and $B(v_r)$ are the same size,

$$| S | = | B(v_r) | = v_r ,$$

and that S has a larger interior than $B(v_r)$,

$$| \nabla S | = | \nabla B(v_r) | + u ,$$

for some positive integer u, and it also follows that

$$| \partial_{in} S | = | \partial_{in} B(v_r) | - u .$$

Because $| \nabla S | = \| \nabla S \|$, we can rewrite (2) by substituting for $\| \nabla S \|$,

$$\Big[| \nabla B(v_r) | + u \Big] + \| \partial_{in} S \| \leq \| W(j) \| ,$$

and, substituting for $| \nabla B(v_r) |$ and rearranging we have

$$\| \partial_{in} S \| \leq \| W(j) \| - v_{r-1} - u .$$

Recalling that r is the edge distance from the origin to $\partial_{in} B(v_r)$, it is easy to verify that $v_r = 3r^2 + 3r + 1$, and that $| \partial_{in} B(v_r) | = 6r$. Substituting for v_{r-1} in the previous inequality results in

$$\| \partial_{in} S \| \leq \| W(j) \| - (3r^2 - 3r + 1) - u . \tag{4}$$

We will show below that expressing the terms $\| \partial_{in} S \|$ and $\| W(j) \|$ in (4) in terms of r, u, and the number of loose vertices in $\partial_{in} S$, gives a contradiction.

We now want to to find an expression for $|| W(j) ||$. We first need to find the value of $j = \mathbf{F}(\hat{S})$. The function f was chosen so that, if we integrate f around C, the result gives us the number of non-loose vertices in $\partial_{in} S$. To see this, note that C follows edges in the graph, and the portion of C traversing the hexagon tile assigned to a specific vertex in $\partial_{in} S$ covers a distance of exactly one edge length d_0. Integrating f along this portion of C gives the value 1 because $f(n(x))$ at a point x where the tangent to C is parallel to an edge is $1/d_0$. That is, if $\partial_{in} S$ has q loose vertices, we have

$$j \quad = \quad \mathbf{F}(\hat{S}) \quad = \quad |\partial_{in} S| \ - \ q \ .$$

Substituting for $|\partial_{in} S$, and recalling that $|\partial_{in} B(v_r)| = 6r$, we have

$$j \quad = \quad 6r - u - q \quad = \quad 6r - t \ ,$$

where we set $t = u + q$. Now that we have j, we can proceed to evaluate $|| W(j) ||$ by finding an expression for the area of a hexagon in terms of its perimeter.

The integral of f around the perimeter of $W(j)$ is

$$\mathbf{F}(W(j)) \quad = \quad j \quad = \quad p_j \frac{1}{d_0} \ , \qquad (5)$$

where p_j is the length of the perimeter of $W(j)$. We can write the area of a hexagon as a function of its perimeter p as,

$$\text{Area}(p) \quad = \quad \frac{\sqrt{3}}{24} p^2 \ .$$

We then have,

$$|| W(j) || \quad = \quad \frac{\sqrt{3}}{24} p_j^2 \ .$$

Solving (5) for p_j, and substituting for p_j on the right-hand side of the above gives us,

$$|| W(j) || \quad = \quad \frac{1}{12} j^2 \ ,$$

and substituting for j and simplifying yields,

$$|| W(j) || \quad = \quad 3r^2 - rt + \frac{t^2}{12} \ . \qquad (6)$$

We next develop a lower bound for the quantity on the left-hand side of (4)

The left-hand side of (4) can be expressed as

$$|| \partial_{in} S || \quad = \quad \frac{1}{3} c_1 + \frac{2}{3} c_2 + \frac{5}{6} c_3 + \frac{1}{2} c_4 \ . \qquad (8)$$

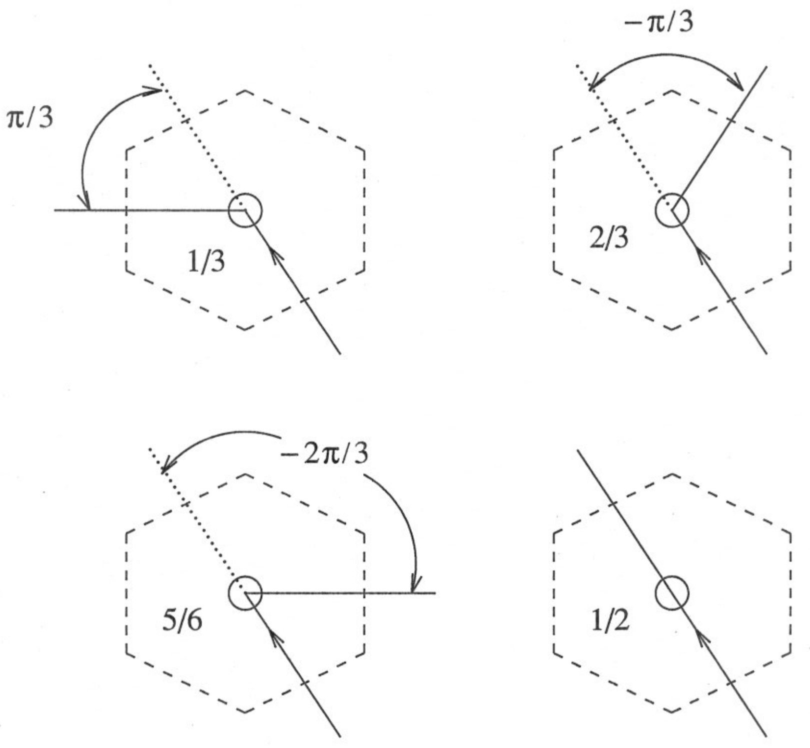


figure 3.28.

The four possible angles the curve C can make at a vertex. The solid line represents a portion of the oriented curve C, and the circle represents a vertex in $\partial_{in} S$. *Assuming the region* $\hat{S}$ *extends to the left and down, the area the vertex contributes to* $|| \hat{S} ||$ *is shown inside the unit hexagon. Note that an angle of* $2\pi/3$ *is impossible at a non-loose vertex.*

To see this, note that for any vertex v in $\partial_{in} S$, there are only four possible arrangements of the curve C through its associated hexagon tile. See figure 3.28. If we traverse C in a counterclockwise direction, the possible angles C can make at v are $\pi/3$, $-\pi/3$, $-2\pi/3$, and 0, and the area contributed to $|| \partial_{in} S ||$ by the associated hexagon in each case is respectively $1/3$, $2/3$, $5/6$, and $1/2$. Letting c_1 be the number of $\pi/3$ angle turns, c_2 the number of $-\pi/3$ turns, c_3 the number of $-2\pi/3$ turns, and c_4 the number of 0 angle turns in a complete counterclockwise traversal of C, and summing the areas for each vertex lying on C, gives (8).

Noting that the sum of the angles in a complete traversal of C must be 2π, we have

$$c_1(\frac{\pi}{3}) \ + \ c_2(-\frac{\pi}{3}) \ + \ c_3(-2\frac{\pi}{3}) \ + \ c_4(0) \ = \ 2\pi \ ,$$

solving for c_1 and simplifying gives,

$$c_1 = 6 + c_2 + 2c_3 . \tag{9}$$

Substituting (9) into (8) yields

$$||\partial_{in} S|| = 2 + c_2 + \frac{3}{2}c_3 + \frac{1}{2}c_4 . \tag{10}$$

A lower bound on this quantity requires a minimization of the right-hand side of (10). With the following two constraints, this becomes an integer linear programming problem. First, all the c_i are non-negative. Second, the sum of the c_i is the number of non-loose vertices in $\partial_{in} S$, that is

$$\sum c_i = j ,$$

which, when the right-hand side of (9) is substituted for c_1 gives

$$2c_2 + 3c_3 + c_4 = (j-6) .$$

The linear program is then

Minimize $(c_2, c_3, c_4,)\cdot(1, 3/2, 1/2)$ subject to

$$(c_2, c_3, c_4,)\cdot(2, 3, 1) = (j-6) \text{ , and}$$

$$c_i \geq 0 .$$

Since any feasible solution lying in the plane $c\cdot(2, 3, 1) = (j-6)$ is also optimal (the cost vector and the normal to the constraint plane are parallel), we get an optimal solution by setting $c_4 = j - 6$, $c_2 = c_3 = 0$. Substituting these values into (10) yields the desired lower bound,

$$||\partial_{in} S|| \geq \frac{j}{2} - 1 = \frac{(6r-t)}{2} - 1 . \tag{11}$$

We now have expressions for all the terms in (4).

We substitute (11) on the left-hand side of (4), and (6) on the right, and after combining terms, rearranging, and simplifying, we have

$$12r + 6 \leq t + q ,$$

and since $q \leq t$, we have

$$6r + 3 \leq t .$$

However, the number of non-loose vertices in the interior perimeter of S must be non-negative,

$$0 \leq j = 6r - t ,$$

so we have

$$t \leq 6r ,$$

and we have reached the desired contradiction. This establishes the following proposition.

Theorem 8: The standard spheres $B(v_r) = \{x \mid d(x, 0) \leq r\}$ are ∂_{in}-extremal in the triangular lattice.

2.4. Applying the Pebbling Bounds to Measure Architectural Optimality

The question that motivated the preceding development of pebbling arguments was whether or not the LGM-1 machine made reasonable use of its resources. That is, can the elements of LGM-1 be rearranged, and the computation steps reordered, so that a significant improvement in throughput is achieved. Here we will answer that question negatively by getting an upper bound on throughput for any machine with equivalent resources, and showing that the throughput of the LGM-1 architecture is within a small factor of this bound.

There are many ways to apply the pebbling arguments to an architecture. Applying them to a real machine, or a proposed architecture, requires that one define the resources of the machine in such a way that one clearly distinguishes between registers assigned to different parts of the machine. The pebbling game itself makes no distinction between memory locations, except that some are assigned one color and others are assigned other colors. In fact, there is no reason to associate any particular collection of registers together. The important element of assigning memory elements to classes is the definition of the communication capacity between the classes.

To define the memory locations for a given color, say color C-i, we enclose those elements in a collection of disjoint volumes (bounded by simple closed surfaces). Say V-i is this collection of volumes associated with color C-i. Whenever a datum node in the computation graph is pebbled with a C-i colored pebble, the meaning is that the given datum (or a copy) resides within some memory element inside this volume. If we wished, we could identify each pebble with a particular memory register, and it is tacitly assumed that this is done; however, there is no practical point in keeping track of the identities of the pebbles. We can identify some, or all, of the machine's memory registers outside of volume V-i with another color C-j by enclosing them in a similar volume V-j.

Communication between different parts of the machine is interpreted by the act of pebbling a node with a C-i colored pebble when it is not already pebbled by a C-i colored pebble. If the node was pebbled by a C-j colored pebble, then the interpretation is that the datum was communicated through whatever physical medium interconnects the two parts of the machine enclosed in V-i and V-j. This interconnection between parts of the machine is thought of as a communication channel. The channel capacity is measured by assuming the machine does nothing but transfer data through the channel. This allows one to separate the machine's processing capability from its communication capability. The capacity of the channel can be defined as the average sustainable rate that useful data can be transferred between the two portions of the machine inside the two volumes V-i and V-j, disregarding delays caused by the required data not being available.

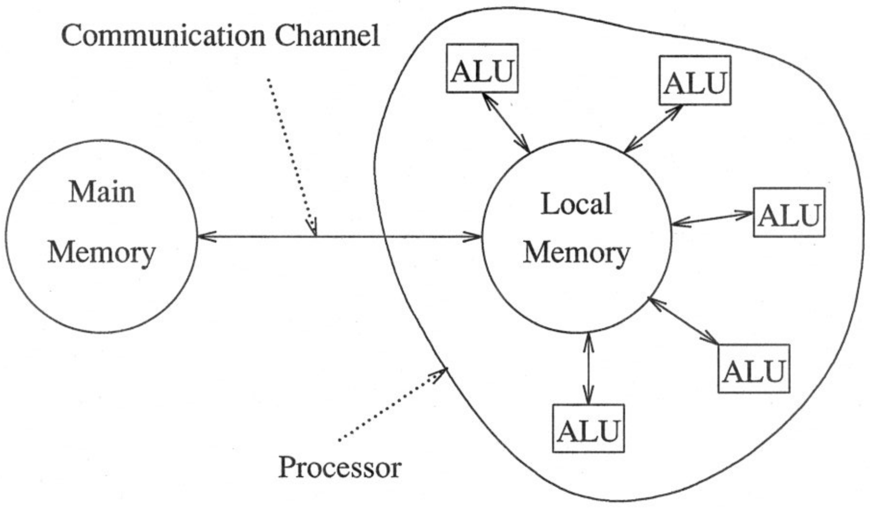


figure 2.29.

The machine model implemented by the standard parallel red-blue pebbling game. The diagram shows the processor as a collection of processor elements (ALU's) communicating through a shared memory. Thus, every local register is available to every local processing element (ALU). This allows infinite internal communication capability within the processor: any ALU that wants to use a datum stored in a local register (the corresponding node in the computation graph is red-pebbled) has direct access to that register. Results are written to local registers as well.

The pebbling bound allows us to bound the throughput of any machine with given resources, no matter how its resources may be arranged and used, so long as that arrangement allows a well defined division of the machine's registers into color classes corresponding to the assumptions the pebbling implies. The first bound we develop for machines that compute lattice-graph based computations restricts the possible rearrangement of resources into thoses that can be divided into two pieces: a ''main memory'' piece, and a ''processor'' piece with a well defined communication channel between them. The range of possible machines is then restricted to those that store the initial state of a lattice-graph based system in some storage medium (this medium could be tapes, disks, RAM chips, or whatever) and sends data to a processor which cannot contain the entire lattice state in its local registers. No updating of the lattice is done in the ''main memory.''

The first bound mentioned in the preceding paragraph restricts the class of architectures to one that is close to the general shared-memory PRAM model.. In the general model the processors can read and write to memory directly without requiring local storage of input or output. The two-piece pebbling forces the machine to have at least enough local registers (registers in the ''processor'' portion of the machine) so that the required inputs to a calculation step can be held locally. The processors may share data among themselves so that the local memory together with the processors form a local shared-memory PRAM. See figure 2.29. This division of resources is modelled by the standard parallel red-blue pebble game, where data held in blue registers is not updated.

The throughput bound for this model is derived from the I/O cost inequality from the beginning of section 2.3:

$$C_{I/O} \geq \left(\frac{z}{\beta} - 1 \right) k \, , \tag{1}$$

where z is the number of nodes in the computation graph (excluding input nodes), β is some bound on the maximum size of a dependency of a kernel dominator, and k is a free parameter defining the number of I/O steps per I/O-division of the pebbling game. The quantity represented by β is defined by

$$\beta \geq \max_{Peb(r)} \{ \max_{i} | \overline{D}(d_i) | \} \, , \tag{2}$$

where $Peb(r)$ is the collection of all pebblings of the computation graph which use at most r red pebbles, and where i ranges over all elements of a k-I/O-division of a pebbling. From Theorem 2 in section 2.2.2, we have d_i as

$$d_i = R_{\sigma_{i-1}} \cup (B_{\sigma_{i-1}} \cap \overline{P}_i^R) \, .$$

We can rewrite the right side of this as

$$= R_{\sigma_{i-1}} \cup \{ (B_{\sigma_{i-1}} \cap \overline{P}_i^R) - R_{\sigma_{i-1}} \} \, . \tag{3}$$

The size of first term on the right is bounded by

$$|R_{\sigma_{i-1}}| \quad \leq \quad r \, ,$$

since r is the maximum number of red pebbles allowed. The second term on the right of (3) represents the number of nodes that were read into the local memory during the i^{th} sub-pebbling (these nodes were blue-pebbled but not red-pebbled at the beginning of the sub-pebbling, and had red pebbles placed on them during the sub-pebbling). Since the total I/O for the sub-pebbling is at most k, we have

$$|d_i| \quad \leq \quad r \; + \; k \; .$$

So, we can write (2) as

$$\beta \quad \geq \quad \max |\overline{D}(M)| \; ,$$

where the maximum is taken over all sets M of size $(r+k)$. This implies that we can set β to

$$\beta \quad \equiv \quad |\overline{D}(M)| \; ,$$

where M is an (ISO-1)-extremal set M of size $(r+k)$.

The throughput bound comes from noticing that the communication capacity of the channel between the red registers and the blue registers must be sufficient to accommodate the communication in the time allowed by the computation. Letting this time be t, we have

$$V_{com} \, t \quad \geq \quad C_{I/O} \; ,$$

where V_{com} is the communication capacity (bandwidth) of the channel between the red and blue registers. Substituting the right side of (1) on the right we have

$$V_{com} \, t \quad \geq \quad (\frac{z}{\beta} - 1) \, k \; .$$

Setting $\lambda = 1 - \beta / z$, and dividing by t this becomes

$$V_{com} \quad \geq \quad \frac{\lambda (z/t) k}{\beta} \; ,$$

and rearranging gives

$$\frac{z}{t} \quad \leq \quad V_{com} \frac{\beta}{\lambda k} \; .$$

The left side of this inequality is the average effective computational speed of the machine. We take the right side as the definition of the bounding throughput V,

$$V \quad \equiv \quad V_{com} \frac{\beta}{\lambda k} \; .$$

This expression holds for any lattice-graph based computation.

We now want to apply this limit on throughput to a lattice graph computation whose layer graph is the two-dimensional toroidal grid. We know from section 2.3.2 that the (IS0-1)-extremal sets of size x, for a lattice-graph computation with layer graph G, are the standard balls in G containing x points, $B(x)$. Suppose

$$\begin{aligned}(r+k) &= v_n \\ &= |\{x : d(x,0) \leq n\}| \\ &= |B(v_n)| \,,\end{aligned} \tag{4}$$

for standard balls in the toroidal grid. Since $v_n = v_{n-1} + 4n$, and $v_0 = 1$, we have

$$\begin{aligned}v_n &= 1 + \sum_{i=1}^{n} 4i \\ &= 2n^2 + 2n + 1 \,.\end{aligned} \tag{5}$$

As was shown previously, the dependency of the standard balls consists of a series of standard balls of decreasing radii. The size of the dependency of the standard ball $B(v_n)$ is then

$$\begin{aligned}|\overline{D}(B(v_n))| &= \sum_{i=0}^{n-1} |B(v_i)| \\ &= \sum_{i=0}^{n-1} v_i \\ &= \sum_{i=0}^{n-1} (2i^2 + 2i + 1) \\ &= \frac{2}{3}n^3 + \frac{n}{3} \,.\end{aligned} \tag{6}$$

Substituting $(r+k)$ for v_n in (5), solving for n and taking n positive we get

$$n = \frac{(2(r+k) - 1)^{1/2} - 1}{2} \,.$$

Substituting for n and v_n in (6) we get an expression for β as,

$$\begin{aligned}\beta &\equiv |\overline{D}(B(r+k))| \\ &= \frac{(2(r+k) - 1)^{3/2} + 5(2(r+k) - 1)^{1/2} - 6(r+k)}{12} \,.\end{aligned}$$

This expression was derived by assuming $(r+k)$ was exactly the size of a standard ball.

It turns out, though, that if one uses a general value for $(r+k)$ and proceeds as above, then the expression given above for β is nonetheless an upper bound. This expression for the size of the dependency of a standard ball on the toroidal grid is valid only when the diameter of the standard ball is smaller than the smallest dimension of the grid. (As the ball wraps around the grid, its dependency increases more rapidly than the above expression.) Let's suppose the grid is $l_1 \times l_2$ sites on the two-dimensional torus, and $l_1 \leq l_2$. The restriction is then that the radius of $B(v_n)$ is less than l_1, that is, $n < l_1/2$. So, there is a restriction on $(r+k)$:

$$r + k \leq \frac{(l_1)^2}{2} + l_1 + 1 , \tag{7}$$

which comes from substituting for n in (5). It's worth pointing out at this point that the pebbling argument gives an essentially infinite bounding value for the throughput when $(r+k) \geq l_1^2$, since the dependency of a set of this size contains the entire computation graph. This is why the pebbling argument is not an asymptotic result in terms of r. Consequently, the limitation on $(r+k)$ given by (7) is not a serious restriction.

We want to compare V against an equivalent expression for the throughput of the LGM-1 machine. The LGM-1 is an implementation of the ''WSA'' architecture introduced in [23]. The WSA architecture consists of an s stage pipeline of identical processors through which the lattice-graph data is passed in a raster scan order. Each stage of the pipeline contains W update units (ALU's), and produces W updated lattice sites per global machine clock tick. The lattice exits the pipeline in raster scan order updated s time steps. See figure 2.30. Suppose the capacity of the communication channel between the pipeline and the main memory is again V_{com}. As we mentioned in the introduction to this article, the LGM-1 machine pipeline is much faster than the memory channel. Consequently, suppose the pipeline clock is triggered by the communication channel and the channel never waits on the pipe. As $2W$ lattice sites are transferred per machine clock tick and W lattice sites updated per pipeline stage per tick, the speed of the machine is then

$$\begin{aligned} V_{WSA} &= \frac{V_{com}}{2W} \frac{(\text{sites/sec})}{(\text{sites/tick})} \; s\,W\,(\text{site updates/tick}) \qquad (8) \\ &= \frac{V_{com}}{2} \, s \,(\text{site updates/sec}) \;. \end{aligned}$$

Since each stage of the pipeline has storage for $2l_1 + W$ sites, we have the total processor memory as

$$r = s(\, 2l_1 + W\,) \;.$$

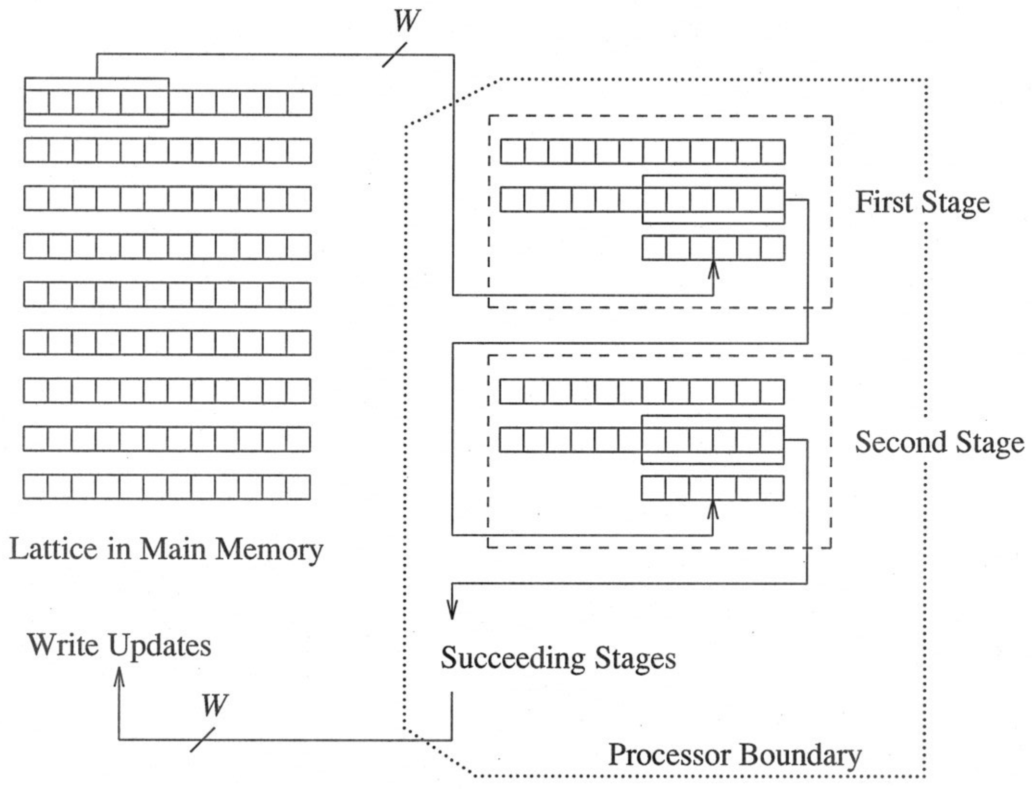


figure 2.30.

The WSA architecture for 2-d lattice-graph computations. The lattice data is stored in a ''main memory'' and fed to a pipeline of processors. The global machine clock ticks once for every transfer of W lattice sites into the pipe. At the same time, W sites are transferred from the pipe to memory. Also, each stage of the pipe passes W updated sites to the next stage down the pipe. Each stage has enough memory to hold two complete lines of the lattice, plus W sites, and has W processors. The pebbling argument will be applied with the division of resources shown by the dotted line enclosing the pipeline. Although the communication lines are shown as W sites wide as they are in the LGM-1 machine, they could, of course, be any width.

Solving for s and substituting for s on the right in (8) gives

$$V_{WSA} = \frac{V_{com}}{2} \frac{r}{2l_1 + W} .$$

This is the ''raw'' processing capability of the WSA architecture as a function of communication bandwidth and the amount of local storage. This throughput is achieved by the WSA architecture in the limit as l_2 becomes infinite. We will consider the effect of a finite l_2 later, but let us first see what the general form of the ratio V_{WSA}/V is.

First, note that k is a free parameter of V and we may adjust it to maximize the ratio V_{WSA}/V. We have estimated an optimal value for k as $k = 2r$ using symbolic and numeric means employing Mathematica[†]. When l_2 goes to infinity, λ becomes 1. Taking s as $r/(2l_1)$, and dropping the lower order terms of β, we have

$$\frac{V_{WSA}}{V} \approx \frac{\frac{V_{com}}{2}\ \frac{r}{2l_1}}{\frac{V_{com}}{2}\ \frac{(6r)^{3/2}}{12r}}$$

$$\approx \frac{\sqrt{r}}{\sqrt{6}\,l_1} .$$

An approximate upper limit for r of $\frac{(l_1)^2}{6}$ comes from substituting $2r$ for k in (7). Setting r to this limit we get

$$\frac{V_{WSA}}{V} \approx \frac{1}{6} . \tag{9}$$

So, the WSA architecture is within a factor of 6 of the optimal throughput, given the fixed resources r and V_{com}. We leave it as a conjecture that the WSA architecture has throughput of at least two-thirds the optimal throughput for fixed resources. There are two facts that strongly support this conclusion.

The first is that the pebbling argument giving the expression for V ignores output I/O: the parameter k actually divides the pebbling into pieces with k inputs steps each. All that is necessary to introduce a factor of 2 improvement in the bound is for the input nodes of a sub-pebbling kernel to be identified with nodes in proximity to the nodes in the kernel. In that case, since the input in most cases must come from nodes computed in the processor, these input nodes must have been written to main memory at some point. Thus, the output I/O is the same size as the input I/O. Then, in the expression for V the denominator becomes $2\lambda k$ instead of λk.

The second has to do with the storage of information. In the pebbling argument, one bounds the amount of computed information in a single I/O sub-division by, in essence, supposing one starts with $(r+k)$ red pebbles and pebbles as much of the computation graph as possible, and these nodes are never recomputed. Suppose one can

† Mathematica is a trademark of Wolfram Research, Incorporated.

identify these nodes spatially in the data dependency set. As the sub-pebbling shape theorems (Theorems 3 and 4) show, these sets would be cones. Applying a little elementary geometry to these cones, it is easy to see that, if one were to proceed in this manner, computing the maximum possible volume of nodes, that at least 3/4 of the nodes computed require recomputation. Suppose we assign the input I/O costs to the nodes by averaging the I/O cost by the number of nodes in the sub-pebbling. The nodes that do not require recomputation incur an average input I/O cost of $k/\beta(r+k)$ each from the sub-pebbling, while the recomputed nodes incur at least twice this much. Summing the input I/O cost over the all the nodes in the data dependency graph results in 7/4 the cost when not considering recomputation, introducing nearly another factor of two in I/O cost.

The preceding two plausibility arguments suggest that a different way of looking at the pebbling is required to improve the bound. This new approach must spatially associate nodes with nodes that are read in support of red-pebbling them. The current I/O-division of the pebbling game associates nodes temporally with input operations. That is, one is really not concerned with when a particular group of nodes is computed, but what the shapes are of regions of nodes that never get blue-pebbled, and the shapes of regions that get recomputed.

The bound in (9) was for an infinite lattice, $l_2 \rightarrow \infty$. When l_2 is finite, the speed V_{WSA} decreases by a factor dependent on the number of stages in the pipeline because a toroidal lattice must be cut in order to allow it to be fed into the pipeline. The sites on one side of the cut are updated incorrectly since the portion of the lattice on the opposite side of the cut is not available. Consequently, one line of the lattice is corrupted at one side of the cut and another on the other side of the cut for every stage in the pipeline. The solution is to pad the input with sufficient lines from the other side of the cut to account for this. Figure 2.31 shows the input preparation required. In that figure one can see that $2s$ lines (or rows) of the lattice must be copied into the portions labelled Z; so, there are an additional $2sl_1$ site values added to the input stream. At the tick marking the input of the last set of W sites into the first stage of the pipeline, the first stage shifts in this last data item (the last group of W sites), and during the time to the next tick, calculates its last correct result. This last result from the first stage is available to the second stage at the next tick. As shown in figure 2.32, the last correctly updated group of W sites from the second stage is calculated one tick after the last correctly updated group is produced by the first stage. In general, the last correct piece of data is output from the last stage of the pipeline s ticks after the last ''real'' input. Thus, if there are z items of input, the pipeline must operate for $z+s$ ticks to output all the

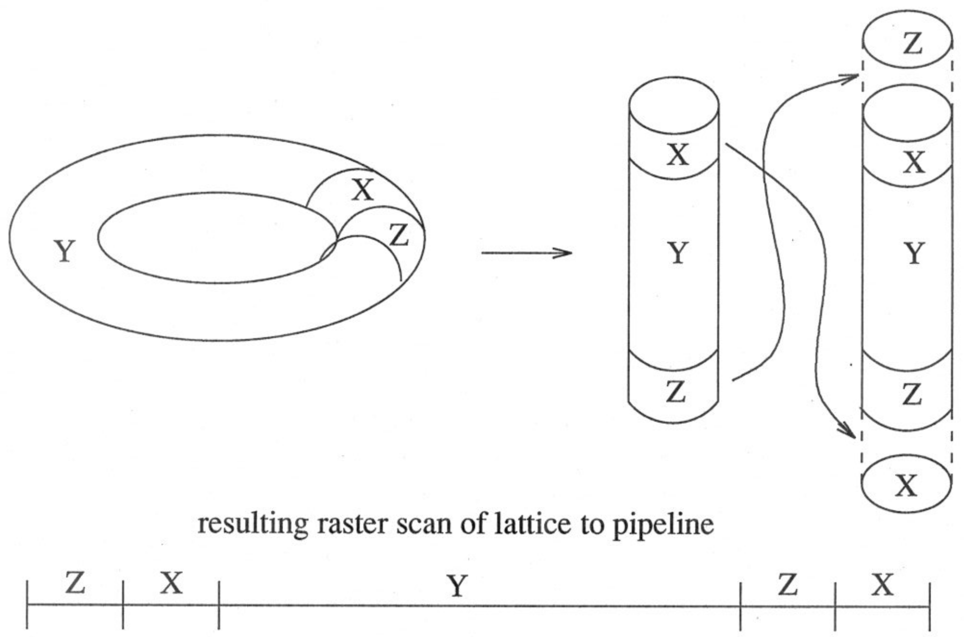


figure 2.31.

Preparing pipeline input of a toroidal lattice. Shown on the left, the torus is cut arbitrarily. The two portions abutting the cut, X and Z, each contains s lines (or circles). The torus is unfolded and the the two abutting pieces are copied so that the information on the opposite side of the cut is available when the X and Z sections are updated. The resulting raster scan of data to the pipeline is shown at the bottom of the figure. In a left-to-right scan, the Z portion is scanned in first, followed by the X portion, the Y portion, the Z portion again, and finishing with the X portion again.

useful update information available, and z ''real'' inputs and s ''dummy'' inputs must be supplied to the first stage as input. The real inputs for an $l_1 \times l_2$ site lattice amount to $zW = l_1 l_2 + 2sl_1$ sites. So, the pipeline produces $s(z + s)W$ total results in the process of updating the lattice s times. The ratio of effective work to actual work done by the pipeline to update a $l_1 \times l_2$ lattice s times is then

$$\frac{s(l_1 \times l_2)}{s(l_1 \times l_2 \ + \ s2l_1 \ + s)} \frac{\text{(effective updates)}}{\text{(actual updates)}} \ .$$

So, the *WSA* throughput becomes

$$V_{WSA} \quad = \quad \frac{\frac{V_{com}}{2} r \, l_1 l_2}{(2l_1 + W) \left[l_1 l_2 + \frac{r}{2l_1 + W} (2l_1) \right]} \qquad (10)$$

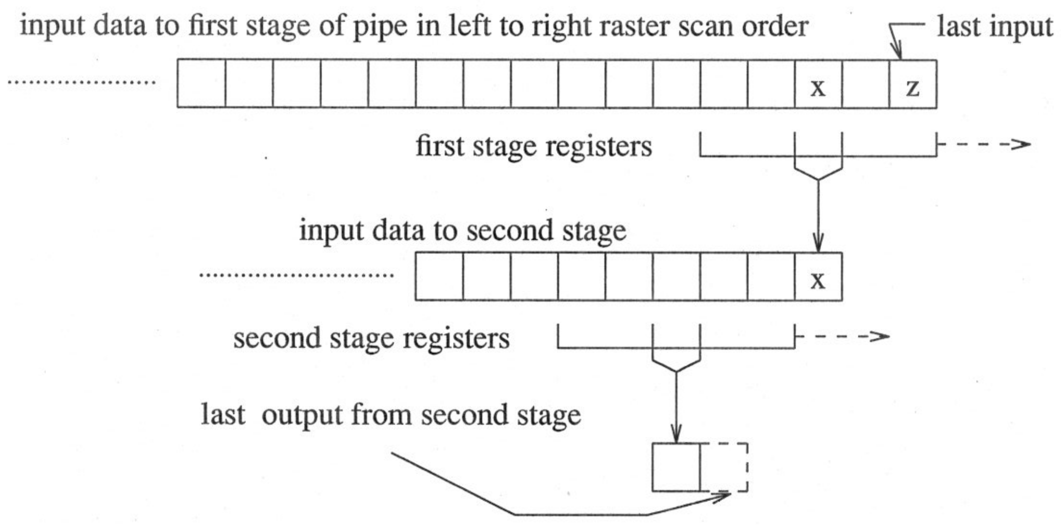

input data to first stage of pipe in left to right raster scan order
last input
x
z
first stage registers
input data to second stage
x
second stage registers
last output from second stage

figure 2.32.

The timing of pipeline output. The input to the first stage of the pipeline is shown as a left-to-right scan of groups of W sites. The last group is labeled ''z''. Below the input is shown a representation of the shift registers of the first stage. The registers can be thought of as moving to the left and containing within them the information directly above. At the z^{th} tick, the first stage shifts in the last ''real'' group of W sites. By the end of this clock period, the group of sites labeled ''x'' is calculated and available to be shifted into the second stage registers (or shifted into main memory, if no second stage exists). The second stage contains the group ''x'' at the $(z+1)^{th}$ tick. Thus, s dummy groups must be shifted into the first stage to produce the last usable result from the s^{th} stage of the pipeline.

Figure 2.33 shows a typical plot of the ratio $\theta = \frac{V_{WSA}}{V}$ as a function of local storage capacity r over the allowable range of r, $2l_1 \le r \le \frac{(l_1)^2}{6}$. The resulting value of θ ranges from approximately $\frac{1}{26}$ to approximately $\frac{1}{5.7}$.

We began this paper with the intention of estimating the degree of optimality of the throughput of the LGM-1 machine with respect to main memory bandwidth and local storage. Recall that the LGM-1 computes a lattice gas on the two-dimensional triangular lattice, and that at the beginning of section 2.3.3 we stated that we would not show the standard spheres for the triangular lattice, $B(x)$, were extremal for arbitrary x. Rather, we showed the specific standard spheres $B(v_r)$ were extremal, and suggested that the same general program that was used to show the standard spheres for Z_2 form an extremal sequence could be applied to the standard spheres for the triangular lattice. So, with the caveat that this has not been done here, and that we are assuming the

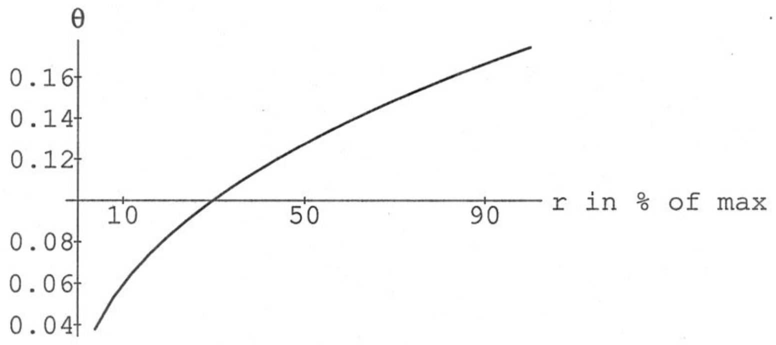


figure 2.33.

The ratio $\theta = \frac{V_{WSA}}{V}$ *for the two-dimensional toroidal grid (see (10)). The argument r, total local memory capacity in lattice sites, is shown in percent of its maximum value as determined by the pebbling argument validity restrictions. The plot is shown over the permissible range of r. The curve is typical over the range of parameter settings. Here the parameters are set to match the LGM-1 resources and its typical usage:* $l_1 = 256, l_2 = 1024, z = l_1(l_2)^2$. *The range of* θ *is from* $\frac{1}{26.3}$ *to* $\frac{1}{5.73}$.

validity of the extremal sequence $B(x)$ for the triangular lattice, we now repeat the steps shown above in the comparison of the WSA architecture, substituting the the triangular lattice for the grid. Using the same parameters as before, we get the curve shown in figure 2.34 for $\theta = \frac{V_{WSA}}{V}$ for the triangular lattice. There θ ranges from $\frac{1}{21.3}$ to $\frac{1}{4.2}$. In order to simplify the design the LGM-1 machine actually uses twice the local storage required by the WSA architecture. Consequently, the performance ratio $\frac{V_{LGM}}{V}$ is about one-half that of $\frac{V_{WSA}}{V}$.

2.5. Summary, Conclusions, and Future Work

This chapter presented a refinement of existing pebbling games in order to model I/O in parallel computations, defined precise characterizations of the subsets in a pebbling, developed estimates on the sizes of these subsets, and used these estimates to establish an upper bound on throughput for two computational problems, the lattice-graph based computations on the two-dimensional toroidal grid and the two-dimensional triangular lattice on the torus. The result shows that the throughput of the

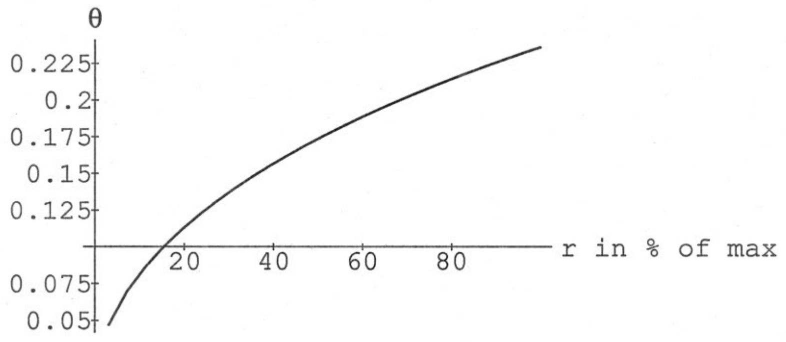


figure 2.34.

The ratio $\theta = \frac{V_{WSA}}{V}$ *for the two-dimensional triangular lattice on the torus. The setting of the parameters are the same as those used in figure 2.33. Again, the range of r is restricted to values valid under the pebbling argument assumptions.*

WSA architecture is within a small factor of the throughput of any machine with identical fixed resources of main memory communication bandwidth and local storage capacity. The factors are about 6 for the two-dimensional grid, and 5 for the triangular lattice-graph. That is, the WSA architecture (a linear pipeline machine) runs a least $1/6^{th}$ the speed of any machine computing similar problems in two-dimensions with similar resources. So, for these types of problems, no other organization of computation steps and internal communication (internal to the processor) —even given infinite internal communication capacity and infinite calculating power (arithmetic-logic operational speed)— can attain significantly better performance, given the fixed amount of local memory and communication to main memory available.

It is not surprising that a pipelined machine makes efficient use of communication resources. In fact, approximate bounding values for throughput were known through previous work, but only to within one or two orders of magnitude of the present results. And while the suspicion that reorganization of the computation steps and machine resources could not improve LGM-1's throughput by more than a small amount initially motivated the work in this chapter, there was another motivation as well. The motivation behind this work was not only to find out precisely how good or bad the WSA architecture is, but the considerable effort put into refining the pebbling arguments stems from the desire to find a method of determining an optimal computational strategy and thereby the most powerful machine possible within the resource constraints.

To find an optimal strategy, we need to understand the nature of the forces induced by resource constraints and how these forces work to form an optimal parallel computational strategy. The optimal strategy is embodied in the moves of an optimal pebble game and can be determined from the shapes of the pebbling sets defined by sub-pebblings: if the shapes of sub-pebblings in an optimal pebbling can be discovered, the optimal order of computation and I/O strategy will be known. Before this effort, the pebbling arguments gave little or no clue to the shape of a sub-pebbling. The present work makes some headway in this direction by suggesting the shapes of sub-pebbling sets when the only constraint is to maximize the number of nodes calculated. The candidate sets are the (ISO-1)-extremal sets presented in section 2.3. We have shown that they can be characterized essentially as cones within the data-dependency graph.

However, the present method of analyzing the pebble game is done only on a temporal basis: the nodes associated in a sub-pebbling set do not necessarily have any connection spatially, but are associated by their proximity in time in the pebbling moves. Consequently, the shapes of these sets do not give direct information about the optimal strategies. An area for further investigation is to add this spatial information to the pebbling analysis. For example, as was mentioned in section 2.4, a factor of two improvement in the bounds derived here can be attained easily, if the spatial relationship of nodes was characterized by the extremal sets presented here. The possible improvement in the bound is a consequence of the fact that the present analysis characterizes only input I/O, and the observation that a node that is not an input node and is used for input must have been used as output previously. If the sub-pebbling sets were spatially related, one could determine where in the graph the nodes used for input were, and thereby determine which nodes where used for output.

In the process of refining the pebbling set size estimations, this chapter also introduces the use of the Wulff Construction to solve a discrete iso-perimetric problem. The method introduced here solves a discrete isoperimetric problem by starting with the solution of a closely related continuous isoperimetric problem and deriving the result for the discrete case. This approach makes an interesting counterpart to previous work by Bollobás and Leader [27] which works in somewhat the reverse direction by ''smoothing out'' a discrete problem to a more manageable continuous one. An interesting question is whether the two methods can be used to show a way of connecting continuous and discrete problems more closely.

Aside from the application of Wulff's Construction to isoperimetric problems, there is another possible use for this tool. It would be interesting to attempt to apply the Wulff Construction directly to the data dependency graph. This would require a careful definition of the integrand to correspond to the costs of recomputation and storage of information. One can look at the data dependency graph as dynamically maintaining a red-and-blue-pebbled surface that descends in the graph as the computation proceeds. The cost associated with the surface of a crystal set would have to correspond to the cost of maintaining this red-blue surface.

Finally, another direction open to investigation has to do with applying the pebbling techniques to expanded classes of machine architecture. The communication bound derived previously used the terminology of red and blue pebbles. The essential distinction was simply that the registers associated with the blue pebbles communicated through some limited capacity channel with the registers associated with red pebbles. In fact, there was no restriction made that every register in the machine was either red or blue. It is easy to see that the machine can be divided up into as many different pieces as desired and each piece assigned a color. The form of the communication bound between different pieces remains the same. This would allow the modelling of communication in a distributed or hierarchical context. Consequently, another area for future work would be the extension of the present methods to allow the analysis of hierarchical and distributed machine designs.

Chapter 3
Optimal Machine Scaling

3.1. Introduction

An important property of any architecture for large scientific computation, such as lattice gas simulations, is scalability. Generally, once the answer to a problem of a given size is known, attention turns to scaling the size of the problem up. This scaling of problem size involves three goals in scaling the corresponding computing machine: (1) scaling throughput, (2) accommodating the new dimensions of the input data, and (3) achieving both of these goals as cost-effectively as possible. The previous chapters have established that the WSA architecture has nearly optimal throughput for any organization of hardware designed for lattice-gas simulations in which the hardware must communicate with a main memory. As well, the general organization of this architecture can be applied to any computation whose data dependency graph is similar to those of the lattice-gas simulations, regardless of the size of the operands represented by the nodes or the complexity of the operations required to produce the values in the nodes. These features make this architecture an interesting one for investigating optimal scaling as defined by the three goals above.

Before we discuss the scaling of the WSA architecture, it is worthwhile to point out a few obvious aspects of scaling in a general context. For the class of high-speed general-purpose machines goal (2) above is of secondary importance because of their flexible data handling capabilities. However, scaling throughput is usually cost-effective only over a small range of improvement in performance. Beyond that range, an entirely different machine is required because the original machine generally has one or more performance bottlenecks for a specific problem that cannot be removed by adding hardware.

For the class of special-purpose machines both goals (1) and (2) can be serious obstacles to cost-effective scaling. Even for special-purpose machines that theoretically can scale throughput, goals (1) and (3) often conflict. For instance, scaling a two-dimensional array processor quickly becomes unattractive because the cost of wiring the two-dimensional processor communication grid across processor boards becomes excessive. Similarly, goals (2) and (3) often conflict because special-purpose machines also often have built in restrictions on the dimensions of the input data.

The WSA architecture is a special-purpose machine for which goals (2) and (3) also conflict: throughput can be scaled cost-effectively by adding additional pipeline stages, but one dimension of the input data is fixed by the design of the custom VLSI chips. This restriction can be eased by using an overlap-save method. Once this method is employed there is no reason to restrict the architecture to a single pipeline. The resulting architecture we call the Multiple Wide Serial Architecture (MWSA). As the previous chapters showed, the main-memory bandwidth is critically important in determining throughput. The MWSA architecture allows scaling of I/O bandwidth as well as all other resources of the machine. Therefore, we can investigate the most cost-effective combination of resources as a function of increasing problem size. The result is that, over a fairly large range of problem size, the MWSA architecture achieves linear speedup as a function of machine cost. The conclusions hold for any other iterated computation with similar layer graphs, including those whose operands at the nodes of the data dependency graph have arbitrarily many bits.

The remainder of the chapter is organized as follows. Section 2 defines speedup and our concept of machine cost. Section 3 describes the overlap-save method as it is used with a single WSA pipeline. Section 4 derives the computational efficiency of a single WSA pipeline using the overlap-save method. Section 5 describes the MWSA schemes for scaling all resources of the WSA pipelines. Section 6 defines the cost function for the MWSA machine, and derives the objective function for finding the maximum throughput configuration at the minimum cost. Section 7 presents the results of the numerical optimization on the objective function presented in Section 6. Section 8 gives a summary and our conclusions. And Section 9 presents open questions and possible future work.

3.2. Speedup and Cost

The usual measure of effectiveness applied to scaling parallel computing machines is speedup. This is usually taken to be a function of n, the number of processors employed in a machine. However, the number of processors is a crude estimate of the cost and complexity of the machine. In fact, one is not really concerned with how many processors a machine has, but how much it costs to build the machine. Particularly when expanding a machine, the question is whether the incremental cost of expanding the machine is reasonable given the resulting throughput. ''Reasonable'' usually means an incremental increase in cost for a incremental increase in throughput. Consequently, we say that a machine has *linear speedup* if the throughput is a linear function of the cost of wiring, local memory and processor chips, data paths, and so on.

Notice that the cost of main memory was not mentioned. The assumption here is that memory must be paid for, if the machine is to hold the problem at all, and that this cost is therefore a constant regardless of the processor used to do the computation. If the entire machine, memory and processor taken together, is evaluated, then the cost may not be linear even though the cost of the processor alone is. In fact, if the memory is random access, the cost probably cannot be made linear as the size of the memory is scaled. However, we will expand the memory by adding memory modules with independent ports so that the memory as a whole does not act as a random access memory. We will not address the question of whether or not such a memory can be scaled linearly, and, consequently, we will ignore the memory cost in scaling the WSA architecture, except as it is reflected in the cost of new ports to new memory modules.

In the definition of linear speedup, the intention is that the cost of the machine is the least cost for the throughput demanded. However, this can conflict with the notion of scalability if the organization of least cost for a given throughput is incompatible with the organization of least cost for a greater throughput. It is intended that, in scaling up a machine, the existing machine is added to, not rebuilt from scratch. This is not the general meaning of the term ''scaling up'' for a machine architecture. Here, the context within which this scaling is to take place restricts our particular notion of scaling. As we will explain below, the context is that of requiring existing hardware to handle ever larger problems of the same type. This is motivated by the economics of scientific computation in that the money for a machine is not infinite, and as time progresses, more money becomes available to expand the research effort. Consequently, our bias is to assume an existing machine is added to incrementally. Further, we assume that the available parts are fixed, that is, chips have a fixed feature size, area, and communication capacity (''pins''), and so forth. For instance, if the shift-registers are contained on-chip, then this organization limits the size of the shift-registers when scaling the machine. Consequently, we will look at the most cost-effective tradeoff of parameters for a given organization (partially determined by the fixed sizes of available parts), and determine speedup as a function of the specific cost function for that organization of resources. Specifically, this will mean that we will not address the the general question of the most cost-effective machine design for a specific throughput performance level, or the effect of technology improvements such as denser chip technology and advances in inter-chip communication capability. However, the results we will show apply, with proper scaling of the costs and capabilities, to the optimal design of a machine using whatever fixed technology is available to the designer.

3.3. The Overlap-Save Method

Before we expand on the context of machine scaling mentioned above, we need to give a brief introduction to the overlap-save method. The overlap-save method of computation allows the computation of problems whose size is larger than can be accommodated naturally by the computing hardware and is a particular type of problem decomposition and sub-problem communication strategy. In the systolic processing literature the problem of fitting large problems into smaller arrays is known as ''partitioning'' (see [34], for instance). Two common methods are called ''coalescing'' and ''cut-and-pile.'' These two methods preserve the communication pattern of the original problem by assigning a region, or ''block,'' of the problem data to each processor [35, 36]. Communication with memory or processors holding surrounding blocks is done whenever information from neighboring blocks is required.

Another approach, called ''overlap-save'' in the signal-processing literature [37], is based on a method that probably predates modern computers†. This technique generally consists of processing a block of the problem by padding it with information from neighboring blocks and computing a result without using any further I/O. The padded data is sufficient to allow the computation of any results that might be needed from neighboring blocks, and thus the shared ''edges'' between blocks are computed multiple times. A recent example of a custom pipelined processor for video processing employing this method can be found in [38].

3.4. Overlap-Save Applied to WSA

The overlap-save method has another motivation besides allowing the introduction of multiple pipelines. As was pointed out by Snyder [33], large scientific problems can be considered ''fixed time'' computations because a scientist is generally interested in solving the biggest problem possible in the time she can afford to wait until results are available. It follows that increases in computing capability will be used to solve larger problems of the same type in this fixed ''tolerable'' amount of time. However, this ''tolerable time'' is only approximately fixed because the situation surrounding the use of the machine and the importance of the delay in getting the result vary. For instance, a tolerable delay during the working day may be several minutes to an hour for the average level of importance, but a simulation may be left running overnight, or

† Richard Feynman relates an anecdote in his book ''Surely You're Joking, Mister Feynman'' which seems to suggest the use of a method similar to overlap-save by a team of people using hand calculators to find numerical solutions to some unspecified engineering problem.

even over a weekend, if the machine is usually unused at those times. Even a delay of a year may be acceptable, for instance, when the simulation result cannot be gotten any other way. As a consequence, a machine that can turn around a job in the minimum tolerable delay should also be able to handle larger jobs with increased delay. Since it is important to be able to handle larger problems than the hardware naturally accommodates, and the overlap-save method allows this even for a single pipeline, it is reasonable to look at the WSA architecture as always using overlap-save.

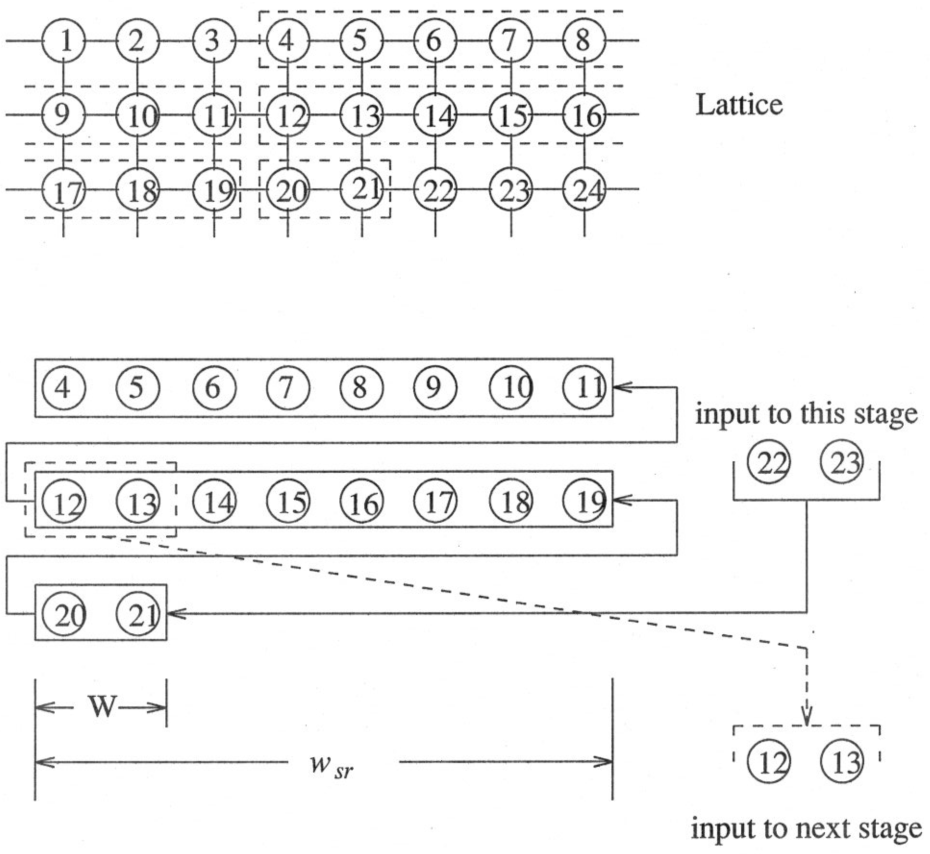


figure 3.1.

A portion of a two-dimensional grid lattice and the shift-registers of one pipeline stage of the WSA architecture for the two-dimensional grid. Here, values for sites 22 and 23 are about to be shifted in, and updated values for sites 12 and 13 are ready as output. Above is a portion of a lattice: the sites whose values are currently held in the shift-registers are shown in dashed boxes superimposed on the lattice. Below are the shift-registers of one stage of the pipeline with their current contents shown. The lattice sites are shifted in one ''word'' at a time (in this case the word size W is 2 sites), and one word of output is produced. The ''width'' of the pipeline, w_{sr}, is 8 site values.

The LGM-1 machine was originally designed to operate on fixed-width lattices determined by the width of the local shift-registers in each stage of the pipeline (see figure 3.1). As that figure shows, there is only one shift-register per stage, but it is convenient to view it as consisting of three serially connected shift-registers: two shift-registers w_{sr} wide where the ''pipeline width'', w_{sr}, is the number of lattice sites in a single ''row'' of a ''natural'' lattice; and a third shift-register holding the W site values where W is the ''data path width'' and is the number of site values passed between stages per global machine tick. The ''natural'' row size for the machine is determined by the total shift-register capacity per stage, that is, if the total capacity is x, then the natural lattice row size is $w_{sr} = x/2 - W$.

We adopted the overlap-save method for use in LGM-1 so that we could process lattices with rows wider than the width of the local shift-registers, w_{sr} sites wide. In this case, the lattice is divided into vertical slices, or blocks, each block is padded to form a sub-lattice that fits the width of the pipeline, and the sub-lattices thus formed are fed individually into the pipeline in raster scan order. See figure 3.2. When every block has passed through the pipeline, the process is begun again with the first block. Processing the lattice in blocks like this allows the processing of arbitrary size lattices. However, it also introduces overhead because the the processors only have communication capability in one direction: down the pipe. Consequently, there is no communication with neighboring blocks when the sites at the edge of the sub-lattice being fed to the pipe are updated. One can imagine bad information traveling inward one column or row for each iteration from any edge of the sub-lattice and corrupting the site information. So, for a one-stage pipeline, the left and right columns of the sub-lattice will contain incorrect values when the block exits the pipe. If the pipe has s stages, then the left s columns and right s columns will be corrupted. In addition, the first and last s rows will also be corrupted. Consequently, only the core of the sub-lattice is saved: the s wide border of the sub-lattice is discarded. Thus, if the shift registers are w_{sr} sites wide, the lattice is sliced into blocks $w_{sr} - 2s$ sites wide, and each slice is padded with s columns from its left neighboring block and s columns from its right neighboring block. Additionally, the top of the block is padded with s rows and the bottom likewise. See figure 3.3.

The WSA architecture on which the LGM-1 was based can be implemented in such a way so as to avoid the corruption of sites at the left and right edges, but only when the lattice row size matches the shift-register width, and when the lattice is assumed to be toroidal (so called ''wrapped around'', or ''periodic'' boundaries). When the lattice has non-periodic boundary conditions (not toroidal), the edges still propagate bad information, unless special hardware provisions are added to introduce

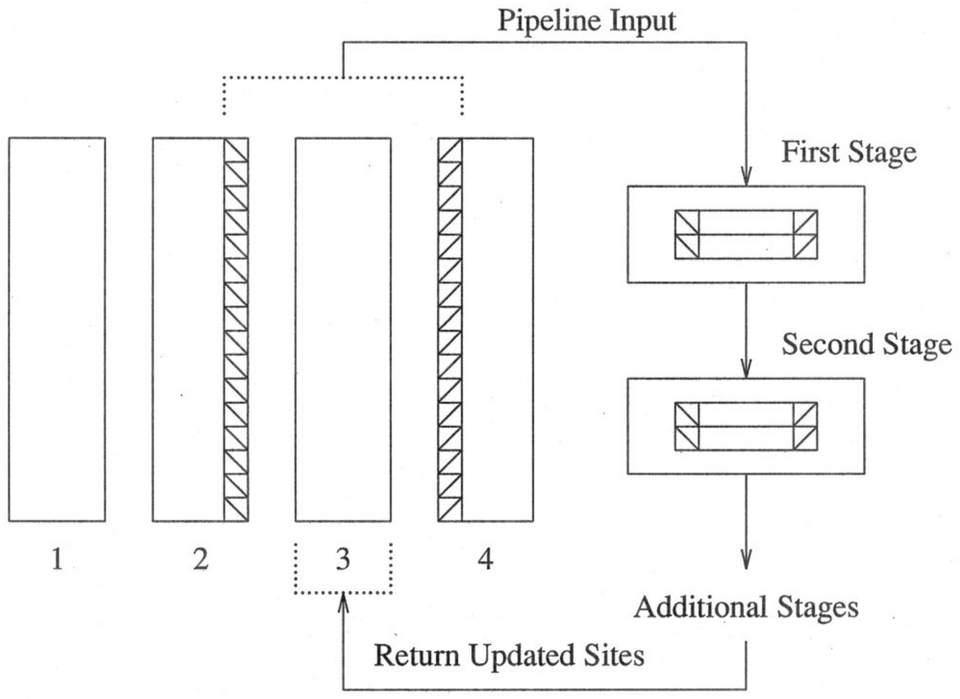


figure 3.2.

The overlap-save method as used in the LGM-1 machine. The lattice on the left is partitioned into blocks numbered 1 to 4. Each block is fed individually to the processor pipe. The block (3) being fed into the pipe is padded on the left by s columns from block 2 and padded on the right with s columns from block 4 (the individual cross-hatched squares represent groups of s adjacent sites on a single row of the lattice). The pipeline width, w_{sr}*, is the width of the blocks plus* $2s$*. The top and bottom of the block also require s rows of padding, but this is not shown. If the pipe has s stages, then this is just sufficient padding so that all the sites in block 3 are updated correctly and written back to the lattice storage as they exit the pipe. The padded sites are corrupted by the pipe and are therefore thrown away.*

boundary information. This was proposed in [10] as an additional special pipeline stage for the LGM-1. The LGM-1 was built with the goal of simulating lattices with boundary conditions which were non-periodic. The lattice was assumed to be embedded in an infinite lattice with an induced flow of particles. In order to correct the corrupted sites after a number of iterations, a special chip was to be inserted in the pipeline every ten stages. This chip would fill the ten columns on each side and ten rows at the top and bottom with random site values biased so as to correspond to the induced flow conditions. This solution works well for the fixed-size lattice and results in a machine with linear speedup (we will come back to the issue of speedup later). However, larger lattices with non-periodic boundaries will still suffer corruption at the left and right edges of blocks, and furthermore, lattices with periodic boundaries at the top and

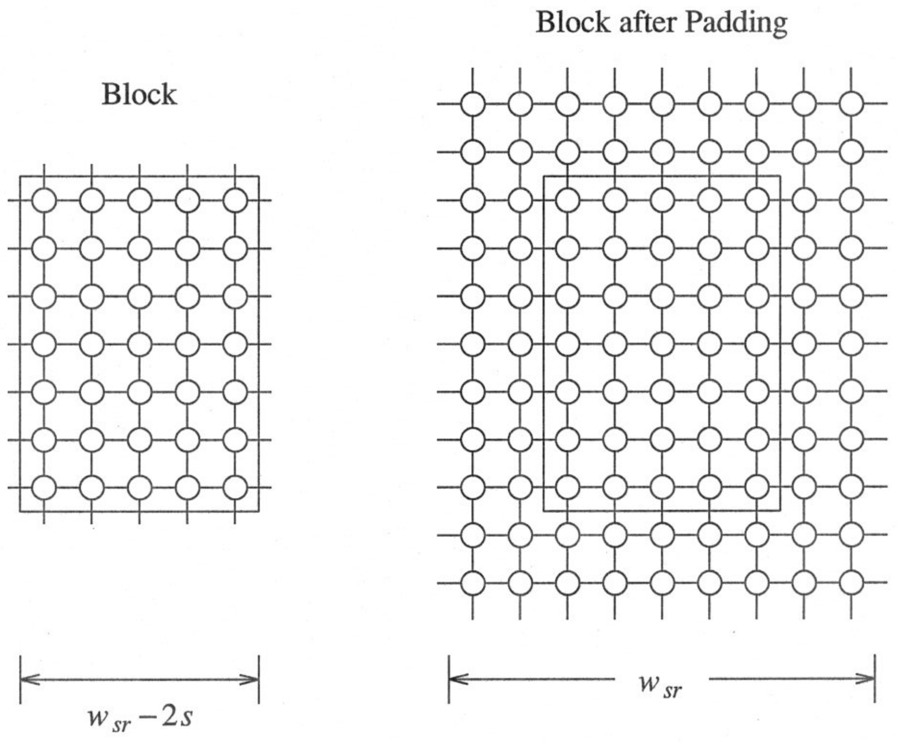


figure 3.3.

Padding a block of a grid lattice. The block consists of the nodes shown inside the rectangle. If the pipeline has s stages (here s = 2), then the left and right edges are padded with s columns of nodes from neighboring blocks. Additionally, s rows at the top and bottom are added. The WSA architecture would have shift-registers with width w_{sr} = 9 for the block shown. In this figure it is apparent that the block could be a rectangular section from the interior of the lattice. In general, there is no advantage in the block having fewer rows than the lattice itself has.

bottom will suffer corruption at all edges. The actual method we used with the LGM-1 machine for setting non-periodic boundary conditions was to precompute a large circular buffer of properly biased random site values. These site values were then used to pad the top and bottom edges of the lattice, to pad the left edge of the leftmost block, and to pad the right edge of the rightmost block. In general, this is the overlap-save technique. Of course, when the boundary conditions were periodic the overlap-save method was also used. (In fact, it was found that for monitoring and testing purposes, it was always advantageous to have periodic boundaries. The non-periodic boundaries were then implemented by padding the ''inside'' of the lattice with random site values.) The performance is identical whether one has periodic boundaries or not, if the lattice is wider than the pipeline. Consequently, we will concentrate our attention on the speedup and cost-effectiveness of the WSA architecture employing the overlap-save technique on the torus.

3.5. WSA Efficiency and Throughput

We first want to determine the throughput of a single WSA pipeline using overlap-save. We can express the throughput V_{WSA} as the product of the raw processing capability and the efficiency in terms of effective site updates per total site updates computed in a fixed time. To get the efficiency of a single WSA pipeline we use the space-time approach [39]. Let the pipeline's processing word size be W sites (W sites are shifted in, and W updated sites are shifted out, in a single shift operation). Let the line length of its shift-registers be w_{sr} sites. And suppose the lattice's total size is $l_1 \times l_2$ sites: l_1 sites per row, and l_2 rows. Now, we can look at the pipeline as processing a w_{sr} wide lattice that is infinitely long (infinite number of rows): the garbage in the shift registers at startup are considered simply site values of an infinite lattice. Just before the first site is read into the first stage of the pipe, we can think of the first stage as updating the W sites lying w_{sr} backwards in raster scan order from the first input site. See figure 3.4.

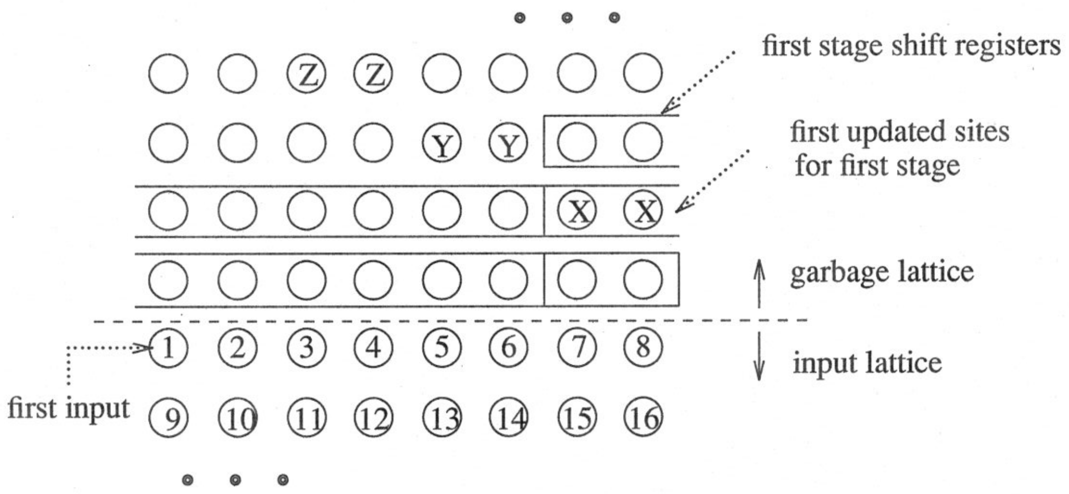


figure 3.4.

The pipeline at the start of input. The garbage data already in the shift registers can be considered part of an infinitely long lattice. The first stage of the pipe will shift in sites 1 and 2 at the first tick of operation (W is 2 in this picture). At that time, the garbage sites marked "X" will be updated by the first stage. Likewise, the second stage will update sites "Y", and the third stage sites "Z."

Since the second stage is reading the output of the first stage, it is positioned $W + w_{sr}$ sites further back in the lattice. By the time the first stage has reached the position of updating the first input site, it will have updated $(W + w_{sr})$ garbage sites. Similarly, the i^{th} stage of the pipe will have updated $i(W + w_{sr})$ garbage sites by the time site 1 is in a position to be updated the i^{th} time.

Let us look at the total space-time used by the last stage of the pipeline, stage s. There will be $s(W + w_{sr})$ garbage sites computed by this stage. The pipeline is halted as soon as the last good output exits the s^{th} stage. With a pipeline width of w_{sr}, the block size must be $(w_{sr} - 2s)$ by l_2. The last output from this stage is the last site in this block. This stage must update s rows of padding at the top of the block. This amounts to sw_{sr} sites. After that, to update all but the last row of the block requires updating $(l_2 - 1)$ rows of w_{sr} sites each. Finally, the last row of the block requires $(w_{sr} - s)$ site updates since the right-hand s columns of the padded sub-lattice are useless and are not needed as space holders. See figure 4.5.

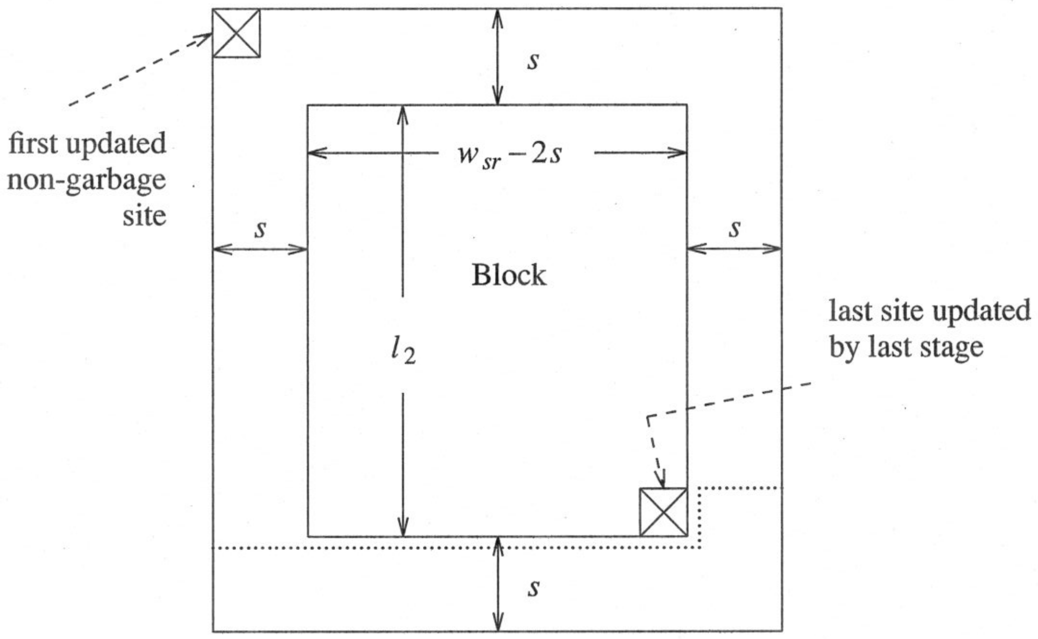

first updated non-garbage site
s
$w_{sr} - 2s$
s
s
Block
l_2
last site updated by last stage
s

figure 4.5.

The lattice sites updated by the last stage of the pipeline, excluding garbage sites. The block is shown with s padding on all sides. The first non-garbage site to get updated and last updated site are shown by the small rectangles with cross diagonals. If the raster scan order is left-to-right and top-to-bottom (row major order), then all sites above the dotted line are updated by the last stage of the pipeline.

The total space-time is identical for every stage, so we have

$$Space-Time/Stage \quad = \quad s(W + w_{sr}) \ + \ sw_{sr} \ + \ (l_2 - 1)w_{sr} \ + \ (w_{sr} - s)$$

$$= \quad l_2 w_{sr} \; + \; s\,(\,2 w_{sr} \; + \; (W - 1)\,)\ .$$

The space-time for the last stage that corresponds to real output is simply the size of the block. Since this is the same for every stage, we have

$$Effective-Space-Time/Stage \quad = \quad l_2\,(w_{sr} \; - \; 2s)\ .$$

The efficiency, e, is then

$$e \quad = \quad \frac{Effective-Space-Time}{Space-Time} \quad = \quad \frac{l_2\,(w_{sr} \; - \; 2s)}{l_2 w_{sr} \; + \; s\,(\,2w_{sr} \; + \; (W - 1)\,)}\ .$$

Now that we have the efficiency of the WSA architecture, we can get its throughput.

The raw throughput, before efficiency is considered, of the WSA architecture is derived under the assumption that the memory modules can deliver W sites of input data and receive W sites of output data per global machine cycle, or tick. In the LGM-1 machine this is accomplished by reading and writing the memory bus sequentially, one bus cycle for a write of W site values and one for a read of W site values, with buffering on the processor board. From the pipeline side, at the abstract level, it appears that there are two W wide data paths to the memory: one for reading and the other for writing. Since we are assuming the machine is scalable, and we intend to scale it to accommodate larger problems, we assume the memory expands to fit the problem. In this case, it is reasonable to assume there are in fact two memory modules: one receiving the write data, and the other supplying the read data, each with W wide ports. The pipeline performs one shift and update operation for every global tick, and reads and writes W site values. If the memory modules perform at ω_{com} ticks per sec, then the raw throughput of the WSA pipeline is,

$$V_{WSA}^{raw} \quad = \quad (\omega_{com})(\text{ticks/sec})\,W(\text{site-updates/tick/stage})\,s(\text{stages})$$

$$= \quad \omega W s(\text{site-updates/sec})\ .$$

The throughput is then

$$V_{WSA} \quad = \quad \omega s W e\ ,$$

where e is the efficiency derived above. In general, we will assume that the memory technology remains fixed, so that the parameter ω is a constant, and therefore the throughput will be normalized by ω, giving

$$V_{WSA} \quad = \quad sWe\ ,$$

as the normalized throughput for a single WSA pipeline machine.

We have assumed that the main memory is sufficiently large so that a problem of size $l_1 \times l_2$ can be stored. Increasing the problem size implies scaling the memory to fit. Also, as the problem size increases, the throughput of the machine must increase to satisfy the constant time requirement. We will show below that it is reasonable to assume the memory scaling can be done with a fixed technology in such a way that the total number of wires for I/O between the WSA machine and memory increases with linearly total memory size. The machine can also have more than one pipeline. Consequently, the machine's throughput can be increased by scaling any combination of the machine resource parameters: number of stages s, data path width W, pipeline shift-register width w_{sr}, and the number of independent pipes p. Different combinations result in machines which scale with different cost-effectiveness.

3.6. Scaling Resources

Before we present the scaling schemes for MWSA, we should explain why simply adding identical stages to the pipeline does not scale the machine effectively. In this case, the only parameter of the throughput V_{WSA} that can be varied is the number of pipeline stages s: the shift-register width w_{sr} and the data path width W are fixed, and there is only one port to memory (and only one pipeline).

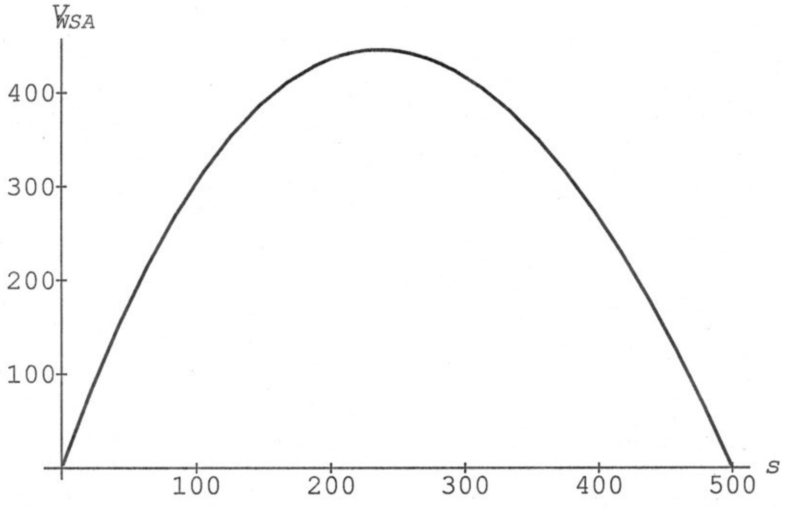


figure 3.6.

WSA throughput as a function of the number of stages in the pipeline (overlap-save method). V is the throughput, s is the number of stages. The lattice is 4000 ($= l_2$) sites deep, the shift-registers are 1000 ($= w_{sr}$) sites wide, and the data path is 4 ($= W$) sites wide. The throughput is normalized to the memory channel clock rate. For a clock rate of 1 MHz, the throughput peaks at almost 1/2 G site-update/second for a pipeline of 250 stages.

As figure 3.6 shows, the throughput of the pipeline falls to zero at about $s = \frac{w_{sr}}{2}$: there is nothing left that hasn't been corrupted by the time the lattice exits the pipe, that is, the efficiency becomes zero. Consequently, some other aspect of the machine must be scaled as well as s, if the throughput is to increase beyond the limit of peak performance for a given w_{sr}.

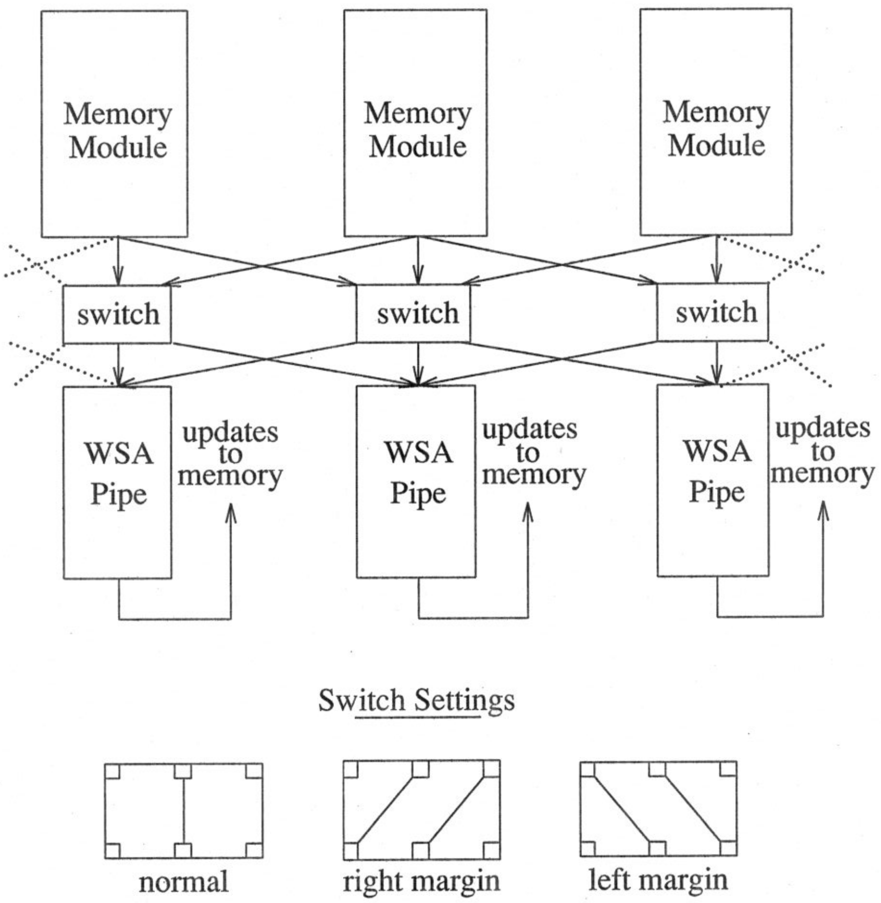


figure 3.7.

Scaling the WSA architecture by adding multiple pipes. Additional memory modules with independent ports and switches connecting modules holding neighboring blocks of the lattice are added as the machine is scaled. The switches allow each independent pipe to receive padding data from neighboring lattice blocks. The overall structure is a ring of pipelines with the left most and rightmost memory modules connected by a switch.

The easiest, but probably not the least expensive, way of scaling up the WSA architecture is to add more memory ports and attaching a separate pipe to each port. See figure 3.7. We assume the memory ports provide their own addressing logic.

Between each port and its associated pipeline input, is a switch. The switch routes the output from the memory either to the pipe, or to one of its neighbors. From the viewpoint of the pipe, in normal operation its input comes from the block of the lattice held in the pipe's memory module. However, when the raster scan reaches the limits of the block and padding from the neighbor block must be used, the input to the pipe is switched to the memory module of its neighbor's pipe. Thus, the left and right limits of the raster scan come from the left and right neighboring modules. Writing does not need to be switched since the padding garbage is ignored and only good site values exiting the pipe belong in the pipe's module. Of course, the pipe is not limited to operating on a single block: the memory module can hold multiple blocks, as long as the neighboring modules hold neighboring blocks. This is accomplished by distributing the blocks to the memory modules modulo the number of modules and connecting the first and last modules to form a ring of memory modules. The effect on the throughput V_{WSA} for the resulting machine is to multiply the previous expression for V_{WSA} by a parameter p, the number of pipelines.

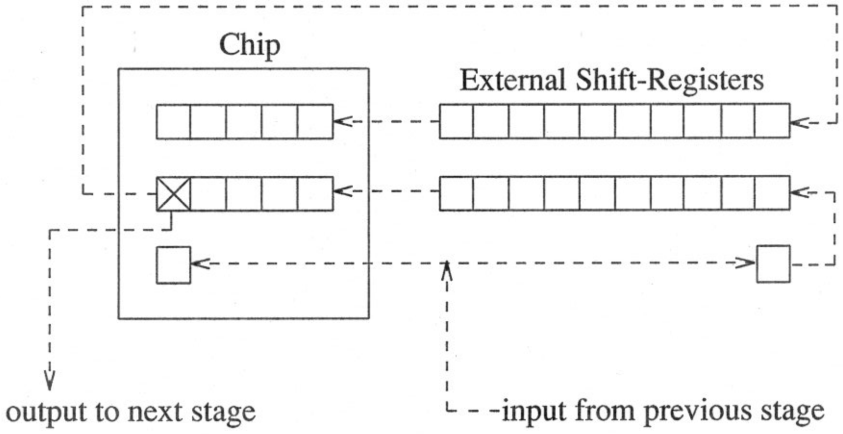


figure 3.8.

Extending the shift-registers off-chip in a single stage of the WSA pipeline. The on-chip shift-registers communicate with the shift-registers off-chip via the paths shown in dashed lines. Each square in the shift-registers represents a data word, W site values. The lower-left single square on the chip would normally communicate with the next shift-register above, but, with the shift-registers extended off-chip, pin usage can be conserved by adding a single word's storage to the external shift-register and copying the input word to that register as well as passing it onto the chip. If the chip has Π *pins available for data (measured in bits per site value), then the maximum data path width is* $\Pi/5$. *When all the shift-registers remain on chip, W can be* $\Pi/2$.

While it is not obvious from the expression for the efficiency, e, that it results in an improvement in throughput, another possibility for scaling the WSA architecture is enlarging the shift-register width, w_{sr}. Generally, up to the limits of the chip area, w_{sr} is made as large as possible since not doing so simply leaves chip area unused. But, it is possible to extend the shift-registers off-chip, and this is probably one of the least expensive expansions available. As figure 3.8 shows, the shift-registers are extended off-chip by sacrificing W since pins must be used to communicate between sections of the shift-register held on-chip, and those held off-chip. As an approximation, if the chip has Π pins available for data, then keeping the shift-registers on-chip allows the data path to be $W = \Pi/2$ sites wide. Moving the shift-registers off-chip allows a data path to be only two-fifths as wide. Balancing this penalty somewhat, is the reward from using denser chip technology for the shift registers, for instance, the off-chip shift-registers can now be DRAMs instead of custom shift-registers.

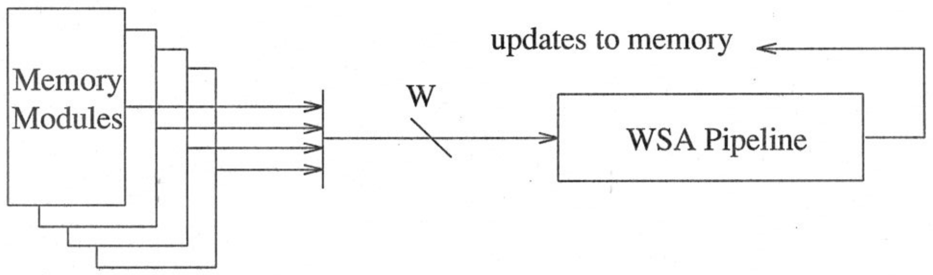


figure 3.9.

Interleaving additional memory modules to form a single memory port with a large data path width W. The i^{th} memory module holds values for sites whose column address is i modulo the number of memory modules.

The last parameter of the WSA architecture that has not yet been scaled is the data path width W. When each stage is contained on a single chip, there is not much opportunity for for adjusting W because the pin limit keeps the data path width strictly limited. However, when scaling the machine, one adds chips and wiring, and there is no inherent reason why one ought to avoid scaling W. The first step in scaling W requires widening the path from the main memory. This is simply accomplished by interleaving additional memory modules to form a wide data path, see figure 3.9. Again, the addressing logic is assumed provided by the memory modules themselves. In this case, no side-to-side communication between modules is necessary: the i^{th} module contains the values for sites that are i^{th}, modulo the number of modules, in raster scan order. Widening the data path in this way can be combined with the previously mentioned expansion in the number of pipes, so that both W and p can be expanded simultaneously.

The next step in scaling *W* is to form the stages of the pipe from slices with fixed data path widths Ω. As figure 3.10 shows, the same modulo decomposition of the raster scan as was used for increasing *W* at the memory module's ports is used to split the *W* wide data stream into separate Ω wide paths in the pipeline. The resulting organization is shown in figure 3.11, and is similar to the Sternberg architecture [40] and consists of identical parallel pipelines with cross communication. (The main difference between the two architectures is the data is fed into the expanded WSA machine in such a way that the individual pipelines update Ω wide strips so that the pattern of sites updated by the several sub-pipes looks like intermingled stripes, while the Sternberg method assigns a fixed-width swath to each sub-pipe.) The shortcoming of this

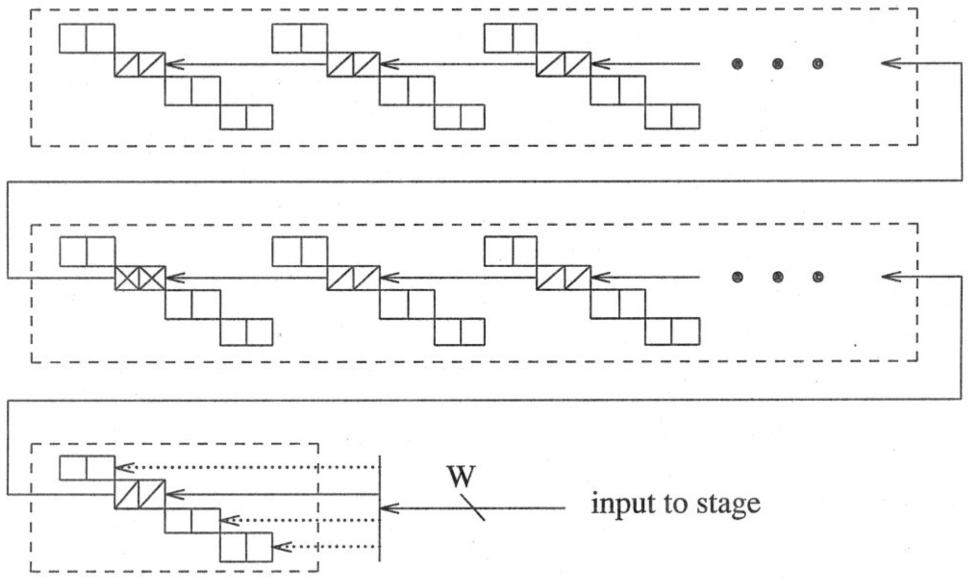


figure 3.10.

Slicing a single WSA stage to accommodate a large data path width W. The logical organization of the shift-registers for a single stage is shown in dashed lines. Each square in the shift-registers represents storage for a single site value. In this figure, there are four physical sub-stages composing a single logical stage. The squares with diagonals represent the shift-registers of a single sub-stage (those with double diagonals represent the sites that get updated), and the communication along the physical shift-register for this sub-stage is shown in solid lines. In this case, the sub-stages can accommodate a data path width of two site values, while the entire stage has a path width of eight site values. If the site values are numbered consecutively starting from zero in raster scan order, then the marked stage receives sites (2,3), (10,11), (18,19), and so on. Each memory module need only communicate with one sub-pipe.

organization is that the side-to-side communication between sub-pipes in the same *W* path requires two-dimensional wiring. Since our cost model will assume wiring cost increases linearly with the number of wires in the machine, and we assume that two-dimensional wiring has a cost function which is at least quadratic in the number of wires, the number of parallel sub-pipes is limited to the number that will fit side-by-side on a single processor board. While this doesn't allow much latitude in adjusting *W*, it at least allows us to consider its effects on cost-effectiveness of the overall machine.

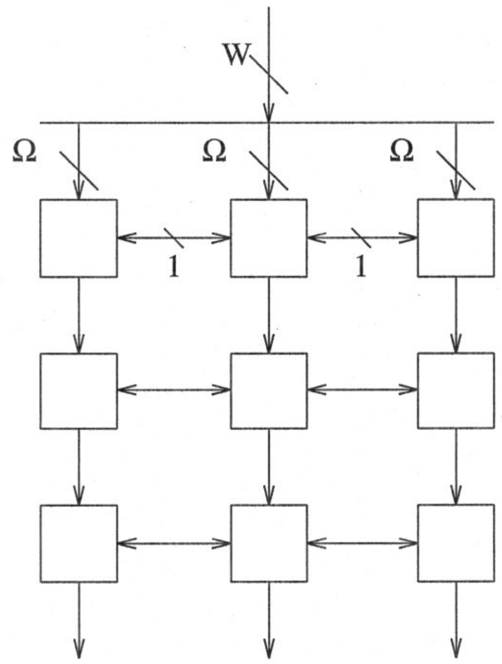


figure 3.11.

The macro-organization of the WSA architecture of a single pipeline with a W wide data path. The individual sub-pipes have data paths of size Ω. Side-to-side communication between the sub-pipes is one site wide. The extreme left and right sub-pipes' side-to-side communication is wrapped around (not shown).

3.7. The Cost and Objective Functions

Optimizing the cost of the design of a scalable machine can be done by optimizing the the cost-effectiveness of the design as the machine is scaled. The resulting curve in the design parameter space defines the cost and throughput at each point along the optimal cost-effectiveness curve. The throughput can then be written as a function of cost, thereby defining the speedup function for the architecture. Thus, the problem is to define the costs for the previously presented scaling schemes, and using cost-effectiveness as an objective function, maximize the objective over the design parameter space variables w_{sr}, s, and W, while scaling the machine. Under the constant-time assumption, the machine's throughput must scale with the size of the lattice, $l_1 \times l_2$, and the number of iterations or global update generations. We will assume the lattice is square and the number of iterations is determined by the number of generations required for a ''sound'' wave to traverse the lattice, which is a constant times the shortest dimension of the lattice, and therefore the design space curve can be parameterized by l_2. In this section we will define the cost of a machine, and state the objective function.

Throughout the following, we use the following conventions. All values are in terms of site datum size (for instance, if the datum of a lattice site is eight bits, and a data path can pass one datum in parallel, we say the path is one wire wide. Properly, this should be called one ''datum-wire,'' which equals eight actual wires, but the meaning should be clear without complicating the wording this way.) We assume there are p independent, parallel pipelines. If the processor chips have Π pins (datum-pins) for communicating site values on and off chip, then Ω is the largest data path the chip can accommodate under the data path expansion scheme shown in the previous section (figure 3.11). Thus, $\Pi = 2\Omega + 4$ when the shift-registers fit on-chip, and $\Pi = 5\Omega + 4$ when the shift-registers are extended off chip.

For a one dimensional homogeneous pipeline where every stage has identical cost, a first order cost estimate given by [41],

$$Cost = \alpha k + \beta ,$$

where α is the cost per stage in the pipeline, and β is a fixed initial cost, is adequate. Expanding on this approach we will assume the cost of a scaled MWSA architecture is linear in each component, and the component costs are linearly related. The cost components we consider are:

(1) memory communication ports at a cost of C_{com} per wire,
(2) shift-register storage area at a cost of C_{area} per site,
(3) processor silicon area at C_{proc} per processor,
(4) downstream (stage-to-stage) pipe wiring at C_{wire} per wire,
(5) W expansion wiring (side-to-side) at C_{wire} per wire.
(6) w_{sr} expansion wiring at C_{wire} per wire.

The costs associated with each of these components for a pipeline with parameters W,

w_{sr}, s, l_2, and p are:

(1) ports for p pipes each W wide,

$$Cost_{com} = C_{com} pW.$$

(2) shift-registers w_{sr} wide for p pipes and s stages,

$$Cost_{sr} = C_{area} ps(2w_{sr} + W).$$

(3) processors, W per stage, s stages, and p pipes,

$$Cost_{proc} = C_{proc} psW.$$

(4) downstream wiring, p pipes, W wide, s stages,

$$Cost_{down} = C_{wire} 2psW.$$

(5) side-to-side W expansion wiring with W/Ω slices per stage,

$$Cost_{side} = C_{wire} 4psW/\Omega.$$

(6) off-chip shift-register wiring,

$$Cost_{sr-wire} = ps3\Omega(W/\Omega).$$

Item (6) can be explained by noting that there are 3Ω wires to the external shift-registers for each stage in a sub-pipe, and there are W/Ω sub-pipes per pipeline of width W. With the assumption of a linear relation between the various costs we can write each of the cost multipliers in terms of a scalar times C_{area}:

$$\begin{aligned} C_{com} &= \alpha_{com} C_{area} \\ C_{proc} &= \alpha_{proc} C_{area} \\ C_{wire} &= \alpha_{wire} C_{area}. \end{aligned}$$

Adding the six cost components, substituting the above expressions for the cost multipliers, factoring out C_{area}, p, s, and W, and normalizing the cost by C_{area}, the resulting cost function for a completely expandable MWSA machine is,

$$C_x = psW(\frac{\alpha_{com}}{s} + \frac{2w_{sr}}{W} + 1 + \alpha_{proc} + \alpha_{wire}(5 + 4/\Omega)) .$$

The cost of the MWSA machine without the expansions of w_{sr} and W is very similar,

$$C_0 = psW(\frac{\alpha_{com}}{s} + \frac{2w_{sr}}{W} + 1 + \alpha_{proc} + 2\alpha_{wire}) .$$

This last cost function can be used when scaling MWSA resources up to the limits imposed by the processor chips on W and w_{sr}. The scaling would thereafter only be allowed by increasing p or s.

Using the cost information for the LGM-1 machine [10] and rules-of-thumb for work-station costs [42] we can get a crude estimate on the proportionality terms in the above cost expressions. The cost of a custom LGM-1 chip with 10^3 shift-register sites is, assuming quantity production, about \$100 with the processors taking up space for about 128 sites worth of storage. The rule-of-thumb estimate gives us a cost of about \$100 per processor board for wiring and glue chips. The communication board is about \$1000. These cost estimates give the
following proportionality constants:

$$\begin{aligned}\alpha_{wire} &= 50\\ \alpha_{com} &= 5000\\ \alpha_{proc} &= 128\end{aligned}$$

The cost of storage on the custom chips is rather high, and if we instead use an estimate based on 1Mbit DRAM chip costs we get the following proportionality constants:

$$\begin{aligned}\alpha_{wire} &= 5\cdot 10^4\\ \alpha_{com} &= 5\cdot 10^6\\ \alpha_{proc} &= 128\end{aligned}$$

Note that the processors are implemented in the same technology as the on-chip storage, and therefore, the cost of processors falls with respect to inter-chip wiring under this cost. Although these proportionality estimates are crude, the general behavior of the cost functions leads to conclusions that can be applied over a plausible range of proportionality values.

Conveniently, because the both the cost functions and the throughput functions are linear in the number of pipelines, p, the objective function for minimizing the cost of a machine at a specific scale turns out to be the cost-effectiveness, V_{MWSA}/C, where C is either C_0 or C_x. To see this, write V_{MWSA} as the product of the number of independent pipes p and a function of the remaining design variables s, W, and w_{sr},

$$V_{MWSA} \quad = \quad psWe \quad = \quad pg(s,\, W,\, w_{sr}) \ ,$$

where e is the efficiency, $e = e(s,\, W,\, w_{sr})$. The cost function can be written similarly,

$$C \quad = \quad pf(s,\, W,\, w_{sr}) \ ,$$

where C is either C_x or C_0. As was mentioned earlier, because of the constant time assumption, the lattice size l_2 and the throughput V_{MWSA} are related by the ''tolerable delay'' time constant. So, scaling the lattice to some size determined by l_2, scales the throughput to some value V. Setting V_{MWSA} to V and solving for p gives,

$$p = \frac{V}{g} ,$$

and substituting this for p in the expression for cost gives,

$$C = V\frac{f}{g} .$$

Minimizing the cost C requires maximizing f/g over the variables s, W, and w_{sr}. This is equivalent to maximizing the cost-effectiveness CE over these same variables since,

$$\begin{aligned} CE &= \frac{V_{MWSA}}{C} \\ &= \frac{g}{f} . \end{aligned}$$

For each l_2 value, minimizing the cost-effectiveness CE over the variables s, W, and w_{sr} gives the least-cost MWSA machine for that size lattice.

3.8. Analysis

We will numerically determine the most cost-effective configuration of resources for the fully expandable scheme whose cost function was shown in the previous section, C_x. That is, we will maximize

$$CE = \frac{g}{f}$$

over the variables w_{sr}, s, and W, for varying lattice size l_2, where

$$g = sW\frac{l_2(w_{sr} - 2s)}{l_2 w_{sr} + s(2w_{sr} + W - 1)} ,$$

and

$$f = sW(\frac{\alpha_{com}}{s} + 2\frac{w_{sr}}{W} + 1 + \alpha_{proc} + \alpha_{wire}(5 + 4/\Omega) .$$

The value of CE will be in units of the cost-effectiveness for a single stage, one word, minimum size machine (essentially a uniprocessor) for an infinite lattice, that is, $CE(w_{sr}, s, W, l_2)$ is normalized by $CE(3, 1, 1, \infty)$.

Since W is the most restricted variable, we begin by exploring its effect on the cost-effectiveness CE. We will use a fixed lattice size $l_2 = 10^4$ which shows the typical behavior of CE for a wide range of l_2 values. The general shape of the CE function

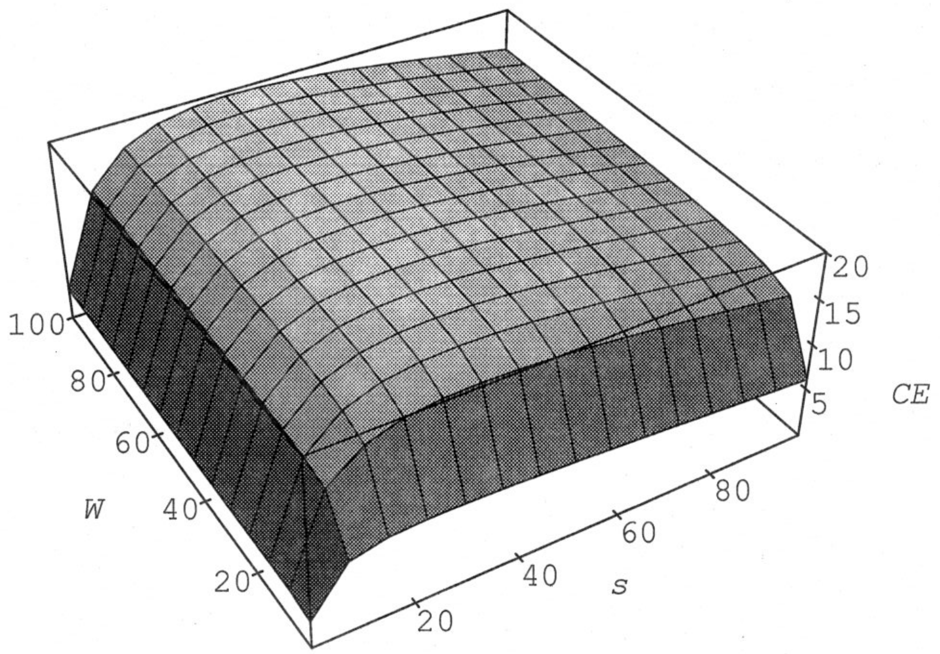


figure 3.12.

Cost-effectiveness CE as a function of s and W. CE is normalized by its value at $(w_{sr}, s, W, l_2) = (3, 1, 1, \infty)$.

for w_{sr} fixed at 500 in shown in figure 3.12. The effect of W there is mostly negligible for W larger than 20. However, although it is not obvious in figure 3.12, a slice through the surface parallel to the s-axis shows that an approximate doubling of CE takes place for W increasing from 1 to about 10.

Fixing s instead of w_{sr} results in the behavior shown in figure 3.13. Here, the effect of W depends on w_{sr}: as w_{sr} increases, the dependence of CE on W becomes flatter, suggesting that W may be almost linearly related to CE in the range of optimal values for s and w_{sr}.

Maximizing CE over (w_{sr}, s) for twelve values of W results in the curves shown in figure 3.14. Here, a steepest decent search method in the (w_{sr}, s) space was used for the values of $W = \{ 1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000 \}$. (Note that the larger numbers for W are not properly modelled in the cost function. However, using them allows us to see the general behavior of the dependence on W.) The resulting CE value is plotted as a function of $\log_{10} W$ for the three cases $l_2 = 10^3, 10^4, 10^6$ (dashed, solid, and dotted lines, respectively). The lowest value on the curve has a CE value of about 11. Generally, the curves show slow monotonic

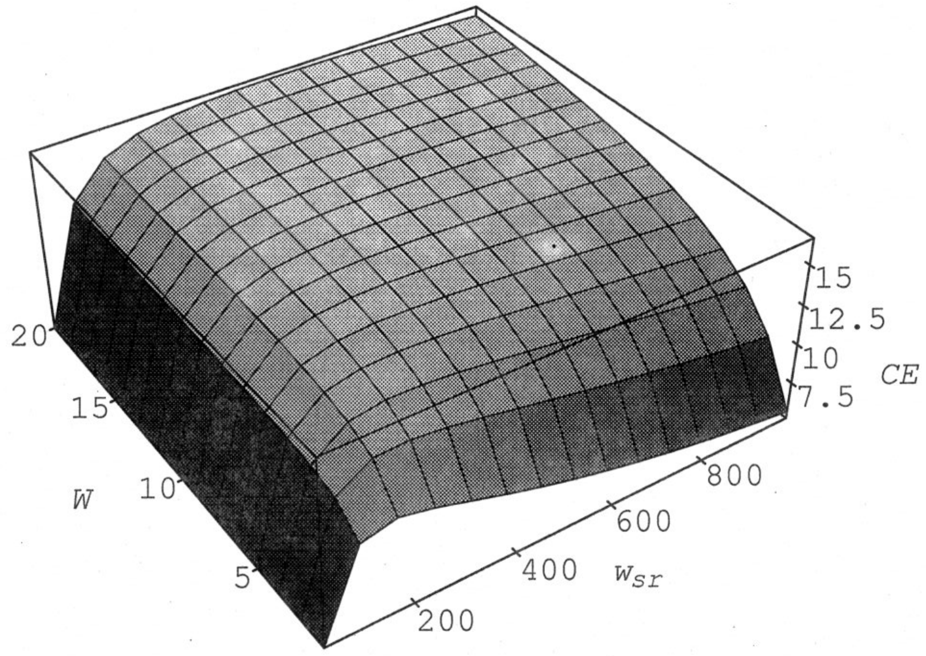


figure 3.13.

Cost-effectiveness as a function of w_{sr} and W. Here $s = 15$.

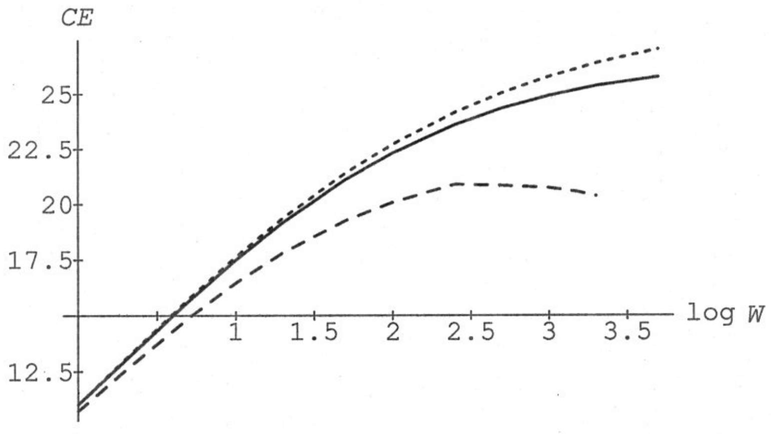


figure 3.14.

Maximum cost-effectiveness as a function of $\log_{10} W$ for three values of l_2. The dotted line has $l_2 = 10^6$, the solid line has $l_2 = 10^4$, and the dashed line has $l_2 = 10^3$.

increases in *CE* as *W* gets larger. However, the curve for $l_2 = 10^3$ shows a peak and thereafter decreasing *CE*. This behavior is the result of the optimal w_{sr} value reaching its limit of $2l_1$. Generally, the optimal values for w_{sr} and s increase with increasing *W*, as is shown in figure 3.15 for the case $l_2 = 10^4$.

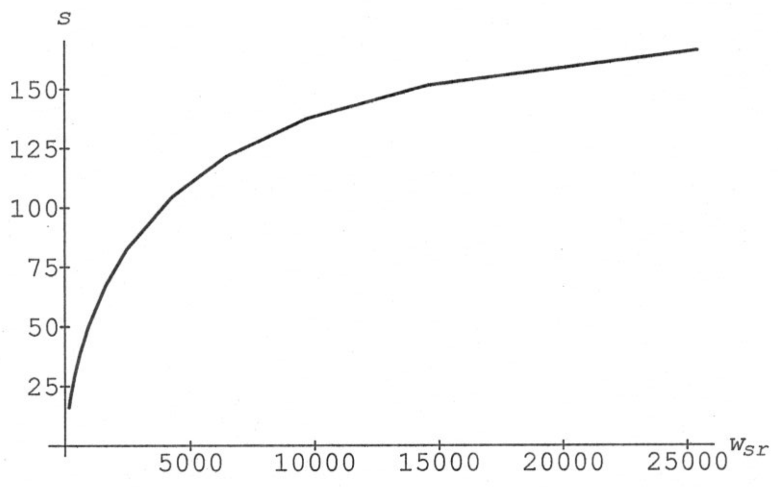


figure 3.15.

The optimal w_{sr} and s combinations plotted as a curve parametrized by W. W increases as the curve moves away from the origin. Lattice dimension l_2 is fixed at 10^4.

The parameter w_{sr} cannot increase indefinitely for fixed l_1, and we have assumed $l_1 = l_2$ in figure 3.14. The other two curves in that figure would show similar peaks, if the range of *W* was extended far enough, and the definition of l_1 remained the same.

The behavior of maximum cost-effectiveness is more strongly dependent on *W* when *W* is small. Figure 3.16 is a plot of the same curves as in figure 3.14, except that here *CE* is plotted as a function of *W* over the range 1 to 100. As this figure shows, most of the increase in *CE* occurs when *W* is between 1 and 100, and *CE* approximately doubles for all curves as *W* covers this range. The conclusion is then that, while it is always more cost-effective to increase *W*, increasing *W* from 1 to some value less than 100 achieves most of the benefit. Since the cost function C_x is only intended to model the cost of increasing *W* when two-dimensional inter-board wiring is not required, the limit on *W* is small, and it is apparently advantageous to set *W* to its upper limit.

A reasonable upper limit on *W*, in the case of the chips and boards used in the LGM-1 machine, is about 10. The reason is that the chips have an Ω of 1 under the present scaling scheme (the chips can accommodate 1 word wide data paths) because of pin constraints, and therefore each stage would require ten chips. Since each board

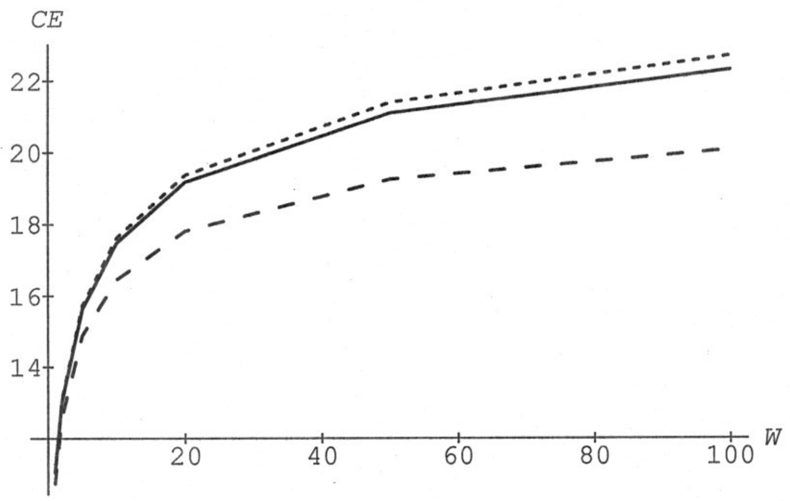


figure 3.16.

Maximum cost-effectiveness as a function of W for three values of l_2. The dotted line has $l_2 = 10^6$, the solid line has $l_2 = 10^4$, and the dashed line has $l_2 = 10^3$.

can only accommodate a maximum of ten chips, widening the data path further would require two-dimensional wiring over board boundaries, which is not accounted for by the present cost function. Although the LGM-1 was built with what is now old technology and the above estimate is rather restrictive by today's standards, the general results hold, with appropriate scaling, for newer technology. With improved packaging, more pins per chip, and larger boards, *W* could probably be extended to at least 20 in the LGM-1 system. However, noting that for the $l_2 = 10^3$ curve, setting *W* to 10 gives about 50% of the maximum possible increase in maximum *CE*, we will fix *W* at this value for the remainder of the discussion.

Having fixed *W*, the next step is to determine the optimal sizes for s and w_{sr} as the problem and the machine are scaled. Figure 3.17 shows a typical *CE* surface as a function of w_{sr} and s for a fixed lattice size, $l_2 = 10^4$ in that figure, and fixed data path width, $W = 10$. A unique peak in the surface makes it possible to define the optimal values of w_{sr} and s as functions of l_2 by using a steepest descent search over the (w_{sr}, s) space for a series of l_2 values. We used nine values of l_2 nearly equally spaced along a $\log_{10}$ scale: $l_2 = 3^i$ for $i = 4, 5, 6, \cdots, 12$, that is, l_2 runs from about 100 to about $0.5\ 10^6$.

Figure 3.18 shows a point plot of the (l_2, s) pairs resulting from the gradient search mentioned above. Also show there is the curve of the best-fit $e^{\log_{10}(l_2)}$ approximation. The approximation curve's equation is,

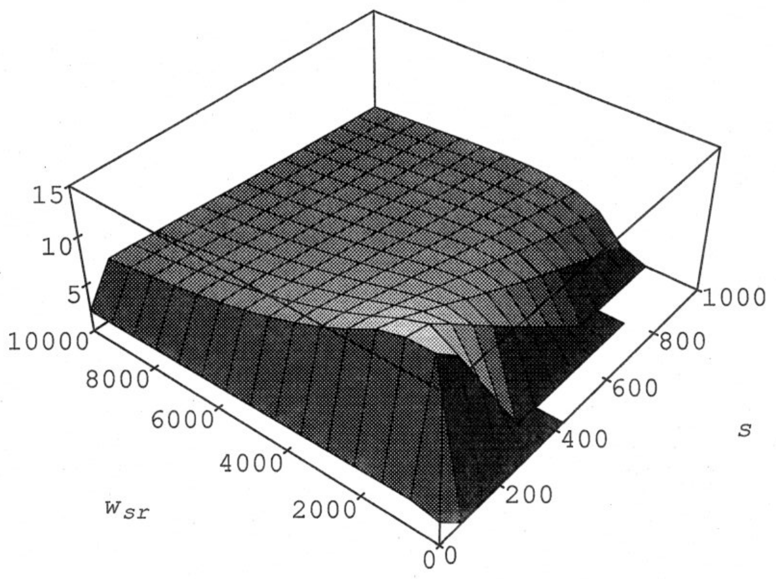


figure 3.17.

Cost-effectiveness as a function of w_{sr} and s for W fixed at 10 and l_2 fixed at 10^4.

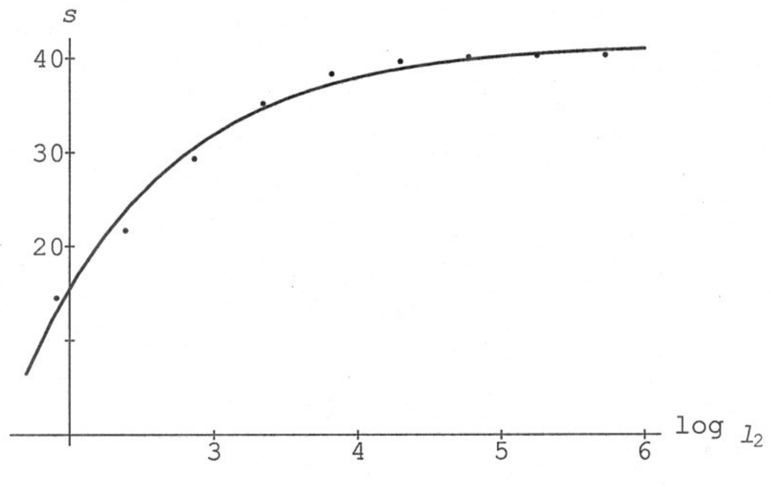


figure 3.18.

The optimal s values as a function of $\log_{10}(l_2)$ *for the nine values of* l_2 *used in the gradient search. The solid curve is an approximate fit to the points.*

$$s(l_2) \quad = \quad 41.4852 \; - \; \frac{191.621}{l_2^{0.434294}} \; .$$

This curve gives reasonably accurate values for s, although s is actually restricted to positive integer values.

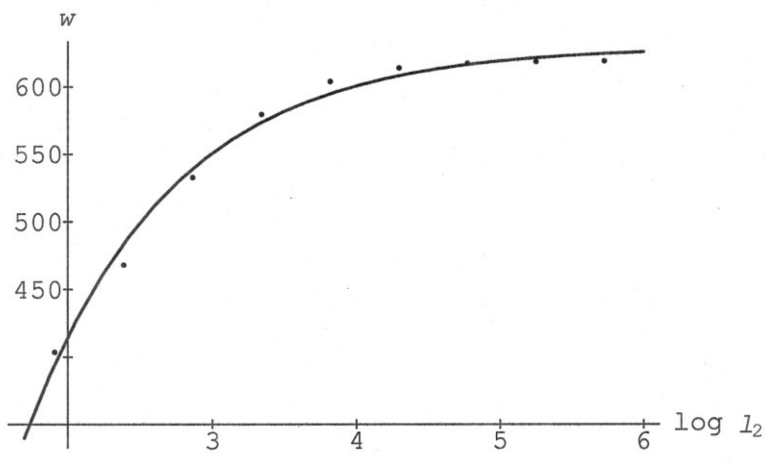


figure 3.19.

The optimal w values as a function of $\log_{10}(l_2)$ *for the nine values of* l_2 *used in the gradient search. The solid curve is an approximate fit to the points.*

Figure 3.19 shows the results for the optimal w_{sr} values. The approximate fit to the data points is given by,

$$w_{sr}(l_2) = 630.011 - \frac{1588.45}{l_2^{0.434294}} .$$

The flattening of the optimal w_{sr} and optimal s curves is a consequence of *CE*'s decreasing sensitivity to l_2 as l_2 becomes large. Cost-effectiveness *CE* is shown as a function of $\log_{10}(l_2)$ in figure 3.20. As this figure shows, and as can be seen from the expression for *CE*, *CE* is independent of l_2 in the limit as l_2 goes to infinity:

$$\lim_{l_2 \to \infty} CE = \frac{(w_{sr} - 2s)}{w_{sr}(\frac{\alpha_{com}}{s} + 2\frac{w_{sr}}{W} + 1 + \alpha_{proc} + \alpha_{wire}(5 + 4/\Omega))} .$$

Consequently, optimizing *CE* fixes w_{sr} and s when l_2 is large. However, for smaller values of l_2, *CE* increases with problem size, and the optimal values for w_{sr} and s increase as the previous figures show.

The remaining parameter of the machine is the number of independent pipelines, p. Under the constant time assumption, p is a function of the total amount of work done, $work = l_1 \times l_2 \times n$ where n is the number of iterations. Given *work*, the required throughput is

$$V = \frac{work}{t_0} ,$$

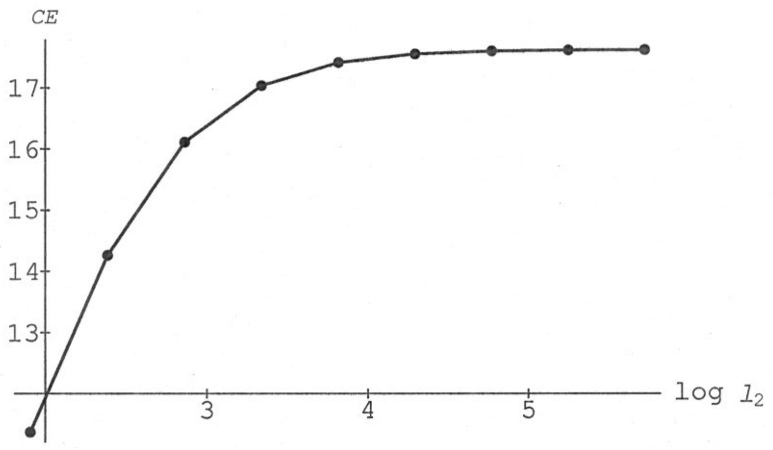


figure 3.20.

Optimal cost-effectiveness CE as a function of $\log_{10}(l_2)$ *plotted at the same data points as were used for the optimal* w_{sr} *and s plots.*

where t_0 is the fixed amount of time allowed for the computation. Setting the un-normalized expression for V_{wsa} equal to the right-hand side of the above, and solving for p gives,

$$p \quad = \quad \frac{work/t_0}{\omega sWe} \;,$$

where ω is the cycle rate of the main memory, and e is the WSA efficiency and is a function of the variables $(w_{sr},\ s,\ W,\ l_2)$. From the previous discussion we have fixed W at 10, and the optimal values of w_{sr} and s are functions of l_2. Also, *work* is a function of l_1, l_2, and n. The variables in the expression for p are then ω, t_0, l_1, l_2, and n. We will make a series of assumptions and reduce p to a function of a single variable.

Generally, the dimensions of the lattice, l_1 and l_2 are not equal. For instance, flow simulations may need the channel length, say l_2, to be several times the channel width, in this case l_1. However, this asymmetry is usually not more than a factor of 4, and we will therefore assume the lattice is square: $l_1 = l_2 = l$. The number of iterations required is not as easy a number to fix because it depends on the nature of the study in which the simulation is being used. For instance, to follow moving, large-scale, turbulence patterns in a flow may require the number of iterations be large enough to allow such a feature to traverse the entire channel. Since its velocity may be only a fraction of the sound velocity in the lattice ($1/\sqrt{2}$ for the hexagonal lattice), the number of iterations can be on the order of 10^3 times l. On the other hand, the number of iterations required for a particle to cross the entire lattice without collisions is only l. As a matter of convenience we will take the number of iterations to be l, and the problem size is then l^3.

Now the fixed time is, as we said at the outset, a rather loose concept. However, for medium-to-large simulations, a reasonable turnaround time is about 1.5 hours. So, we set $t_0 = 5\times10^3$ seconds. Finally, we set the main memory cycle time to $\omega = 10$ MHz.

So, with the above parameters set accordingly,

$$\begin{aligned} l_1 &= l \\ l_2 &= l \\ n &= l \\ work &= l^3 \\ W &= 10 \\ t_0 &= 5\cdot10^3 \\ \omega &= 10^7 \end{aligned}$$

and w_{sr} and s set to their limiting values, the expression for p becomes approximately,

$$p(l^3) \quad = \quad 2\cdot10^{-13}\, l^3 \ . \tag{1}$$

More precisely since p is at least 1, the ceiling function should be applied to the right side. Actually, the fit to a log-log plot of p versus l^3 gives a slightly less than linear dependence of p on l^3, as is expected since w_{sr} and s are not fixed.

Using the expression for p in (1), we can determine the throughput and cost of the optimal machine for each value of l_2. Figure 3.21 shows the nine resulting cost-throughput points in a log-log plot, and a linear fit to the points. The fitted curve shows a slightly super-linear relationship between throughput V_{MWSA} and C_x:

$$V_{MWSA} \quad = \quad k \ + \ \frac{1}{1316}\,(C_x)^{1.01941} \ ,$$

where k is an undetermined constant. As we mentioned previously, as l_2 goes to infinity, w_{sr} and s become fixed, and, consequently, each independent pipeline increases the throughput and the cost of the machine by fixed increments. This implies that the MWSA architecture has linear speed-up asymptotically. However, there are two problems with this conclusion: there is a further constraining relationship between the parameters p, w_{sr}, and s. This comes from noting that the total number of blocks that the lattice is partitioned into times the width of the blocks cannot exceed the width of the lattice, l_1. Since each independent pipe must be associated with at least one

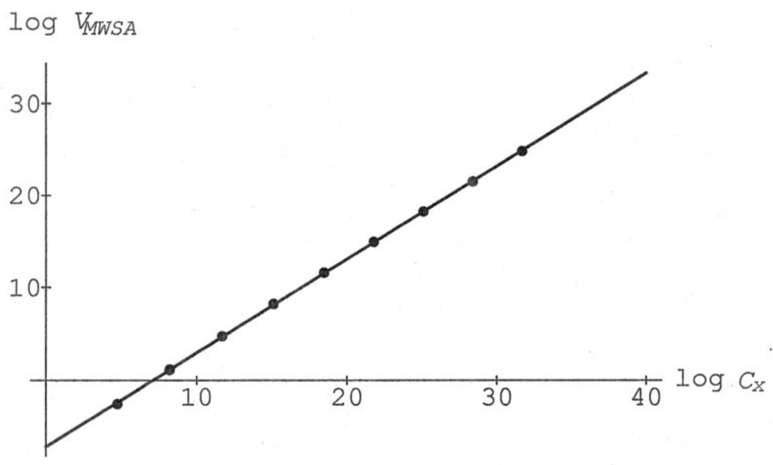


figure 3.21.

MWSA throughput plotted as a function of cost using the optimal values of w_{sr}, *s, and p found by maximizing cost-effectiveness. The scales for x and y axes are the natural logarithms of* C_x *and* V_{MWSA}, *respectively. The value used for p comes from equation (1).*

block, and the width of a block is $(w_{sr} - 2s)$, we have the following constraint:

$$p\,(w_{sr} - 2s) \quad \leq \quad l_1 \;. \tag{2}$$

As a consequence of this constraint, the ninth point (corresponding to $l = 531441$) in plot 57 is actually infeasible, and the machine cannot be scaled in this manner for larger values of l. Scaling the machine optimally beyond the point where (2) makes the current scaling scheme infeasible, requires a different optimization procedure, which we have not attempted. However, we can say one thing about the asymptotic behavior: under the constant-time constraint, the required throughput cannot be achieved. This follows from the fact that the maximum parallelism possible is $l_1 \times l_2$, or l^2 in the present case. Consequently, the total time for the computation is at least some constant (dependent on the clock speed) times l, which is certainly not constant unless one proposes ever increasing clock speeds, that is, $\omega \rightarrow \infty$ as $l \rightarrow \infty$. Thus, asymptotically, in order for linear speed-up to hold, clock speed would have to increase linearly with cost.

Finally, we take a look at the optimal configurations of the MWSA machine for l in the range where the current optimizing method yields feasible machines. Constraint (1) also makes the first point (corresponding to $l = 81$) in plot 57 infeasible. The value for p given by (1) is less than one in this region. In fact, p is less than one for the first five points in figure 3.21. As a consequence, the actual value for p is 1 for all these values of l, and under the current optimization method, all machines are essentially identical up to the point where $l = 19683$. Again, a different optimization method would have to be applied to determine the optimal machine configurations for l below this value. The optimal machine configurations are shown in table 3 for the three values of $l = 19683$, 59049, and 117147. This table shows the optimal combinations of w_{sr}, s, and p for these three l values used in the optimization. At the bottom of the table is shown the cost of the machine (in millions of dollars), according to the un-normalized cost function C_x, and the throughput (in Giga updates/second).

Optimal MWSA Machine Configurations			
l	*19683*	*59049*	*177147*
p	*1*	*12*	*318*
w_{sr}	*614*	*618*	*619*
s	*40*	*41*	*41*
cost	*0.033*	*0.392*	*10*
V	*3.4*	*41*	10^3

table 3.

The feasible configurations shown in table 3 represent a ''window'' of feasibility for the least-cost configurations found using this optimization method. When costs are fixed, the placement of this window along the l line is determined by the parameters ω and t_0: decreasing either moves the window to smaller l values, increasing either has the opposite effect, although as we assumed throughout, t_0 is essentially fixed and cannot therefore be adjusted. Outside this window, the cost-effectiveness decreases as the machine parameters are forced away from their optimal values by the constraints. Consequently, the configurations within the window give a good representation of the least cost that can be attained.

3.9. Summary, Conclusions, and Future Work

We have presented a scalable architecture, MWSA, based on multiple, independent WSA-architecture pipelines for two-dimensional cellular automata. The scaling scheme for this architecture avoids wiring a two-dimensional wiring pattern across circuit boards which justifies our definition of the cost function for the machine as a linear function of its size. Avoiding two-dimensional wiring is accomplished in the MWSA architecture by using an overlap-save strategy to allow the computation of independent blocks of the lattice without cross communication between neighboring pipes. We have derived the efficiency and throughput of the MWSA architecture, and we have derived its cost function by employing known costs from previous machines and rules-of-thumb in workstation construction. We have introduced a definition of throughput speed-up based on the least total machine cost rather than the number of logical processing units, where speed-up is taken to be the relative increase in throughput as the machine resources and problem size are increased. Using a numerical gradient search method to find the least-cost configuration of MWSA machine resources for a given size lattice problem under the constant time constraint, we have shown that the MWSA scalable architecture has slightly superlinear speed-up, as a function of cost, over a moderately large range of lattice sizes. We have shown that the optimal-cost configurations have essentially fixed, short (circa 40 stages) pipelines, with moderately wide local shift-registers (circa 600 lattice sites), and narrow data paths (circa 10 lattice sites wide), for the range of lattices with 10^4 to $2 \cdot 10^5$ lattices sites on an edge.

As the previous chapters have emphasized, the bandwidth to main memory is critically important in the cost of the machine. In this chapter, the bandwidth has been scaled by adding multiple ports. However, the total machine cost is linearly dependent on the memory cycle rate, ω. Thus, although the previous chapters suggested total bandwidth is the important quantity, this chapter stresses the importance of the speed of the communication channel. A special-purpose processor for scientific computation has to compete favorably with general-purpose machines to be cost-effective. The conclusion suggested by this work is that the economy of VLSI cannot alleviate the need for cheap, fast communication in special-purpose machines.

Future Work

The cost function we used was based on relatively high cost per shift-register cell. The above analysis can be redone using a cost more in line with current costs of 1Mbit DRAM's. We have started this and found that, in this case, the asymptotic values for s and w_{sr} are about 200 and 10^4 respectively. This causes the p limit mentioned above to be reached much sooner, and consequently, incorporating this constraint in the optimization becomes more important. It would also be interesting to see what effect this new cost has on the overall machine cost as compared with the above results.

In general, it would be interesting to find the optimal configurations outside the p limit window. This would allow finding the complete speedup curve over l. Then, using the non-expansion cost function C_0, and finding the complete speedup and cost functions for this scaling scheme, the different scaling schemes could be compared in total machine cost. More ambitious work along this line would be the task of comparing the cost of these MWSA scaling schemes with other commercial and special-purpose machines. This would address the general question of whether special-purpose machines are cost-effective and the applications in which they can be employed effectively.

The advancing developments in communication bandwidth using optical technology suggests two areas for further work. One is to look at the consequences of improved bandwidth across chip boundaries (increasing Ω). The other area has to do with the decreasing cost and increasing bandwidth of inter-board wiring. These technology improvements will significantly effect the above analysis, and make much larger W values practical. As well, they may allow less chip area be devoted to storage and thereby allow more processors per chip.

Finally, it would be interesting to include the main memory cost in the analysis. Two questions should be addressed here: whether or not it is possible to attain linear speedup with fixed-access-time memory, and what the optimal organization might be.

Chapter 4
Testing Parallel Lattice-Gas Simulators

4.1. Introduction

Since Frisch, Hasslacher, and Pomeau [43, 44] introduced the use of lattice-gas automata to simulate hydrodynamics, their FHP models [3] and variants have been used in many simulation studies. Some simulations have used commercial supercomputers or parallel processors (see references [45-54], for example) and others have used special purpose hardware [8, 55, 10]. Because it is not yet known how well the lattice-gas automata model physical systems, there has been interest in comparing lattice-gas simulations with theoretical and experimental results. The validity of such comparisons depends on the correctness of the implementation. (The situation is illustrated in figure 4.1.)

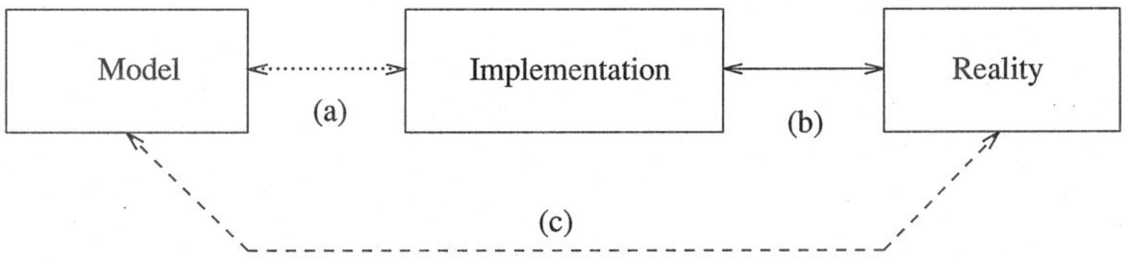


figure 4.1.

The solid line (b) shows the comparison we really make when we compare a computer model of a system to experimental results for that system. We would like to say that the comparison we are making is effectively between the model and the physical experiment (dashed line (c)). The dotted line (a) suggests the missing piece of information that would allow us to say this with conviction: the knowledge of the correctness of the implementation.

It is not usually possible to establish independently the correctness of an implementation because of the complexity of the operations used in the implementation. For instance, the complexity of floating point arithmetic makes verifying the correctness of an implementation of a finite-difference scheme for integrating the Navier-Stokes equations impossible in practice. The state of affairs for lattice-gas simulations is quite different: the data movement and logic operations are simple. It is therefore possible and practical for the correctness of an implementation of a lattice-gas automaton to be tested exhaustively before runtime and monitored during runtime.

We became interested in verifying the functional correctness of a lattice-gas simulator while running fluid flow simulations on a custom VLSI processor, LGM-1 [10], built here as an experimental prototype. The particular simulation project we undertook involved comparing our simulation results for a specific flow problem with the results from other methods for the same problem. In making these comparisons we discovered that it was impossible to determine whether the discrepancies we saw were caused by the differences between the methods, or by artifacts of incorrect implementation of our system. Furthermore, interspersed with simulations we were also modifying both the hardware and software of the system, requiring a concrete testing method for debugging purposes. From this experience we realized that a complete system testing method was needed which could be run independently of any simulations to verify functional correctness of the simulator system.

Our experience with simulations on LGM-1 also convinced us that system functional testing was not sufficient: we also needed runtime fault detection. We often ran simulations continuously for 24, 36, and more hours, and discovered that, aside from the errors caused by incorrect implementation of the algorithm, there were other sources of error of a more transient nature. For instance, we found that during long runs the host system or the network facility could cause errors, even though the system did not crash and the simulation ran to completion. Similarly, temporary failure of custom chips, pin connections, and so forth, could occur during a simulation, and not be detectable either before the run or after its completion. Although we realized that detecting every transient error during simulation was probably not possible, we guessed that the most likely kind of transient error was not the random single bit error, but failures that would effect large pieces of the simulation, large either in time or space. We therefore began to look for ways of embedding runtime fault detection in the initial state of the simulated automaton.

In this chapter we describe a testing method for lattice-gas simulators. The method can be categorized as specification-based system-level functional testing [56] because we use the specification of the behavior of a lattice gas to derive input data that tests the correct functional operation of a simulator system. This means we do not do any modelling of hardware faults although the test method is used to test custom VLSI chips of a special-purpose hardware simulator. Our approach consists of using graphic display of the lattice-gas state to detect errors in the evolution of cyclic sub-lattices. The collection of sub-lattices exhaustively exercises the update logic of the simulator, and is built from a small library of hand-coded test pattern templates. The test system consists of the template library, a small function library for creating patterns in a lattice-gas state, and a library of routines for image manipulation and display. In practice, we have used the test method in a ''prevention oriented'' [57] manner in the construction of the testing facilities themselves. That is, during construction of the template library, image manipulation, display software, and a simulator for the special-purpose hardware, the test patterns were used to prevent, detect, and debug implementation errors. Using a ''destruction oriented'' approach we have used the test method to test custom VLSI chips, custom circuit boards, and general control software of the special-purpose system. Finally, we have used the testing facility to embed test patterns in the input data of lattice-gas simulations used in fluid flow experiments to indicate simulator system functional runtime errors.

The remainder of the chapter is organized as follows. Section 2 briefly describes the FHP-III and LGM-1 lattice-gas models. Section 3 introduces some terminology and definitions. Section 4 gives a general description of the test ensemble and some description of the methods used in its construction. Section 5 gives a detailed description of a single test pattern template. Section 6 describes the construction of specific collections of test patterns that constitute the test ensemble. Section 7 discusses the error detection and experimental results for multiple error coverage. Section 8 discusses the issue of applying the test method to different architectures. Section 9 contains a summary of our experiences using the test method, and our conclusions.

4.1.1. Lattice-gas automata

Our version of a lattice gas is based on the FHP-III model. As described in [3] this type of lattice-gas automaton consists of a two-dimensional lattice graph [1] and a set of update rules for variables associated with each node in the lattice graph. The lattice is the triangular lattice on the plane generated by the unit vectors $\mathbf{e}_1 = (1, 0)$ and $\mathbf{e}_2 = (1, \pi/3)$, in polar coordinates. The edges of the graph connect nearest neighbors in the lattice (see figure 4.2). In cellular-automata terms [6, 5], each site together with its variables constitutes a cell of the automaton; the edges define the cell's neighborhood. Each cell has eight bits of state information: seven one-bit dynamical variables and one bit defining the type of site. In the lattice-gas view, at each site each incident edge has an associated variable representing the presence or absence of a unit mass particle with unit velocity directed toward the site's neighbor along that edge (see figure 4.3). The seventh dynamical variable encodes the presence or absence of a unit mass particle with zero velocity positioned at the lattice site (called a ''rest-particle''). The eighth bit encodes the presence of a barrier at the lattice site. With this interpretation of the variables the update rule is designed so that, letting the

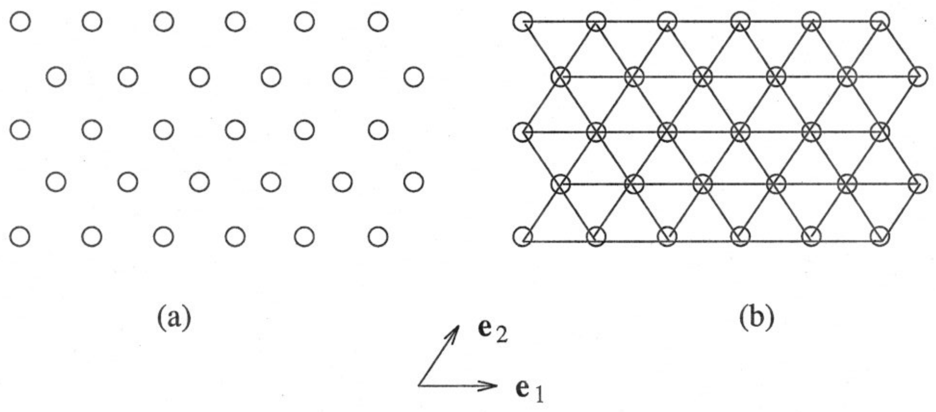


figure 4.2.

(a) A finite lattice generated by the unit vectors $\mathbf{e}_1$ *and* $\mathbf{e}_2$*. (b) The lattice-graph produced from (a) by nearest neighbor connections.*

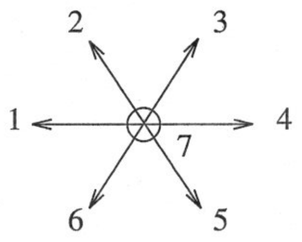


figure 4.3.

The seven velocity vectors at a lattice site. The seventh is represented by a circle and has zero magnitude.

edges have unit length, the lattice is populated with particles traveling along graph edges and colliding at lattice sites (see figure 4.4).

A lattice-gas automaton evolves by synchronously updating the state of every cell of the automaton: the next state of a cell is determined by the states of its neighbors and its own state, and the automaton's update rule table. For the lattice gasses the automaton update rule table is called a *collision rule set* , the initial configuration of states determining a cell's new state is called a *collision,* the next state entry in the rule table is called the *result* of the collision, and the combination of a collision and its result is called a *collision rule*. A particular lattice gas is defined by specifying a collision rule set that gives the results of every possible collision. These rules can be thought of as

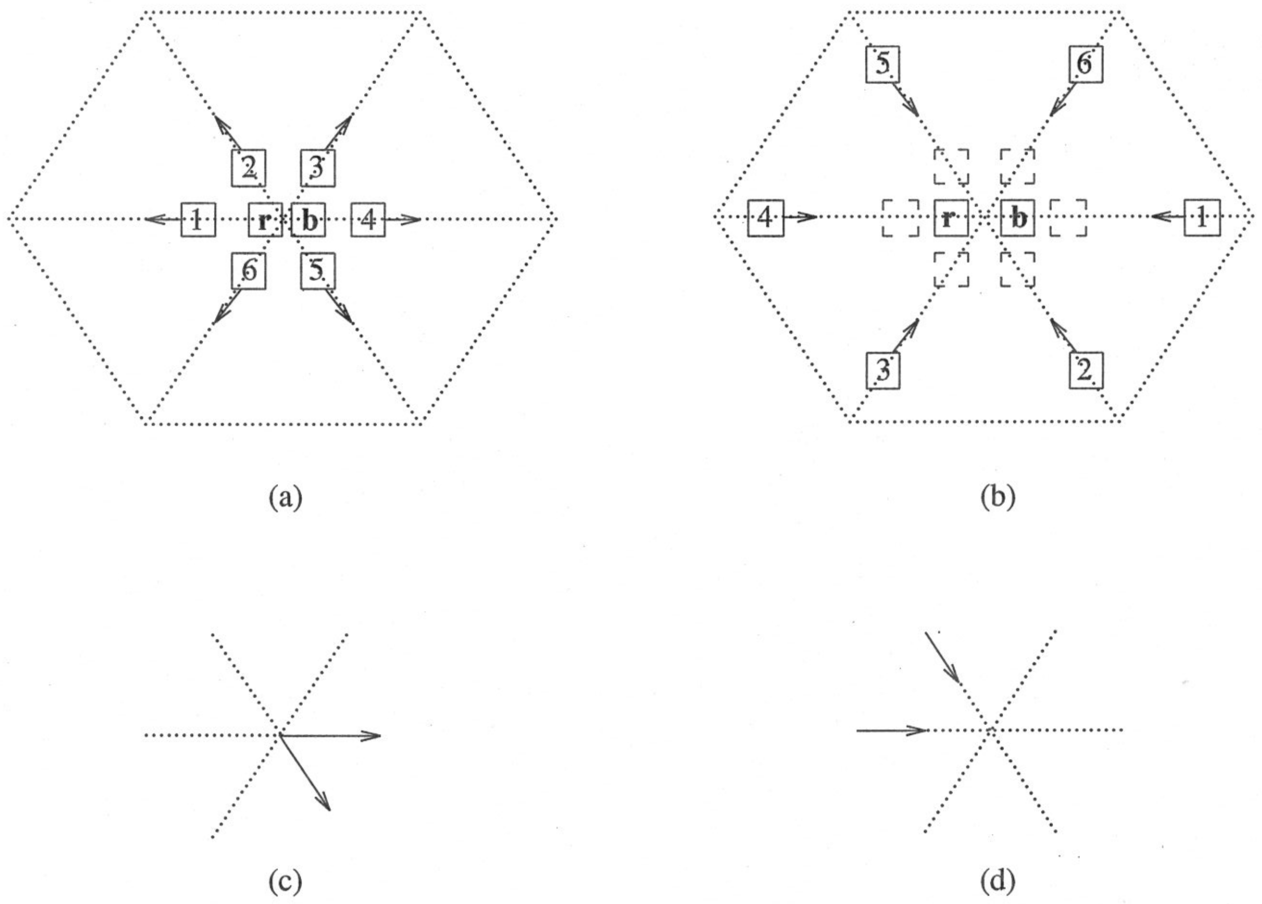


figure 4.4.

(a) The state of a lattice site and associated one-bit variables. **r** *represents a rest-particle,* **b** *a barrier site. (b) The input variables that affect the next state of the lattice site. (c) Shorthand notation for the state of a lattice site. (d) Shorthand notation for the input to the next-state computation. The combination (d) — (c) represents an update rule: (c) is the next state of the lattice site after an input of (d).*

rules about the action of particles (variables set to one) or equivalently as rules about the action of holes (variables set to zero), as they collide at lattice sites. In an FHP gas, particle collisions generally conserve momentum and mass, and are symmetric with respect to rotation by integer multiples of $\pi/3$, and time reversal; and for the FHP-III gas, collisions are also symmetric with respect to hole/particle duality (complementation of the dynamical variables).

These symmetry properties and hole/particle duality make it possible to define a collision rule set in a compact way: for each equivalence class of collisions induced by equivalence under duality and rotation transformations, pick one example, called a *canonical collision,* and show its next-state result; the combination of canonical collision and result is called a *canonical collision rule.* There are 28 canonical collisions for the eight bit FHP-III and LGM-1 lattice gasses, and figure 4.5 shows the fourteen ''non-barrier'' canonical collision rules for these two automata. Using the list of fourteen canonical collisions, c_0 through c_{13}, all 256 possible collisions for the LGM or FHP-III gasses can be generated by applying the symmetry transformations plus barrier presence or absence. When the barriers implement the ''non-slip boundary condition'', the remaining fourteen rules can be stated in a single sentence: all particles colliding at a barrier site reverse their direction of travel, and the last row of figure 4.5 shows a generic example.

The second column of figure 4.5 shows the results of the canonical collisions for the FHP-III gas. Where there are two entries for a single canonical collision we mean that the two possible results occur equally often determined by some explicit rule. LGM-1 implements this by using one rule on even rows and the other rule on odd rows.

The third column of figure 4.5 defines the LGM-1 gas. Only the results that differ from FHP-III are shown. The c_3 and c_5 results are different for the dual cases only, while the c_8, c_{10} , and c_{11} results differ for both non-dual and dual cases. Because of the lack of dual symmetry in the c_3 and c_5 collisions, we have been mildly deceptive in figure 4.5: the c_3 and c_5 collision classes are not equivalence classes for the LGM-1 gas. Nevertheless, we shall continue to speak of them as if they were, pointing out the difference when necessary.

There is one other difference in the LGM-1 rules: a rest-particle at a barrier site vanishes. This violates conservation of mass, but does not change the behavior of the lattice gas. Because this last property of the LGM-1 gas creates some difficulties in testing, we will have more to say about this when we discuss the construction of a complete test set for LGM-1.

The canonical collisions can be used to specify a particular collision by providing the rotation, duality, and barrier status. The collisions listed in figure 4.5 are presumed in rotation 1, written $c_{k(1)}$ for collision c_k. Clockwise rotation by increments of $\pi/3$ are written $c_{k(2)}$, $c_{k(3)}$, ..., $c_{k(6)}$. The dual of a collision is written by $\overset{*}{c}_k$, and the presence of a barrier by c_k^b. For example, figure 4.6 shows that canonical collision c_2 generates 24 distinct collisions: six rotations, their duals, six rotations with barriers, and their duals.

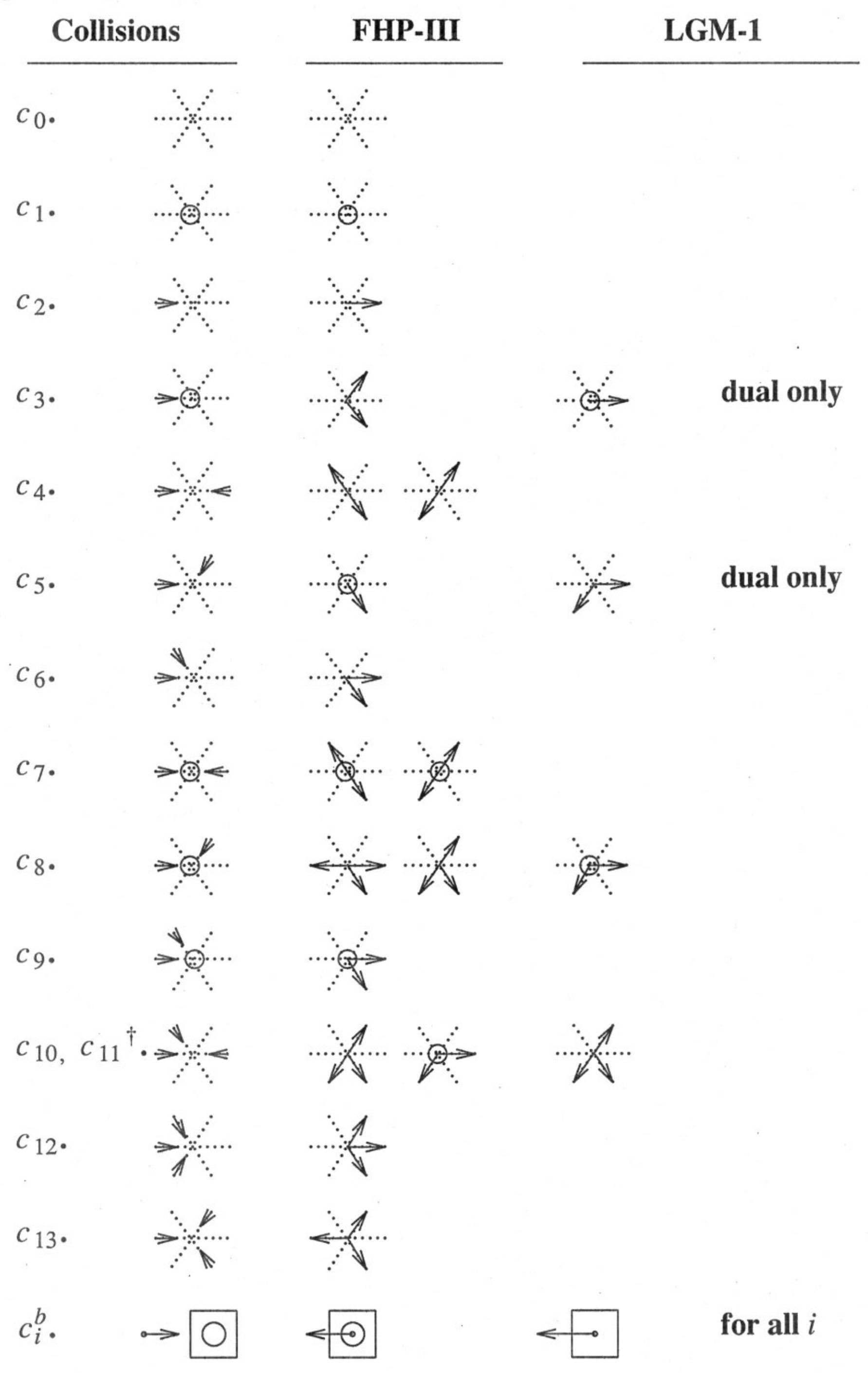


figure 4.5.

The rule sets for the FHP-III and LGM-1 gasses. The first column shows all canonical collisions, c_0 *through* c_{13}*, and a single example of the barrier version,* c_i^b*. The second column shows the results for the FHP-III gas. The LGM-1 rules are shown only in the cases where they differ from the FHP-III gas. (†) The canonical collisions* c_{10} *and* c_{11} *are reflections of each other through a horizontal line. Shown is the canonical rule for* c_{10}.

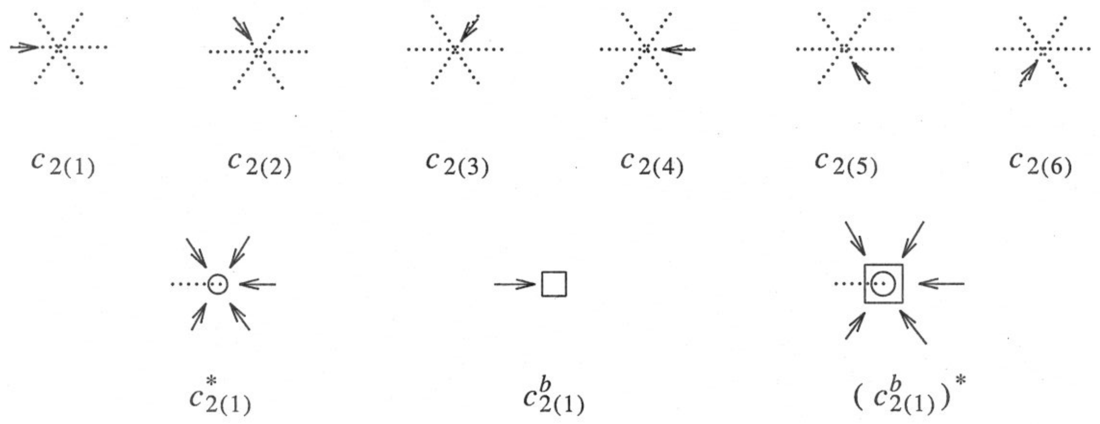


figure 4.6.

The actual collisions generated by canonical collision c_2. *There are 24 total: six rotations for each of* c_2, c_2^*, c_2^b, *and* $(c_2^b)^*$.

4.1.2. Definitions and Notation

Before beginning a discussion of the test mechanisms, we need to establish some definitions and conventions. As we described above, a particular lattice-gas automaton consists of a rule set, a lattice-graph, and the variables associated with the elements of the lattice-graph. Throughout the following we will assume the rule set and the form of the lattice-graph are both fixed and correspond to the LGM-1 lattice gas unless otherwise stated. Consequently, we may think of any such automaton as a system of individual particles that obey the collision rules and exist on a lattice-graph. An instance of a lattice-graph together with a specification of the barrier sites we call a *space,* a combination of a collection of particles and a space we call a *system,* and an arrangement of particles in the space a *state* of the system. With each system there is an associated time t defined by assuming some specified initial arrangement of a system is assigned time $t = 0$ and each subsequent update of the automaton state increases t by one. A sub-system may have a local time t_{local} with an origin different from the global time origin; we write local time as a function of global time: $t_{local}(t)$. The idea is that an initial state of a system is built up from smaller systems whose states at global time zero are achieved by starting the local systems in their initial states and running them forward or backward to $t_{local}(0)$. Thus, a local system can have its time origin shifted from the global clock's. A system is termed *cyclic* if in its evolution it ever repeats a state. A system is said to be *closed* if boundaries are placed in such a way that particles outside the system can never enter. When we wish to indicate the state of a system Γ at time t we write $\Gamma^{(t)}$. For the period of Γ we write δ_Γ.

A collision is said to *occur at time t* in system Γ if the collision exists at some site in $\Gamma^{(t)}$. If a collision occurs at some positive t in Γ we say Γ *contains* the collision. When we discuss systems that detect evolution errors we will want to know not only if a collision is contained in the system, but also whether the system detects bit errors in the result. Before we go on, we should pause to explain what we mean by *detection* of bit errors.

Detection of bit errors in this testing method does not have the usual meaning. For instance, even though the final result of a computation may have one or more incorrect bits, this is not necessarily a detectable error by our definition. Detection of an error here is somewhat subjective in that the human detects the error by looking at a signal on a computer screen. Determining that the signal is not correct depends on the level of detail and complexity that the human eye and brain can cope with ''easily.'' The tests we have devised give a clear, visual, graphical signal of failure for most errors. However, for some errors the signal is not apparent to the human eye, unless close inspection is done. In that case, we say the error is not detectable. Because of this human factor, we cannot say with mathematical precision what determines a detection event; however, it should be clear, at least intuitively, what we mean by detection, and the figures presented below should make the matter obvious.

We will say a system *covers* a bit error if the system detects the error. There are eight possible one-bit errors in any collision result, which we denote $\{e_r,\ e_b,\ e_1,\ e_2,\ e_3,\ e_4,\ e_5,\ e_6\}$, corresponding to the eight variables associated with a lattice site. When we wish to specify a particular error in a particular collision we will append the bit-error notation to the collision notation with a colon separating the two: for instance, $c_{2(1)}{:}e_r$ shows that the result of a c_2 collision at rotation 1 has the rest-particle bit set to the complement of its value in the correct result.

4.2. Test Sets and Their Constituents

A particle system that tests a simulator for all one-bit errors in the evolution of a lattice gas we call a *complete test ensemble*. A test ensemble consists of a collection of cyclic sub-systems called *test-cycles*. Test-cycles are built from elementary generic sub-systems called *pattern templates*. (A *pattern* is produced from a pattern template by setting parameters related to the size of the pattern, its spatial orientation, the number of particles contained, and so on. Any such pattern could serve as a template, and we shall sometimes refer to templates and patterns interchangeably as patterns.) A test-cycle consists of a collection of sub-systems derived from a set of pattern templates by applying symmetry operations and time translations to the patterns produced from the templates. A test-cycle can constitute a portion of a complete ensemble, or it may be inserted into a larger system simulating a fluid flow problem, to serve as a runtime error detector.

While one might consider trying to compile a pattern template set that has as few members as possible, each one covering as many collisions as possible, we found that a simple greedy algorithm lead to a set of templates that allowed us to construct a complete test ensemble that was small enough to be practical. To be more specific we proceeded as follows:

(1) Select a canonical collision

(2) Design a pattern template that contains that collision.

(3) Simulate every one-bit error for that collision and check to see that the error is detected by that pattern. If a single-bit error is not detected, return to (2) for a new test pattern template. Record which canonical collisions are covered.

(4) Repeat (1)-(3) until all collisions are covered.

Note that it is only necessary to simulate bit errors (Step (3)) for an update of the canonical collision. If an error is detected in the canonical configuration, every bit error in its equivalence class will also be detected by an appropriately transformed version of the pattern.

After we had a complete set of pattern templates we could proceed with the construction of a complete test ensemble. Starting with an empty ensemble and using the test pattern collision coverage table, we proceeded as follows:

(1) Select a canonical collision/one-bit error combination not yet covered by our ensemble of test-cycles.

(2) Select a pattern covering that error.

(3) Build a test-cycle from the pattern by applying all the symmetry transformations to the pattern so that all actual collisions generated by the canonical collision are contained in the test-cycle.

(4) Complete the test-cycle started in (3) by making time delayed versions[†] of the system constructed in (3).

† We will explain this in detail later. The reason for time-delayed versions is to ensure that every processor in a pipeline will ''see'' the collision.

(5) Continue in this way, checking off the covered collisions , until all one-bit errors are covered by our ensemble of test-cycles.

Our complete set of test pattern templates contains 51 templates. Each pattern is enclosed in a square 30 $\times$ 30 box of barriers. If we naively build the test-cycles for the LGM-1 lattice gas, we need, for each pattern, six rotations of the pattern and its dual, and about three different time delayed versions of each of these. In addition, a complete ensemble for the LGM-1 architecture requires every collision on both even and odd rows. Altogether the complete ensemble built this way requires close to thirty-five hundred copies of the patterns. On the host machine for LGM-1, a SUN 3/160C‡ workstation, we can display about one-thousand patterns per screen, and thus we can see the entire ensemble in four screens of bit-mapped graphics. In practice, we reduce this considerably in two ways. First, we skip step (4) above: no time delayed versions are required if we make the pipeline length relatively prime to the least common multiple of pattern cycle times. The length of our pipeline has been set to a prime number of stages. Second, we are slightly more careful in constructing the ensemble: not every pattern requires all twelve combinations of rotations and duals. The ensemble we use for a complete system test contains 786 patterns and its display occupies about three-quarters of the host's screen. We will return to the topic of display when we discuss error detection, but now we want to describe patterns and test-cycles in detail.

4.3. A Test Pattern

Patterns are generic, self-contained sub-systems designed to contain one or more canonical collisions. Our collection of pattern templates is divided into groups of similar type designated by roman letters *A* through *K*. There may be more than one version of a particular type; for instance, type *K* has twenty-eight versions K_1 through K_{28} (see appendix B for a catalog of the complete template library).

The patterns are built using a small library of particle/barrier devices which act as control mechanisms for collections of moving particles and individual moving particles. Groups of moving particles are called *chains*. A chain is simply a collection of particles laid out uniformly in a line, all following an identical path. The control mechanisms for chains consist of devices to deflect a chain, called *turns*. There are also mechanisms called *gates* that reflect single particles at only one phase of a cycle, while letting particles pass at all other phases, and *detectors* that detect the incorrect

‡ SUN is a trademark of Sun Micro Systems, Incorporated.

presence or absence of a particle involved in some cyclic collision. We will discuss a few of the components in detail below as we describe the construction of pattern A; descriptions of the other components can be found in appendix B.

Pattern A was designed to test the c_3 and c_5 collisions, and consists of three chains traveling around a roughly triangular circuit (see figure 4.7).

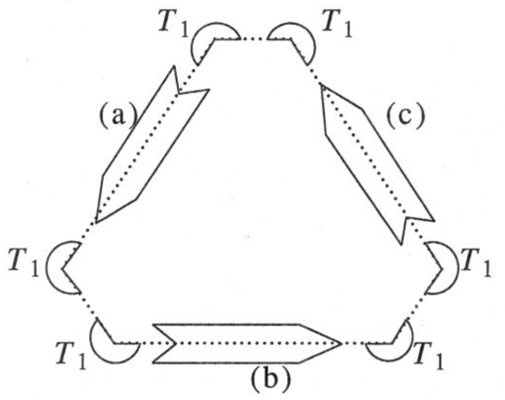


figure 4.7.

Pattern A schematic. (a), (b), (c) are chains of particles with velocities $\mathbf{v}_6$, $\mathbf{v}_4$, *and* $\mathbf{v}_2$, *respectively. The deflectors at the corners are type* T_1 *turns.*

The chains are of the ''paired'' type: they consist of pairs of particles such that the members of a pair are separated by an empty site. The turns in A are designated type T_1 turns, and are designed to handle paired chains (see figure 4.8). The collisions contained in A occur, for the most part, in the turns, which we now describe in detail.

The T_1 turns used in A consist of a rest-particle in the path of the oncoming stream, and a barrier at an adjacent lattice site. The location of the rest-particle is called the *origin* of the turn: the incoming stream will experience a $\pi/3$ deflection from its path at the origin. Let us follow a pair of particles, p_1 and p_2, through a counterclockwise T_1 turn (see figure 4.9). The lead particle of the two-particle stream, p_1, collides with the rest-particle, r_1, in a c_3 collision at $t = 0$. The resulting state of the origin has two particles leaving the site: one at $\pi/3$ counterclockwise from the p_1's initial direction and one at $\pi/3$ clockwise, which we take to be p_1 and r_1, respectively. The barrier is located one site away from the origin, and at $t = 1$ particle r_1 experiences a c_2^b collision with the barrier sending r_1 back to the origin. At $t = 2$ particles r_1 and p_2 collide in a c_5 configuration at the origin, leaving r_1 in its original location at rest and sending p_2 out the same edge that p_1 used to exit the turn. The pair of particles have thus turned a corner, and any stream consisting of similar pairs can likewise

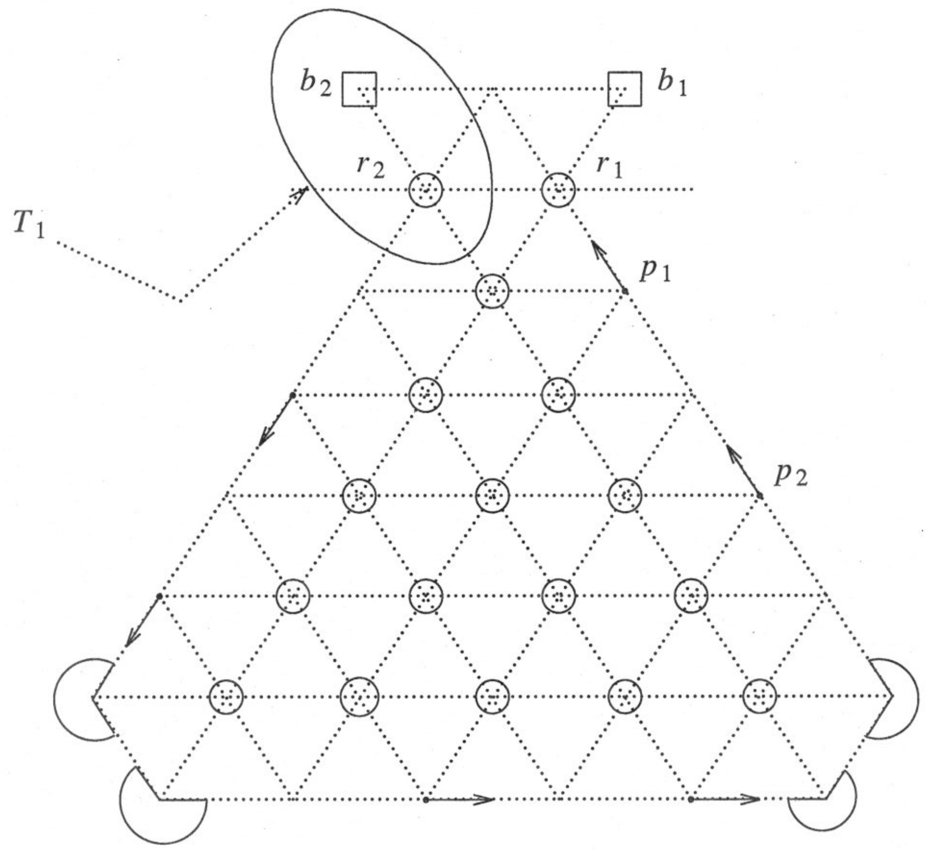


figure 4.8

Pattern A detail. Particles p_1 and p_2 constitute a one-pair chain with velocity $\mathbf{v}_2$. Two type T_1 turns are shown in detail at the top of the figure: rest-particle r_1 and barrier site b_1 constitute one turn, r_2 and b_2 constitute the other.

be turned.

In this section we have shown the details of a single test pattern. The next task is that of constructing a test-cycle from such a pattern, which is taken up in the following section. To do that we will need to know the error coverage of our patterns, that is, what errors are definitely detected by each pattern. (See figure 4.12 for an example of pattern *A* detecting an error.) By simulating the patterns as closed systems with a simulator that has its rule set altered by the corresponding error, we can test error coverage of the patterns. Table 1 shows the confirmed error coverage for all errors. From that table we can see that pattern *A* covers all one-bit errors for the non-dual c_3 and non-dual c_5 collisions. Another piece of information we will need is the collision timing table for our patterns. Table 2 shows the local time of each of the collisions c_3 and c_5

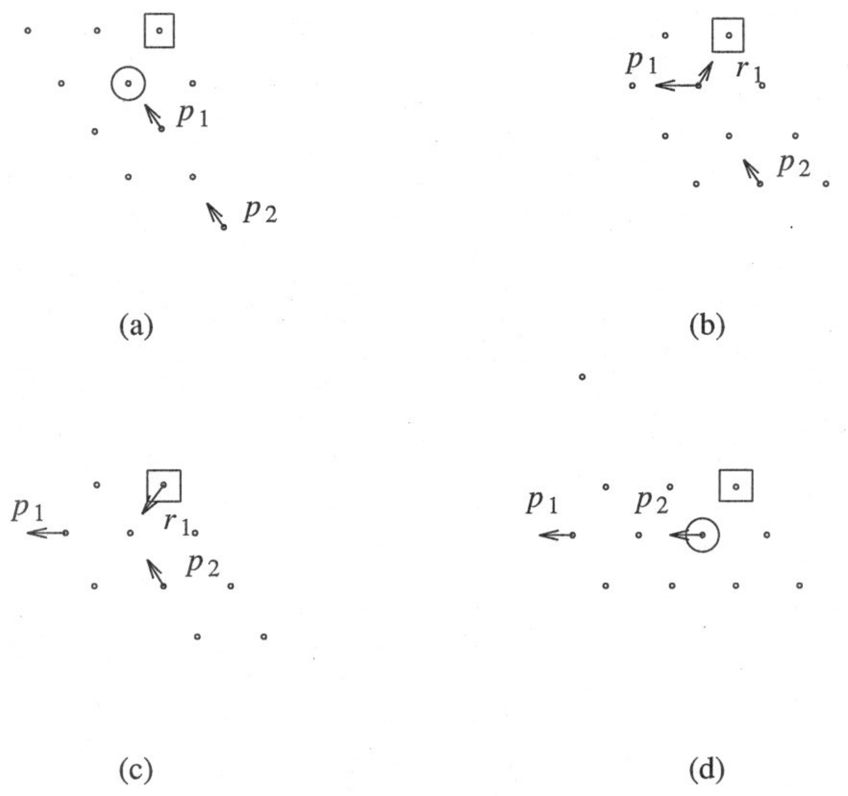


figure 4.9.

Four consecutive time steps of a pair of particles being deflected by a T_1 turn. (a) p_1 and the rest-particle collide in a c_3 collision. (b) p_1 is deflected by $\pi/3$, r_1 collides with a barrier in a c_2^b collision. (c) p_2 and r_1 collide in a c_5 collision. (d) p_2 is deflected by $\pi/3$, r_1 is again at rest.

in pattern A.

4.4. Constructing a Test-Cycle

This section shows the construction of a test-cycle from several copies of pattern A. We have two goals in building test-cycles: (1) to build a system that contains the complete collision class for one or more canonical collisions, and (2) to build a system that can present every processor in a multi-processor machine with collisions. The second goal can be achieved for our parallel machine, LGM-1, by including in the test ensemble patterns with shifted time origins, and we will return to this below. The first goal can be achieved by looking at the collisions contained in a pattern and adding enough symmetry transformed copies of the pattern to form a system that contains all the collisions of a particular class.

Confirmed One-Bit Error Coverage							
collision	error			collision	error		
	1-6	r	b		1-6	r	b
c_0	E_1	E_1	K_{27}	c_0^b	E_2		K_{27}
	E_1^*	K_{20}	E_1^*		E_2^*		E_2^*
c_1	E_3	K_{26}	E_3	c_1^b	E_4		K_{28}
	E_3^*	K_{21}	E_3^*		E_4^*		E_4^*
c_2	B_1	B_1	B_1	c_2^b	B_1		B_1
	B_1^*	K_{16}	B_1^*		B_1^*		B_1^*
c_3	A	A	A	c_3^b	G_2		G_2
	H_2^*	K_{17}	H_2^*		G_2^*		G_2^*
c_4	C_1	K_5	C_1	c_4^b	G_1		G_1
	C_1^*	K_6	C_1^*		G_1^*		G_1^*
c_5	A	A	A	c_5^b	G_6		G_6
	B_2^*	K_{18}	B_2^*		G_6^*		G_6^*
c_6	F_3	K_1	F_3	c_6^b	G_3		G_3
	F_3^*	K_2	F_3^*		G_3^*		G_3^*
c_7	C_2	K_7	C_2	c_7^b	G_1		G_1
	C_2^*	K_8	C_2^*		G_1^*		G_1^*
c_8	B_1	B_1	B_1	c_8^b	G_6		G_6
	B_1^*	K_{19}	B_1^*		G_6^*		G_6^*
c_9	F_1	K_3	F_1	c_9^b	G_3		G_3
	F_1^*	K_4	F_1^*		G_3^*		G_3^*
c_{10}	B_1	K_{12}	B_1	c_{10}^b	$D_1(cl)$		$D_1(cl)$
	B_1^*	K_{13}	B_1^*		$D_2^*(cl)$		$D_2^*(cl)$
c_{11}	B_1	K_{12}	B_1	c_{11}^b	$D_1(ccl)$		$D_1(ccl)$
	B_1^*	K_{13}	B_1^*		$D_2^*(ccl)$		$D_2^*(ccl)$
c_{12}	F_2	K_{10}	F_2	c_{12}^b	G_4		G_4
	F_2^*	K_{11}	F_2^*		G_7^*		G_7^*
c_{13}	$D_1(cl)$	K_{14}	K_{23}	c_{13}^b	G_5		K_{24}
	$D_1^*(cl)$	K_{15}	K_{22}		G_5^*		K_{25}

table 1.

For each collision each one-bit error was simulated in both dual and non-dual forms. The upper rows in each collision category are the non-dual cases, the lower rows are the duals.

Pattern A Collision Containment						
	period $\delta_A = 6$					
phase	0	1	2	3	4	5
collision	c_0 $c_{2(2,4,6)}$ c_1 $c_{3(1,3,5)}$	c_0 $c_{2(2,4,6)}$ $c^b_{2(1,3,5)}$ $c_{3(2,4,6)}$	c_0 $c_{2(2,4,6)}$ $c^b_{2(2,4,6)}$ $c_{5(1,3,5)}$	c_0 $c_{2(2,4,6)}$ c_1 $c_{5(2,4,6)}$	c_0 $c_{2(2,4,6)}$ c_1	c_0 $c_{2(2,4,6)}$ c_1

table 2.

The period of pattern A is adjustable by lengthening its chains; shown here is the containment for A with minimum period. Where more than one rotation of a collision occurs in a phase we have written the rotations in a list: for instance, $\{c_{2(2)}, c_{2(4)}, c_{2(6)}\}$ *becomes* $c_{2(2,4,6)}$.

From the collision containment of pattern A listed in table 2, one can see that a single instance of A contains collisions c_0, c_1, c_2, c^b_2, c_3, and c_5. The dual of A, A^*, would contain the duals of the collisions just mentioned. In pattern A, the canonical collision c_3 occurs in every rotation, $c_{3(1)}$ — $c_{3(6)}$, as do all the collisions in table 1 except c_2, which appears only in even rotations. Consequently, if we want a test-cycle that covers all c_2 collisions we must combine A with a copy of A rotated by $\pi/3$ by abutting the two patterns side-by-side without interference. Let us call this combination test-cycle Ω^1_A.

The test-cycle Ω^1_A covers every rotation of the collisions listed in table 2. If we build a new test-cycle Ω^2_A by combining a copy of Ω^1_A and a dual copy $(\Omega^1_A)^*$, again abutting them side-by-side, we will have a test-cycle containing the complete collision classes for every collision contained in pattern A. Unfortunately, Ω^2_A does not achieve our goal for the collision class c_3. As we mentioned earlier, the c_3 and c_5 classes are each split into two because of dual asymmetry. As a consequence, pattern A does not cover any of the dual cases for c_3, but patterns H_2 and K_{17} together do, as can be seen in table 1. Consequently, a test-cycle combining Ω^1_A with test-cycles built from these two additional patterns is required. Nevertheless, for the sake of simplicity in the following, we will assume that Ω^2_A covers the c_3 and c_5 classes.

The second goal in test-cycle construction involves collision containment for multi-processor implementations of the lattice-gas simulator. If the machine doing the lattice-gas simulation is a uniprocessor machine, we would be satisfied with this Ω_A^2 test-cycle. Simulating this test-cycle would cause the machine to repeatedly update the test-cycle and every site would be processed by the same processor. Then, in a single period of the test-cycle, the processor would ''see'' every collision in the collision classes contained in Ω_A^2. If, however, we are interested in testing a multi-processor machine, we need to insure that every processor sees every collision. We have been interested primarily in testing the LGM-1 machine, which is a linear pipeline in which each stage of the pipeline executes one update of the entire lattice-gas. In each stage there are two processors: one updates odd rows of the lattice, the other updates even rows. We next discuss the construction of a test-cycle specifically targeted for the LGM-1 architecture (see figure 4.10).

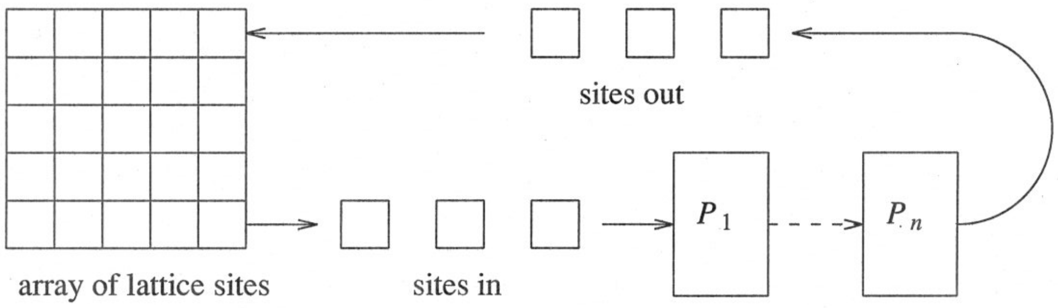


figure 4.10.

The LGM-1 architecture. The lattice sites are stored in an array and fed in raster scan order to a pipeline of processors. Each processor advances the lattice gas system one evolution step and sends the data out in the same raster scan order.

For the moment we will assume each stage of LGM-1 has only one processor. If we look at the second stage of the pipeline, $P^{(2)}$, we see that it updates the automaton at global update steps $t_g \equiv 1 \pmod{N}$, where N is the number of stages in the pipeline. Consequently, in order for $P^{(2)}$ to see collision $c_{k(i)}$ we must build our test-cycle so that at some $t_g \equiv 1 \pmod{N}$ we have $c_{k(i)}$ occurring somewhere in the cycle.

Because all the copies of pattern A in test-cycle Ω_A^2 have the same phase, $t_{local}(0) = 0$, individual collisions are contained in some phases of Ω_A^2 and not others. For instance, from table 2 we see that the period of pattern A, δ_A, is six global update steps and that collision c_3 occurs in even rotations at local time $t \equiv 0 \pmod{\delta_A}$, and odd rotations at $t \equiv 1 \pmod{\delta_A}$. Therefore, a copy of Ω_A^2 with its local time set to zero, $\Omega_A^2(0)$[†], has a period of six and contains all rotations of c_3 at $t \equiv 0 \pmod{\delta_A}$, and at $t \equiv 1 \pmod{\delta_A}$. Nevertheless, unless the period is relatively prime to N or $N/3$, every stage of the pipeline will not see c_3. One way to overcome this is to build a test-cycle that contains every collision on every phase of its cycle. As table 2 shows, we can cover the missing cycle steps by reproducing Ω_A^2 three times with local time origins 0, 2, and 4; the result we will call Ω_A^3. The complete test-cycle Ω_A^3 containing six copies of pattern A has a cycle time of six and contains c_3 in every rotation at every phase of the cycle.

As we mentioned above, LGM-1 uses two processors within each stage; one processor handles the odd numbered rows and the other processor, the even ones. To handle this we must check that our test-cycle also contains every collision on both even and odd numbered rows. The simplest way around this problem is to make two copies of the test-cycle, installing them so that their row positions differ by one. Alternately, one can ensure that every collision is represented for both row parities by adding more turns to pattern A to make a new pattern.

4.5. Error Detection and Experimental Results.

Our method of testing a simulator consists of simulating a complete test ensemble for many generations, and after the simulation, graphically displaying the ensemble system to detect errors in its evolution. For our machine, LGM-1, the state of a lattice-gas system is stored as a two-dimensional array of bytes in raster scan order, each byte containing the state of a single lattice site. Because we want to be confident that we are actually seeing the state of the system, we want the process of displaying it to minimize the amount of transformation done to the original data. We therefore treat the array as a bit-map graphics file, and display it with a color map that color codes each byte.

The color coding is done so that if there is only a single particle present at the site, the presence of the particle is indicated by a color corresponding to its velocity (see figure 4.11); if there are several particles present, their colors will add in such a way that the resultant color corresponds to the vector sum of the velocities. Higher intensity corresponds to greater number of particles present. So, for instance, if all six non-zero velocities are present the site will appear bright white (white represents zero velocity), if no particles are present it will appear dim white, and if only a rest-particle is present

† System S with local time origin shifted so that $t(0) = x$ will be written $S(x)$. That is, the state of $S(x)$ at global time 0 is $S^{(x)}$.

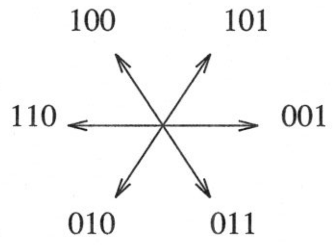


figure 4.11.

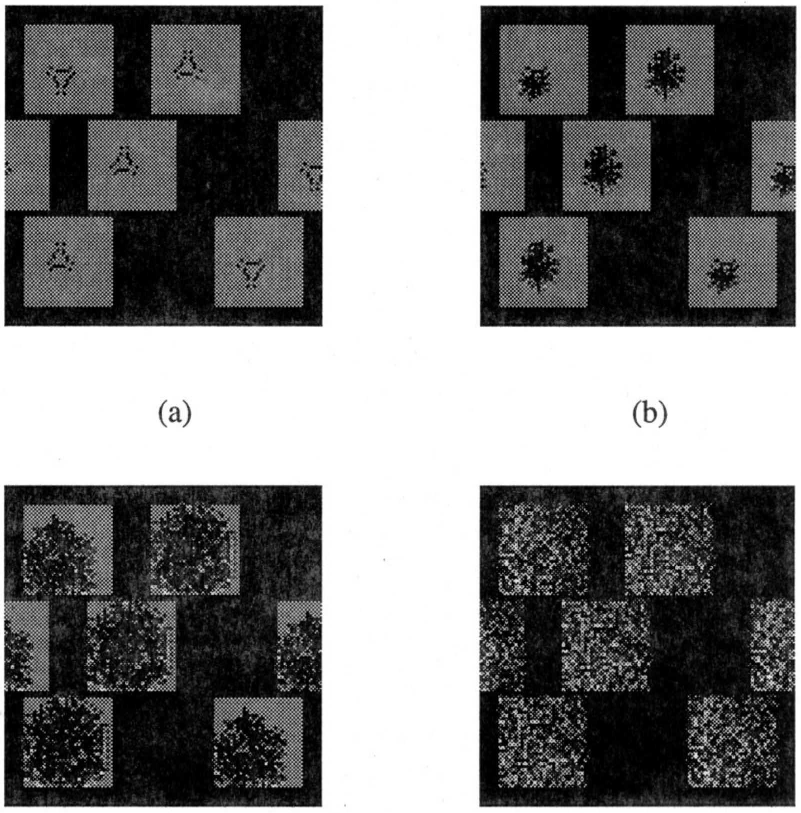
Color coding of unit vectors. Each direction vector is associated with an rgb (red, green, blue) color coding, and the zero vector is coded 111 (white). We imagine that the colors change continuously and linearly with the angle of rotation. For instance, $\mathbf{v}_1$ *is associated with the triple 110, and* $\mathbf{v}_2$ *is associated with the triple 100. Any vector between* $\mathbf{v}_1$ *and* $\mathbf{v}_2$ *will have a triple of the form* $\{1, x, 0\}$, *where* $0 \le x \le 1$. *Given a collection of velocity vectors at a lattice site, adding the corresponding rgb triples and normalizing gives an rgb triple that codes a color that matches the resultant vector sum's direction. The number of particles present determines the brightness of the color. Thus* $\mathbf{v}_1 + \mathbf{v}_2 \rightarrow 210$, *and normalizing gives* $\{1, 0.5, 0\}$ *as the sum's color with a relative brightness of 2.*

it will appear a noticeably brighter white than when no particle is present.

Different color maps can be used to bring out different features: for instance, in the color map described above, barrier sites are coded without color so that they appear black unless other particles are present at the site. If seeing the location of the barriers is important, the barriers can be coded with some distinctive color.

We want the presence of an error to result in a state of the system that is visually easily distinguishable from any of its correct states. Our patterns have the general property that an error will disrupt the cycle and result in one or more particles straying beyond the sites occupied by the pattern. In most of our patterns we put a rest-particle in any lattice site which is not part of the pattern but inside the containing box of barriers. A stray particle passing into the space occupied by rest-particles usually results in the rest-particles erupting in a chain reaction that floods the box with moving particles. This chaotic state is visually unmistakable.

Shown in figure 4.12 is a small, example test-cycle evolving in the presence of a one-bit error: the lattice-gas rules in a software simulator have been altered to contain the one-bit error $c_{3(1)}{:}e_1$. The test-cycle consists of six copies of pattern A and is designed for periodic boundaries so that one of the patterns wraps around the sides. The pictures in the figure are the color-coded bitmaps of the states converted to gray scale: each pixel represents a lattice site, and the color scheme described above has



(a) (b)

(c) (d)

figure 4.12.

A test-cycle containing six copies of pattern A simulated with the $c_3{:}e_1$ error at every step. (a) The cycle in its initial state, $t = 0$. (b) $t = 10$. (c) $t = 20$. (d) $t = 100$. The second row of patterns in each picture above contains two patterns: one is "wrapped around" the left and right edges of the lattice.

been mapped to a gray scale.

In (a) the initial state of the test-cycle is shown, and each succeeding image shows the system ten time steps older. As is easily seen even in gray scale, the chaotic result is clearly distinguishable from the initial state. Because any legal state of the test-cycle looks very similar to the initial state, we have no problem in distinguishing the error indication from the system's correct states.

Besides having extension in evolution time, these test sub-systems also have extension in the space of the lattice-gas, allowing them to detect errors such as those caused by incorrect data addressing by the lattice-gas simulator. For detecting these types of errors, the patterns that are ''loops,'' such as pattern *A*, are useful. For instance, in LGM-1 a lattice-gas system is ''cut'' into strips that are fed to the pipeline one at a time and ''sewn'' together as they exit the pipeline. Placing a loop pattern across such strip boundaries makes it easy to see if there are any addressing errors made in the cutting and sewing operations. Likewise, if the lattice-gas's boundaries are periodic, the loops can be placed across the boundaries. In LGM-1 the boundaries can be either periodic or not, and we test both cases with different test ensembles.

4.5.1. Detection Difficulties

Our ensemble detects all significant one-bit errors, but not all are detected by chaotic conditions. One reason for this is the lack of dual symmetry for the collision of a single particle with a rest particle in the LGM-1 gas: we cannot have ''explosions'' in a dual world. We have two methods of getting around this.

One method is to build a pattern with many particles moving in an orderly fashion. Disruption of the pattern results in many stray particles and, while not giving the magnitude of chaos in an explosion, it is easy to see when the system fails to evolve correctly.

Another method is to enclose the dual pattern in a box with a device that functions as a gate for dual particles (holes). The dual particle is transformed to a real particle and the real particle can cause an explosion outside the dual box.

For some of our patterns, instead of employing either of these two methods, we have relied on being able to see one or two stray particles or a global change that is not chaotic. While this is not the most desirable method of detecting errors, employing it allowed us to complete the set of patterns without spending time attempting to produce explosions for every error. Indeed, the attempt might be futile: it is an open question whether or not it is possible to find a collection of patterns that results in chaotic conditions for every one-bit error.

Another problem in detection is a consequence of the mismatch in scales caused by detection without chaos and the large number of patterns in our test ensemble. As we mentioned in section 4, our ensemble contains close to one-thousand patterns. A chaotic change in any of one-thousand 30×30 pixel boxs displayed at one time is easy to see at a glance. But, because the size of the ensemble is much larger than the area of a single pixel (which may show the presence of a stray particle), the patterns that do not become chaotic require closer inspection. We have handled this by scanning a display of the test ensemble using a mouse driven ''magnifier'' that allows us to see the patterns individually. While this works, it is not entirely satisfactory.

One way to approach this, besides trying to make all patterns detect by chaotic results, is to reduce the size of the ensemble. If the ensemble were small enough, a single particle would occupy a sufficient portion of the display so that, again, a glance would suffice. We have made no attempt to minimize our set. Rather, we have been interested in ensuring its completeness, and we have, for the most part, used each pattern as a test of a single canonical collision. As the description of test pattern A showed, a single pattern contains several collisions, and for this reason we estimate that our complete set is considerably larger than necessary.

For instance, the pattern that was specifically designed to test the c_0 (quiescent lattice site) collision is simply an empty box. If any particles or barriers appear in the box the error is detected, but the result will not necessarily be chaotic. It seems likely that the c_0 test pattern is superfluous, and other patterns will detect any error that it can. Confirming this could be done by simulating the eight possible one-bit errors for the canonical collision c_0 on the entire ensemble and checking to see that the errors were detected by some other pattern in every case.

There is one type of one-bit error which we make no claim of detecting. Because these errors do not effect the lattice gas behavior in a way that changes its modeling capability we consider these types of errors insignificant. As table 1 shows, these errors are all rest-particle errors in collisions in which a rest particle and a barrier exist at the same site. As we mentioned when we described the LGM-1 rule set, in the LGM-1 gas the rest-particle disappears in these configurations. Consequently, any test pattern for this type of collision can only be used as a one-shot test, and this test must occur at $t = 0$.

The patterns we have devised for these collisions do detect all one-bit errors, but the one-shot nature of this testing forces us to deny fully covering these collisions for two reasons: one is that the errors cannot be detected in any processor other than the first one in an LGM-1 type pipeline machine, and the other is that the patterns must be inspected very closely to detect the error of a rest-particle remaining at the site after $t = 0$. In general, this is a difficult type of change to detect because the rest-particle at a barrier lattice site has no interaction with other particles. Altogether, twenty-eight of our test patterns K_1 through K_{28}, were created especially to deal with these rest-particle/barrier problems.

4.5.2. Experimental Results

In this section we describe the results of experiments with simulated multi-bit errors. The experiments we have done for the c_3 collision show it is unlikely the ensemble will fail to detect multi-bit errors. Using a software simulator for the LGM-1 gas, we ran simulations of the test-cycle in figure 4.12. For each simulation run we altered the rule set to include an error in the result of the $c_{3(1)}$ collision. A total of 117 simulations were run, each for 20 update steps: all 8 possible one-bit errors, all 28 possible two-bit errors, all 56 possible three-bit errors, and 25 selected four-bit errors. The four-bit errors were all those of the form $e_{3,\,5,\,x,\,y}$ and $e_{5,\,r,\,x,\,y}$ because these were the types of errors that resulted in detection difficulties in the two-bit and three-bit cases. After each simulation the state of the test-cycle was observed using the technique described in section 7.1. As table 3 shows, there was only one error that failed the test completely: the $c_{3(1)}{:}e_{5,\,r}$ error.

The reason the $e_{5,\,r}$ error escaped detection is that in this test-cycle this error partially erases its own mistake in such a way that the cycle continues on undisturbed. Of course, a careful inspection of the individual pixels would reveal the error, but from our viewpoint this is not an acceptable method of error detection. Rather, we want the error to expose itself in such a way that a glance at a monitor screen would suffice to determine its existence. One remedy for this particular detection problem is to make a copy of pattern *A* with the particle chains traveling around the loop in the opposite direction. The new pattern would detect the $c_{3(1)}{:}e_{5,\,r}$ error, but would fail for the $c_{3(4)}{:}e_{6,\,r}$ error, which one would expect since the two patterns are mirror images of each other as are the two errors just mentioned.

4.6. Architectures and Applicability of the Test Method

The testing method we have described was conceived with the LGM-1 pipeline architecture in mind. Coincidently, the method also works well for testing software implementations. The element that makes the method applicable to LGM-1 is the scanning style of data flow through the update processors. Because of this data scanning, the entire lattice is passed through every processor, every collision contained in the lattice-gas system is processed by every processor, and thus all the update logic is tested.

Experimental Results for Simulated Errors					
number of incorrect bits		1	2	3	4
non-chaotic detection results	stray particles	0	$e_{3,5}$	$e_{3,4,5}$	0
	lack of particles	0	0	$e_{3,5,r}$	$e_{3,4,5,r}$
	undetected	0	$e_{5,r}$	0	0

table 3.

The results of experiments simulating multi-bit errors in the evolution of the test-cycle shown in figure 4.12. All errors simulated resulted in a chaotic system except those listed above. For instance, the two-bit error $e_{3,5}$ resulted in "stray particles" occurring in the test-cycle. Stray particles are defined as particles in the test-cycle which are moving beyond the limits of the particle paths in the correctly functioning cycle. A "lack of particles" result means that all the moving particles have disappeared from the cycle. An "undetected" result means that the cycle is visually indistinguishable from a correctly evolving test-cycle. This last result could be detected by a program directly comparing two cycles.

Contrasted to this scanning data flow is the parallel implementation of a lattice-gas automaton that assigns a single processor to each lattice site. Here, the test method described in this chapter implies building a lattice-gas system that has every collision occurring at every lattice site. This is probably impractical for two reasons, one of which is that the test ensemble would be excessively large. For instance, one would need a separate lattice-gas system for every possible collision, which amounts to $28 \times 6 \times 2 = 336$ lattice-gas systems the size of the entire automaton. Which brings us to the second reason this is a bad idea: even if the set of test patterns was much smaller, the detection would have to be done without the aid of chaotic "explosions" contrasting with an ordered system. While it is true that an entire lattice initialized to contain the same collision at every lattice site may appear organized to start with, detecting the failure of a small percentage of randomly located processors would probably require looking closely at every lattice site because the lattice-gas system would likely appear randomized after a few updates. An alternative that could be investigated in the future would entail shifting the lattice state so that the test patterns travel from processor to processor. However, this approach is beyond the scope of the present work.

Between the the architecture mentioned above using one processor per lattice site, and the architectures that scan the entire lattice, such as uni-processors and linear pipelines like LGM-1, lies a continuum of architectures we dub ''frame-oriented architectures.'' A frame-oriented architecture processes a lattice by assigning the update of some fixed region, or frame, of the lattice to each processor and exchanges information about the frame boundaries between appropriate processors. Our test method is directly applicable when the number of frames is small because we can duplicate the test ensemble for each frame and proceed as usual, treating each frame as if it were handled by a separate machine. The LGM-1 machine is a two-frame architecture because the lattice sites in even and odd rows are processed by what amounts to two separate pipelines of processors. As we said earlier, we handle this by duplicating the test ensemble and translating it by one row in the lattice. In general, this approach is only worthwhile when the number of frames is small, and consequently this presents a trade-off in the design of architectures for lattice-gas simulations. That is, one must trade testability, at least by our method, against the number of frames the architecture employs.

4.7. Summary and Conclusions

This chapter has presented a testing method based on visual detection of errors in a complex simulation. Because the detection mechanism is based on the human/machine interface, it presents both advantages and some disadvantages over traditional concepts of error detection. Here, the ''detection'' of an error is subjective: a human must be able to ''easily'' see that the test pattern is displaying incorrect results. The advantages of this method are mostly related to the interactive debugging of hardware and software. The individual patterns have limited error coverage so that an error can be traced to a specific rule failure in the implementation. Because timing information is implicitly displayed, a faulty processor can be singled out in a pipelined environment. As well, the types of errors encountered in data manipulation show up in particularly characteristic ways that help to pin point the fault. Finally, the level of confidence in correct behavior is enhanced because the operator can see the test patterns in the simulation data rather than depending on an abstract message from a ''hidden'' testing facility which, in the complex computing environment, can be completely unrelated to the object the operator supposed was being tested.

We have shown how to test lattice-gas simulators with reasonable resources. Easily discernible changes in cyclic ''particle'' patterns signal an evolution error in the lattice gas, and a brief visual scan of bitmapped graphics suffices to determine whether or not an error has occurred. More precisely, our test ensembles are built from a library of 51 pattern templates, each pattern occupying roughly 50 lattice sites contained in a ''box'' of barrier sites 30 lattice spacings on a side. A naively constructed ensemble to test a software simulator of an LGM-1 gas requires 612 patterns: two duals and six rotations of all 51 pattern templates. The ensemble we have used for this purpose consists of 393 patterns. An ensemble for the two-frame architecture of the LGM-1 requires twice that many, or 786 patterns. The display of this 786 pattern ensemble occupies about 77% of the host's display screen. For the purposes of testing every processor in an LGM-1 type architecture containing an arbitrary number of stages, time shifted copies of the patterns are required resulting in an ensemble about three times larger, that is, containing about 2400 patterns displayed on two and one-half screens of bitmapped graphics. In practice, we have always set the number of pipeline stages so that the number of stages is relatively prime to the cycle times of the patterns, and consequently only 786 patterns are needed for our complete ensemble. Because of the hole/particle symmetry of the FHP-III gas, a complete ensemble for it requires only about two-thirds the number of patterns as the LGM-1 gas.

These ensembles detect all significant one-bit errors in the evolution of a simulated lattice-gas system, and our experiments suggest that multiple errors are unlikely to escape detection. Simulating the complete ensemble tests the correctness of the implementation of the update rules, the data addressing logic, and data transmission and general system functions to the extent that they effect the lattice-gas system's evolution. Because the patterns are cyclic, testing continues for as long as the lattice-gas system containing them is evolved. This allows the ensembles to test every processor in a pipelined architecture such as LGM-1, and it also allows them to act as runtime error detectors by incorporating them into real fluid flow lattice-gas systems. Also, these embedded tests allow some detection of transient failures in the simulating system during simulation runs.

The technique described in this work can be used in any software implementation. For hardware implementations the technique's applicability depends on the way the lattice sites are assigned to the processors. If the lattice is split into non-overlapping pieces, or frames, that are updated by disjoint sets of processors, the ensemble must be duplicated for each such frame. As the number of frames increases the technique soon becomes impractical, and there is consequently a tradeoff between testability by this

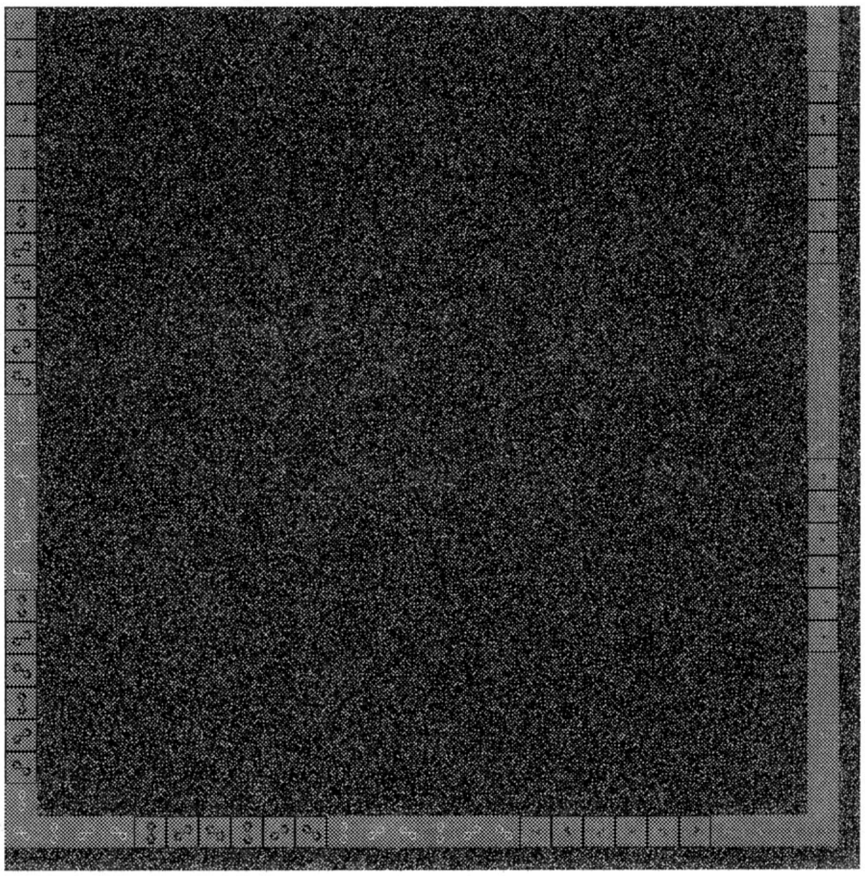


figure 4.13.

Test patterns embedded in an 800 × 800 site lattice-gas fluid flow experiment. The state of the lattice is represented by a gray scaled version of the color scheme mentioned in section 7. The light gray border on three sides contains 76 test patterns. The stippled center section contains the lattice-gas flow, seen here after 10,000 update generations. Although it cannot be seen, there is a forced flow across the top edge of the image. The test patterns are incorporated into the boundaries defining the shape of the flow ''well.''

method and the architectural parameter associated with the number of frames.

Of course, as with any testing facility, the issue of the correctness of the thing being tested becomes the issue of the correctness of the test. For our test method, the issue becomes one of verifying the correct implementation of the test patterns, and the question then becomes one of establishing the correctness of the software, amounting to several thousand lines of code, that constructs and manipulates the test ensemble. While confirming the software is free of bugs by traditional software testing and verification methods is difficult, we have been able to refer to the resulting patterns themselves for confirmation. The reason is that the patterns are simple and easy to understand visually, even when displaying them amounts to no more than color coding an octal dump of the data. Thus, we have been able to use the patterns themselves to debug the software that creates them.

The use of this test method in practice has shown it to be an efficient aid in the construction, verification, and operation of lattice-gas simulators. In the construction of the machines we have used it to to screen for faulty custom chips that contain the processors of the LGM-1 lattice-gas simulator, and to detect hardware design errors in the custom boards. For these purposes we have used a subset of the complete set of patterns and found that screening a single chip requires about three minutes of real time. Most of this time is spent inserting the chip in the test socket, setting up the simulation, and displaying the result. We have used the test method during construction of software controlling the simulation. We found that writing simulation software was speeded up considerably by the availability of a concurrent test facility: the code could be quickly written, modified, or redesigned because a simple five minute test would detect errors as they were introduced into the system and allow them to be corrected immediately. In fact, the test template library routines, the software simulator, and the image handling library were developed in parallel: the simple graphical nature of the test patterns allowed debugging of each concurrently, even though none of these systems was complete. For verification of the system before and after simulation runs, the entire system test required about five minutes of real time. We have also conducted fluid flow experiments on LGM-1 using an 800×800 lattice containing embedded test patterns to detect runtime errors (see figure 4.13). These embedded test patterns added about 10% to the simulation time, and the complexity of specifying the initial state of the lattice was increased by approximately a factor of two.

Acknowledgement

We would like to thank David P. Dobkin and Eleftherios Koutsofios for their encouragement, advice, and assistance in the graphic display of the test patterns and ensembles. The displays were done on SUN 3/160C and IRIS 4D 220GTX workstations† using the *Cheyenne* [59] graphics library.

† SUN is a trademark of Sun Micro Systems, Incorporated. IRIS is a trademark of Silicon Graphics, Incorporated.

Chapter 5
Conclusions

5.1. Summary, Conclusions, and Future Work

Chapter 2 presented a refinement of existing pebbling games in order to model I/O in parallel computations, defined precise characterizations of the subsets in a pebbling, developed estimates on the sizes of these subsets, and used these estimates to establish an upper bound on throughput for two computational problems, the lattice-graph based computations on the two-dimensional toroidal grid and the two-dimensional triangular lattice on the torus. The result shows that the throughput of the WSA architecture is within a small factor of the throughput of any machine with identical fixed resources of main memory communication bandwidth and local storage capacity. The factors are about 6 for the two-dimensional grid, and 5 for the triangular lattice-graph. That is, the WSA architecture (a linear pipeline machine) runs at least $1/6^{th}$ the speed of any machine computing similar problems in two-dimensions with similar resources. So, for these types of problems, no other organization of computation steps and internal communication (internal to the processor) —even given infinite internal communication capacity and infinite calculating power (arithmetic-logic operational speed)— can attain significantly better performance, given the fixed amount of local memory and communication to main memory available.

It is not surprising that a pipelined machine makes efficient use of communication resources. In fact, approximate bounding values for throughput were known through previous work, but only to within one or two orders of magnitude of the present results. And while the suspicion that reorganization of the computation steps and machine resources could not improve LGM-1's throughput by more than a small amount initially motivated the work in this chapter, there was another motivation as well. The motivation behind this work was not only to find out precisely how good or bad the WSA architecture is, but the considerable effort put into refining the pebbling arguments stems from the desire to find a method of determining an optimal computational strategy and thereby the most powerful machine possible within the resource constraints.

To find an optimal strategy, we need to understand the nature of the forces induced by resource constraints and how these forces work to form an optimal parallel computational strategy. The optimal strategy is embodied in the moves of an optimal pebble game and can be determined from the shapes of the pebbling sets defined by sub-pebblings: if the shapes of sub-pebblings in an optimal pebbling can be discovered, the optimal order of computation and I/O strategy will be known. Before this effort, the pebbling arguments gave little or no clue to the shape of a sub-pebbling. The work in Chapter 2 makes some headway in this direction by suggesting the shapes of sub-pebbling sets when the only constraint is to maximize the number of nodes calculated. The candidate sets are the (ISO-1)-extremal sets presented in section 2.3. We have shown that they can be characterized essentially as cones within the data-dependency graph.

However, the present method of analyzing the pebble game is done only on a temporal basis: the nodes associated in a sub-pebbling set do not necessarily have any connection spatially, but are associated by their proximity in time in the pebbling moves. Consequently, the shapes of these sets do not give direct information about the optimal strategies. An area for further investigation is to add this spatial information to the pebbling analysis. For example, as was mentioned in section 2.4, a factor of two improvement in the bounds derived there can be attained easily, if the spatial relationship of nodes was characterized by the extremal sets presented there. The possible improvement in the bound is a consequence of the fact that the present analysis characterizes only input I/O, and the observation that a node that is not an input node and is used for input must have been used as output previously. If the sub-pebbling sets were spatially related, one could determine where in the graph the nodes used for input were, and thereby determine which nodes where used for output.

In the process of refining the pebbling set size estimations, Chapter 2 also introduced the use of the Wulff Construction to solve a discrete iso-perimetric problem. The method introduced there solves a discrete isoperimetric problem by starting with the solution of a closely related continuous isoperimetric problem and deriving the result for the discrete case. This approach makes an interesting counterpart to previous work by Bollobás and Leader [27] which works in somewhat the reverse direction by ''smoothing out'' a discrete problem to a more manageable continuous one. An interesting question is whether the two methods can be used to show a way of connecting continuous and discrete problems more closely.

Aside from the application of Wulff's Construction to isoperimetric problems, there is another possible use for this tool. It would be interesting to attempt to apply the Wulff Construction directly to the data dependency graph. This would require a careful definition of the integrand to correspond to the costs of recomputation and storage of information. One can look at the data dependency graph as dynamically maintaining a red and blue pebbled surface that descends in the graph as the computation proceeds. The cost associated with the surface of a crystal set would have to correspond to the cost of maintaining this red-blue surface.

Finally, another direction open to investigation suggested by Chapter 2 has to do with applying the pebbling techniques to expanded classes of machine architecture. The communication bound derived previously used the terminology of red and blue pebbles. The essential distinction was simply that the registers associated with the blue pebbles communicated through some limited capacity channel with the registers associated with red pebbles. In fact, there was no restriction made that every register in the machine was either red or blue. It is easy to see that the machine can be divided up into as many different pieces as desired and each piece assigned a color. The form of the communication bound between different pieces remains the same. This would allow the modelling of communication in a distributed or hierarchical context. Consequently, another area for future work would be the extension of the present methods to allow the analysis of hierarchical and distributed machine designs.

In Chapter 3, we have presented a scalable architecture, MWSA, based on multiple, independent WSA-architecture pipelines for two-dimensional cellular automata. The scaling scheme for this architecture avoids wiring a two-dimensional wiring pattern across circuit boards which justifies our definition of the cost function for the machine as a linear function of its size. Avoiding two-dimensional wiring is accomplished in the MWSA architecture by using an overlap-save strategy to allow the computation of independent blocks of the lattice without cross communication between neighboring pipes. We have derived the efficiency and throughput of the MWSA architecture, and we have derived its cost function by employing known costs from previous machines and rules-of-thumb in workstation construction. We have introduced a definition of throughput speed-up based on the least total machine cost rather than the number of logical processing units. Using a numerical gradient search method to find the least-cost configuration of MWSA machine resources for a given size lattice problem under the constant time constraint, we have shown that the MWSA scalable architecture has slightly superlinear speed-up, as a function of cost, over a moderately large range of lattice sizes. We have shown that the optimal-cost configurations have essentially fixed, short (circa 40 stages) pipelines, with moderately wide local shift-registers (circa 600 lattice sites), and narrow data paths (circa 10 lattice sites wide), for the range of lattices with 10^4 to $2 \cdot 10^5$ lattices sites on an edge.

As Chapter 2 emphasized, the bandwidth to main memory is critically important in the cost of the machine. In Chapter 3, the bandwidth has been scaled by adding multiple ports. However, the total machine cost is linearly dependent on the memory cycle rate, ω. Thus, although results in Chapter 2 suggested total bandwidth is the important quantity, this chapter stresses the importance of the speed of the communication channel. Since a special-purpose processor for scientific computation has to compete favorably with general-purpose machines to be cost-effective, the conclusion suggested by Chapter 3 is that the economy of VLSI cannot alleviate the need for cheap, fast communication in special-purpose machines.

In Chapter 3, the cost function we used was based on relatively high cost per shift-register cell. The above analysis can be redone using a cost more in line with current costs of 1Mbit DRAM's. We have started this and found that, in this case, the asymptotic values for the number of pipeline stages and the pipeline width are about 200 stages and 10^4 lattice sites, respectively. Possible future work would look at what effect this new cost has on the overall machine cost as compared with the results in Chapter 3.

In Chapter 3, the analysis was only carried out in the regions of parameter space where a single scaling scheme was applicable. Another possibility for future work would be to extend the analysis beyond the points where the match between the problem size and the machine size cause some parameters to become fixed. For instance, the present analysis breaks down when the number of independent pipelines in the optimal configuration becomes less than one: obviously, this is an impossible configuration. The analysis can be carried forward, however, by fixing the number of pipelines at one, and scaling other parameters accordingly. Similarly, at the upper limit of the parameters in the current analysis the size of the shift-registers exceed the size of the lattice, which is again an impossible configuration, and the analysis could again be carried out by fixing the shift-register size. This would allow finding the complete speedup curve for each scaling scheme. Then, the different scaling schemes could be compared by comparing total machine cost. More ambitious work along this line would be the task of comparing the cost of these MWSA scaling schemes with other commercial and special-purpose machines. This would address the general question of whether special-purpose machines are cost-effective and the applications in which they can be employed effectively.

The advancing developments in communication bandwidth using optical technology suggests two areas for further work. One is to look at the consequences of improved bandwidth across chip boundaries. The other area has to do with the decreasing cost and increasing bandwidth of inter-board wiring. These technology improvements will significantly effect the above analysis, and make much larger data paths practical. As well, they may allow less chip area be devoted to storage and thereby allow more processors per chip.

Finally, it would be interesting to include the main memory cost in the analysis. Two questions should be addressed here: whether or not it is possible to attain linear speed-up with fixed-access-time memory, and what the optimal organization might be.

Chapter 4 has presented a testing method based on visual detection of errors in a complex simulation. Because the detection mechanism is based on the human/machine interface, it presents both advantages and some disadvantages over traditional concepts of error detection. The advantages of this method are mostly related to the interactive debugging of hardware and software. The individual patterns have limited error coverage so that an error can be traced to a specific rule failure in the implementation. Because timing information is implicitly displayed, a faulty processor can be singled out in a pipelined environment. As well, the types of errors encountered in data manipulation show up in particularly characteristic ways that help to pin point the fault. Finally, the level of confidence in correct behavior is enhanced because the operator can see the test patterns in the simulation data rather than depending on an abstract message from a ''hidden'' testing facility which, in the complex computing environment, can be completely unrelated to the object the operator supposed was being tested.

Chapter 4 showed how to test lattice-gas simulators with reasonable resources. Easily discernible changes in cyclic ''particle'' patterns signal an evolution error in the lattice gas, and a brief visual scan of bitmapped graphics suffices to determine whether or not an error has occurred. These ensembles detect all significant one-bit errors in the evolution of a simulated lattice-gas system, and our experiments suggest that multiple errors are unlikely to escape detection. Simulating the complete ensemble tests the correctness of the implementation of the update rules, the data addressing logic, and data transmission and general system functions to the extent that they effect the lattice-gas system's evolution. Because the patterns are cyclic, testing continues for as long as the lattice-gas system containing them is evolved. This allows the ensembles to test every processor in a pipelined architecture such as LGM-1, and it also allows them to act as runtime error detectors by incorporating them into real fluid flow lattice-gas systems. Also, these embedded tests allow some detection of transient failures in the simulating system during simulation runs.

The technique described in Chapter 4 can used in any software implementation. For hardware implementations the technique's applicability depends on the way the lattice sites are assigned to the processors. If the lattice is split into non-overlapping pieces, or frames, that are updated by disjoint sets of processors, the ensemble must be duplicated for each such frame. As the number of frames increases the technique soon becomes impractical, and there is consequently a tradeoff between testability by this method and the architectural parameter associated with the number of frames.

Of course, as with any testing facility, the issue of the correctness of the thing being tested becomes the issue of the correctness of the test. For our test method, the issue becomes one of verifying the correct implementation of the test patterns, and the question then becomes one of establishing the correctness of the software, amounting to several thousand lines of code, that constructs and manipulates the test ensemble. While confirming the software is free of bugs by traditional software testing and verification methods is difficult, we have been able to refer to the resulting patterns themselves for confirmation. The reason is that the patterns are simple and easy to understand visually, even when displaying them amounts to no more than color coding an octal dump of the data. Thus, we have been able to use the patterns themselves to debug the software that creates them.

The use of this test method in practice has shown it to be an efficient aid in the construction, verification, and operation of lattice-gas simulators. In the construction of the machines we have used it to to screen for faulty custom chips that contain the processors of the LGM-1 lattice-gas simulator, and to detect hardware design errors in the custom boards. For these purposes we have used a subset of the complete set of patterns and found that screening a single chip requires about three minutes of real time. Most of this time is spent inserting the chip in the test socket, setting up the simulation, and displaying the result. We have used the test method during construction of software controlling the simulation. We found that writing simulation software was speeded up considerably by the availability of a concurrent test facility: the code could be quickly written, modified, or redesigned because a simple five minute test would detect errors as they were introduced into the system and allow them to be corrected immediately. In fact, the test template library routines, the software simulator, and the image handling library were developed in parallel: the simple graphical nature of the test patterns allowed debugging of each concurrently, even though none of these systems was complete. For verification of the system before and after simulation runs, the entire system test required about five minutes of real time.

Future work suggested by Chapter 4 includes automating the pattern design process so that the specification of a lattice gas or other similar simple cellular automaton can be used directly to produce the test patterns. Also, an interesting and obvious extension of this work would be to tackle the problem of testing simple 3-dimensional automata. In line with automating the pattern generation, the development of a graphical interface of this type could be explored as test and debugging tool for more general programs. A logical first step in this direction would be to explore the possibility of adopting the present work to automata whose state information consists of finite precision floating point values. Another direction for future work is to include the possibility of more flexible automaton rules such as would be found in non-homogeneous automata or automata with dynamic boundary conditions. Finally, developing the theoretical basis to understand the error coverage limits of this testing technique would be interesting.

5.2. Related Future Work

There several other areas for future work that are less specifically related to the present work or are broader in scope than the above. The testing work raises interesting epistemological issues related to human/machine interaction and confidence in results in the complex interactive computer environment. It would be interesting to try to define and quantify the theoretical relationship between testing, interfaces, and confidence. For instance, how inaccurate is a person's feeling of confidence about results returned by a computer, and how much effect can graphical tools, or other data presentation media, have on the accurate evaluation of these results.

In the area of architectures and machines, there are several possible projects of interest. One would be to actually build a multiple pipeline machine using the MWSA design using optical data paths wherever possible. This naturally leads to the option of using more general purpose pipeline stages allowing flexibility in the automaton functions. The increased functionality of the pipeline stages brings up several interesting questions. For instance, if the pipeline stages are programmable, how does one program them without adding additional communication lines?

Another interesting problem brought up by considering generalizing the MWSA functionality has to do with the post/pre-processing required by simulations. The results of the simulation must be processed to produce images, spatially averaged results, and so on, sometimes while the simulation is running. The work presented in this thesis has not addressed the architectural requirements of this collateral processing.

What features added to the present architecture would best serve these collateral purposes? How much does this affect the basic premises of the design tradeoffs, and can this type of processing be incorporated into the pipeline stages?

The work presented here on throughput bounds suggests several possibilities for further work. One would be to adapt the present discrete analysis to cover systems, such as analog neural nets, that compute and communicate using analog signals. Another would be to reconsider the definition of main memory bandwidth used in the present work to accommodate the analysis of machines, such as the Connection Machine † whose main memory is essentially distributed. Also, as communication speeds become faster, it may be useful to reconsider the pebbling analysis by introducing the computation time (that is, number of calculation pebbling steps) required together with the I/O time.

The use of discrete, deterministic, cellular automata to model continuous systems presents some interesting questions as well. How does the complexity of the lattice-gas simulation compare with an equivalent floating point finite-differencing method when the operations are compared at the bit level? What are the relative sensitivities to random bit errors of the two methods? Is there some general relationship between finite precision floating point methods and discrete bit level methods suggested by the relationship demonstrated between lattice gasses and fluid flow modelling? Additionally, it is still unknown exactly what the complexity of the lattice gasses are.

† Connection Machine is a trademark of Thinking Machines, Incorporated.

Appendix A
Supergraph Complexity Theorem (Chapter 2)

Essentially, this theorem states that the I/O complexity of a data dependency graph cannot be more than the I/O complexity of any supergraph. The theorem actually shows that any dependency set of a set of nodes M in the original graph is a subset of the dependency of the set M in the supergraph. The proof hinges on the fact that if H is a subgraph of a computation graph G, then $N^H(x) \subseteq N^G(x)$. This is seen by noting that if $y \in N^H(x)$ then there is an arc (y, x) from y to x in H. As this arc is also in G it follows that y is also in $N^G(x)$.

Theorem: Suppose we have a computation graph $G = (V, A)$, where V are the vertices of G and A are the arcs belonging to G. Let $H = (V^H, A^H)$ be a subgraph of G, where V^H are the vertices of H and A^H are the arcs belonging to H. Let M be a subset of the vertices of H. Consider the vertices of H that are in the dependency of M where the dependency is with respect to G, $\bar{D}^G(M) \cap V^H$. Then the number of such vertices is at most the number of vertices in the dependency of M where the dependency is with repect to H. That is,

$$(\bar{D}^G(M) \cap V^H) \subseteq \bar{D}^H(M) .$$

proof: The proof will be an induction on the dependency levels.

(basis):

Suppose $x \in (D_1^G(M) \cap V^H)$. As the support neighborhood of x is in M it follows immediately that

$$N^H(M) \subseteq N^G(M) \subseteq M \text{ , and thus}$$

$$x \in D_1^H(M) .$$

(induction):

Assume $(D_k^G(M) \cap V^H) \subseteq D_k^H(M)$ for $k < i$.

Suppose $x \in (D_i^G(M) \cap V^H)$. We need to show that $x \in D_i^H(M)$. Applying the definition of $D_i^G(M)$ we have

$$N^G(x) \subseteq M \cup \Theta_{i-1}^G \quad \text{where} \tag{1}$$

$$\Theta_{i-1}^G = \left(\bigcup_{j=1}^{i-1} D_j^G(M) \right) .$$

From the observation made in the preceeding paragraph we have

$$N^H(x) \subseteq (N^G(x) \cap V^H),$$

and applying (1) to this expression gives

$$\begin{aligned} &\subseteq (M \cup \Theta_{i-1}^G) \cap V^H \\ &= M \cup \left\{ \bigcup_{j=1}^{i-1} (D_j^G(M) \cap V^H) \right\}. \end{aligned}$$

Applying the inductive hypothesis to the term in parentheses in this last expression results in

$$N^H(x) \subseteq M \cup \Theta_{i-1}^H \quad \text{, and}$$

appealing to the definition of $D_i^H(M)$ establishes the desired result.

□

Appendix B
Test Pattern Library Catalog (Chapter 4)

This catalog contains the descriptions of 51 test patterns for the LGM-1 lattice-gas simulator. Some of the patterns depend on a function of the row number of the lattice site designated as the origin for the pattern. This function is written site_rotation(x), and pertains to the result of the c_4 and c_7 collisions at the lattice site x. As shown in figure 5, there are two possible results for these collisions, and in the LGM-1 machine the choice is determined by the row number of the lattice site at which the collision occurs. The *rotation* of a lattice site is defined to be the direction of the smallest rotation that transforms an input vector into an output vector for the c_4 collision. Site_rotation(x) returns the symbolic value *clockwise* if the head-to-head c_4 collision results in a clockwise rotation at lattice site x, and *counterclockwise* otherwise.

Some of the patterns have a *rotation* parameter. This is not to be confused with the parameter of the same name used to describe the ''turn'' devices. For patterns, the rotation indicates the handedness of the layout of the pattern. Reversing the handedness amounts to reflecting the pattern through a horizontal line.

To save space, many patterns have devices symbolized by a small square labeled with the device name and its parameters. Barriers are also indicated by squares, but they are larger.

T_1.

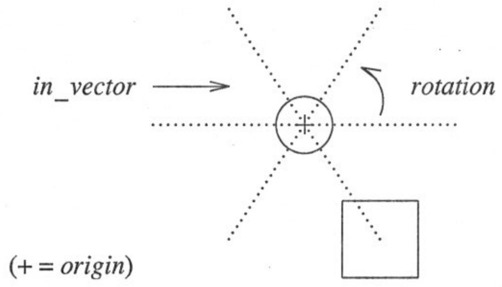
T_1(*origin*, *in_vector*, *rotation*)

acts on: paired chains

delay: none

deflection: $\pi/3$ in *rotation* direction

collisions: c_2, c_3, c_5



T_2.

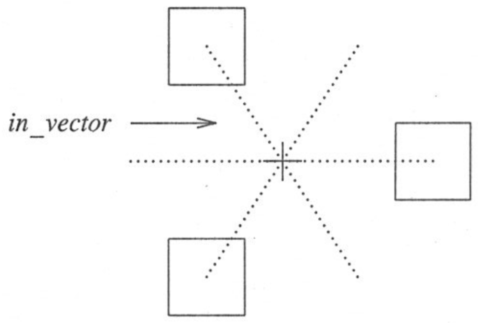
T_2(*origin*, *in_vector*)

acts on: solid chains (*length* $\equiv 0 \pmod 2$)

delay: 2 steps

deflection: $\pi/3$ in site_rotation(*origin*) direction

collisions: c_2, c_2^b, c_4, (c_6)



T_3.

T_3(*origin*, *in_vector*, *rotation*)

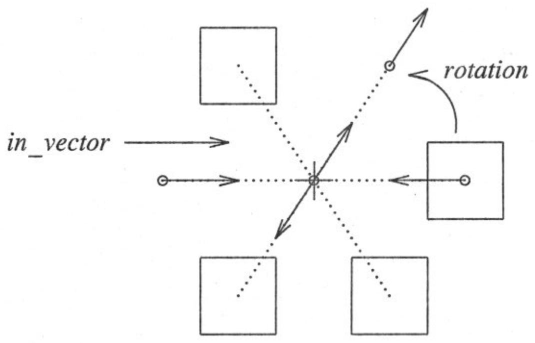
rotation = site_rotation(*origin*)

acts on: endless solid chains

delay: 2 steps

deflection: $\pi/3$ in *rotation* direction

collisions: c_2^b, c_4



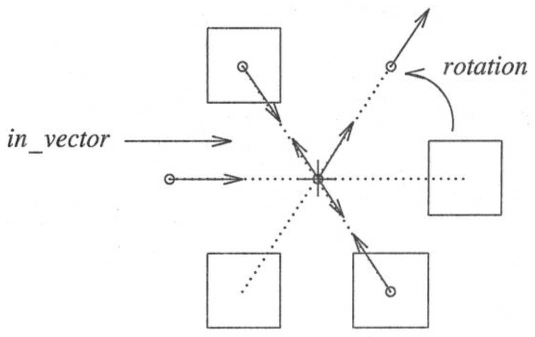
rotation ≠ site_rotation(*origin*)

acts on: endless solid chains

delay: 4 steps

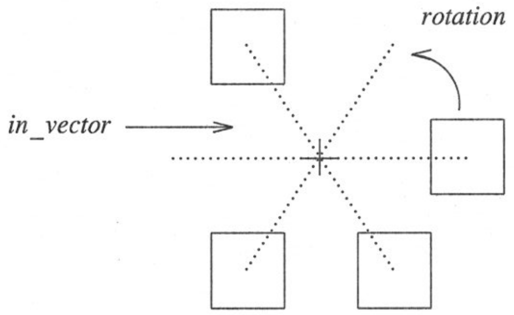
deflection: $\pi/3$ in *rotation* direction

collisions: c_2^b, c_{10} (c_{11})



T_4.

T_4(*origin*, *in_vector*, *rotation*)



acts on: solid chains (*length* $\equiv 0 \pmod 4$))

deflection: $\pi/3$ in *rotation* direction

rotation $\neq$ site_rotation(*origin*):

  delay: 4 steps

  collisions: $c_2, c_2^b, c_4, c_6, c_{10}$ (c_{11})

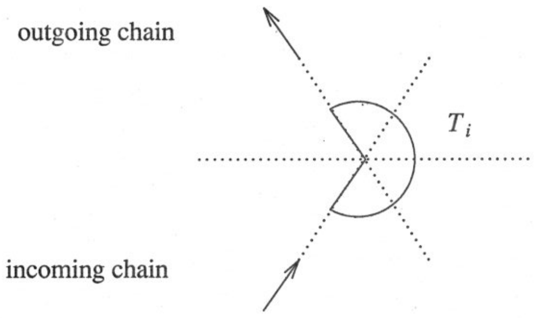
rotation $=$ site_rotation(*origin*):

  delay: 2 steps

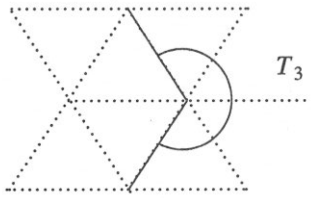
  collisions: see T_2

Turn Symbols

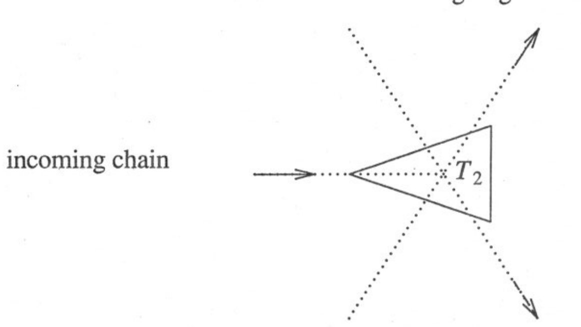
T_1, T_4:



T_3:

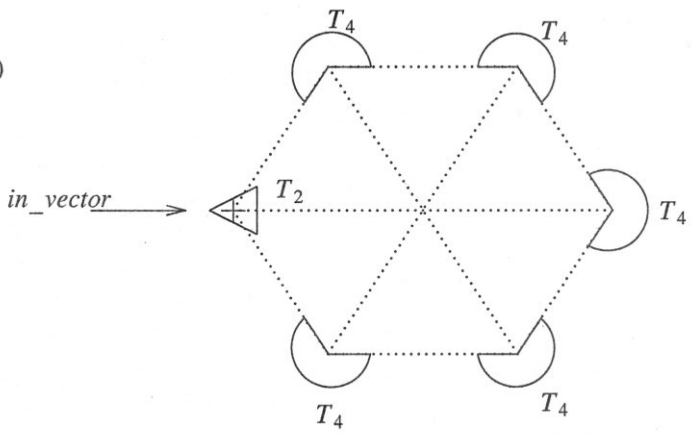


T_2:



T_5.

T_5(*origin*, *in_vector*)



acts on: solid chains with *length* ≤ delay, and *length* ≡ 0 (mod 4)

deflection: 2π

delay: 22 steps for *in_vector* $= \mathbf{v}_1, \mathbf{v}_4$; 26 steps otherwise

collisions: $c_2, c_2^b, c_4, c_5^b, c_6, c_7^b, c_{10}, c_{10}^b, (c_{11}), (c_{11}^b)$

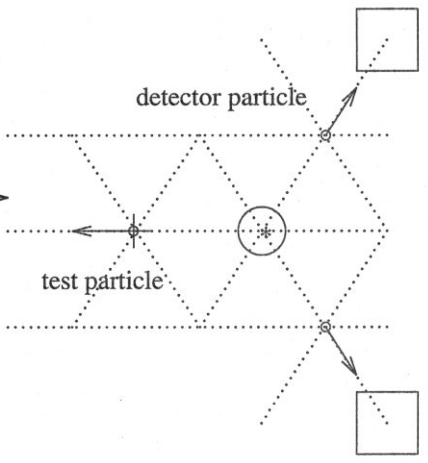
M_{return}.

M_{return}(*origin*, *axis_vector*)

axis_vector $\longrightarrow$

(* = local origin)

peroid: $\delta_{M_{return}} = 4$

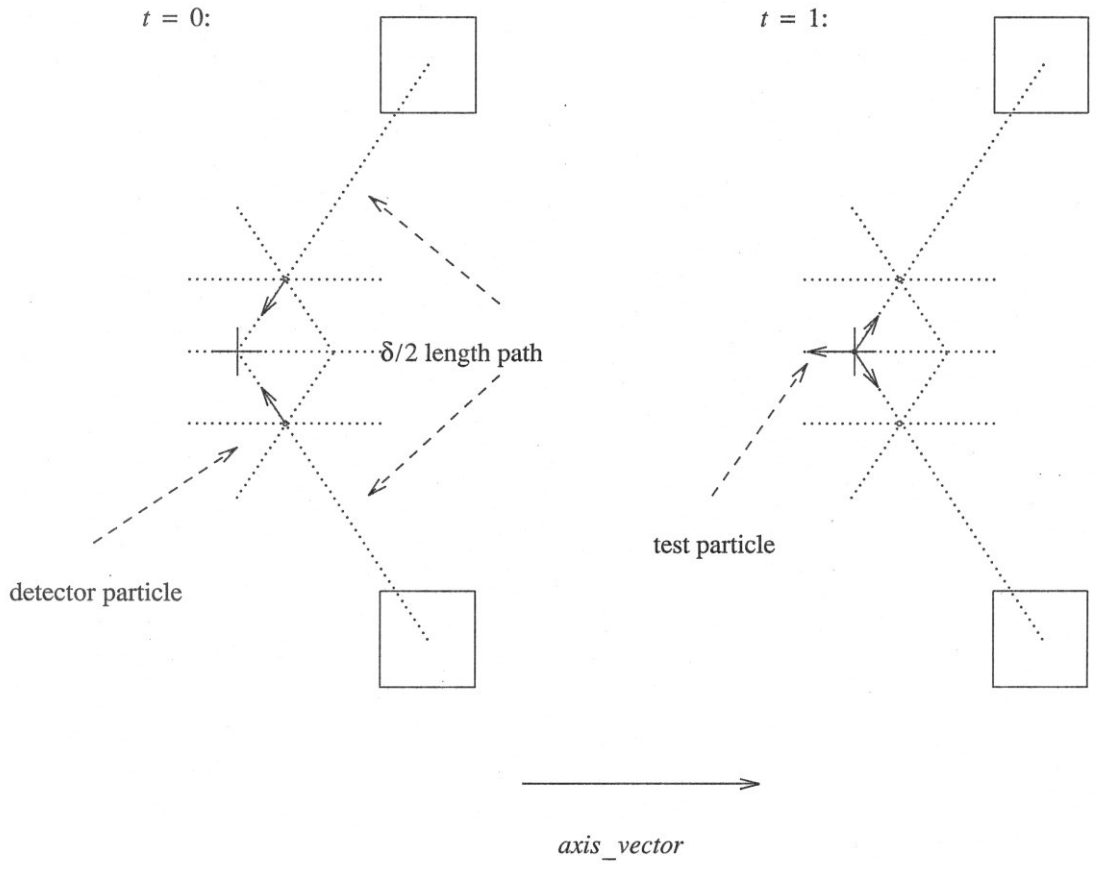


Missing Particle Return Detector:

The test particle is expected to return to origin at $t = 2$ (mod 4). If the test particle does not return, the two detector particles will escape. If the test particle does return the system will return to its original state at $t \equiv 0$ (mod 4).

M_{gate}.

$M_{gate}(\ origin,\ axis\_vector,\ \delta,\ t\ )$



period: $\delta_{M_{gate}} = \delta$

The M_{gate} reflects a single test particle at the phase t of its period. At all other phases particles may pass through the origin in the positive or negative *axis_vector* direction. When $t > 1$ the detector particles are stepped forward in their paths appropriately.

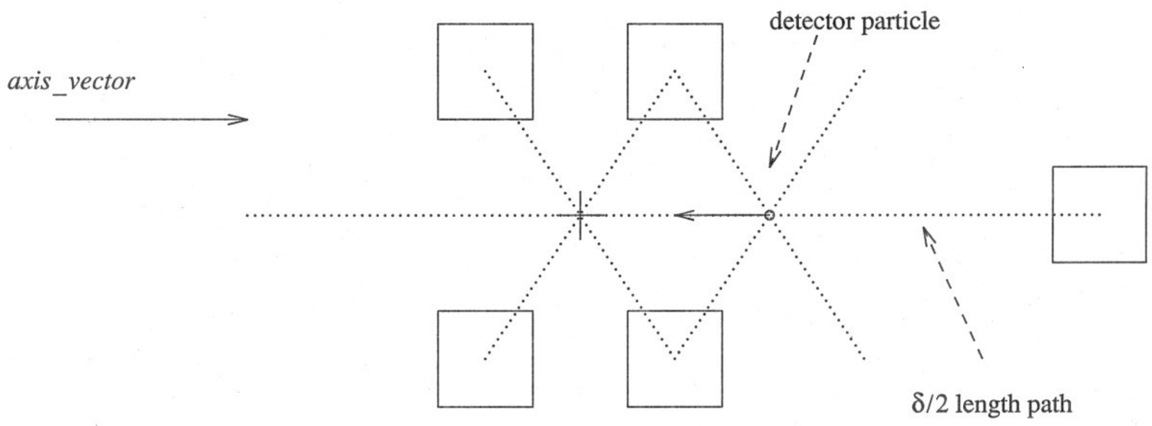
T_{gate}.

$T_{gate}(\, origin,\ axis\_vector,\ \delta,\ t\,)$

reflection delay = 6.

total periond = δ + reflection delay

$t = 0$:

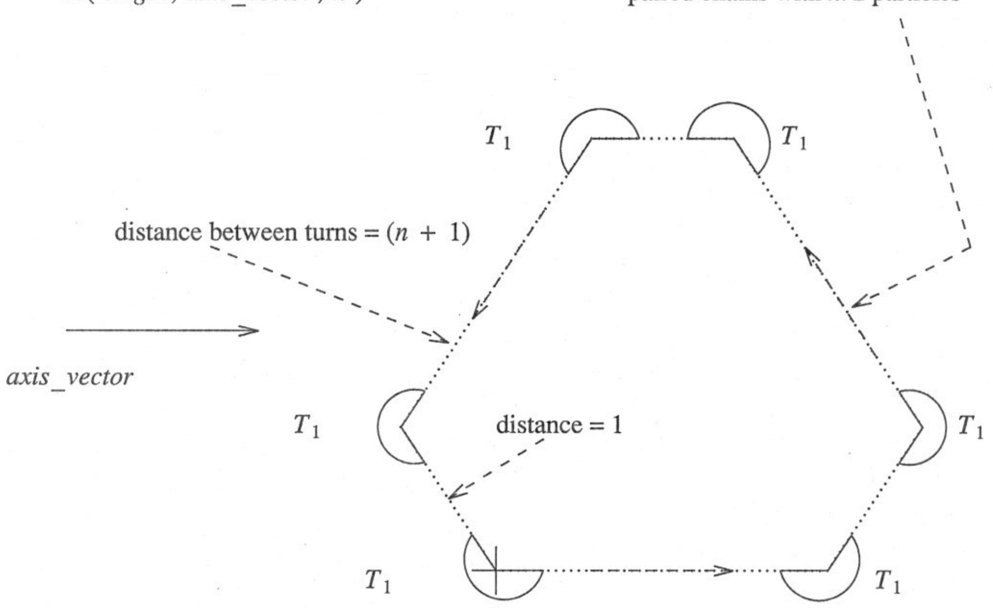


Reflects a single test particle with velocity *axis_vector* at the origin. The reflection occurs because of a c_4 collision between the detector particle and the test particle. If the phase t is advanced the detector particle is moved forward in its cycle and the reflection occurs at the origin at global time $t_{global} = -t$.

Patterns

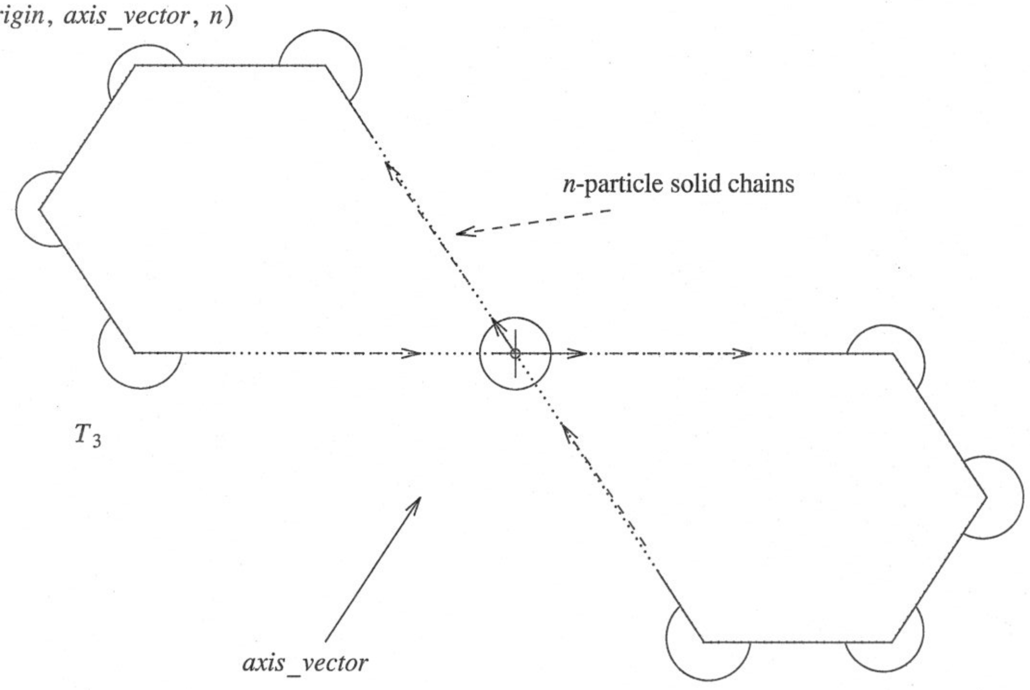
A.

A(*origin*, *axis_vector*, *n*)


paired chains with $n/2$ particles
T_1
T_1
distance between turns = $(n + 1)$
axis_vector
T_1
distance = 1
T_1
T_1
T_1

B.

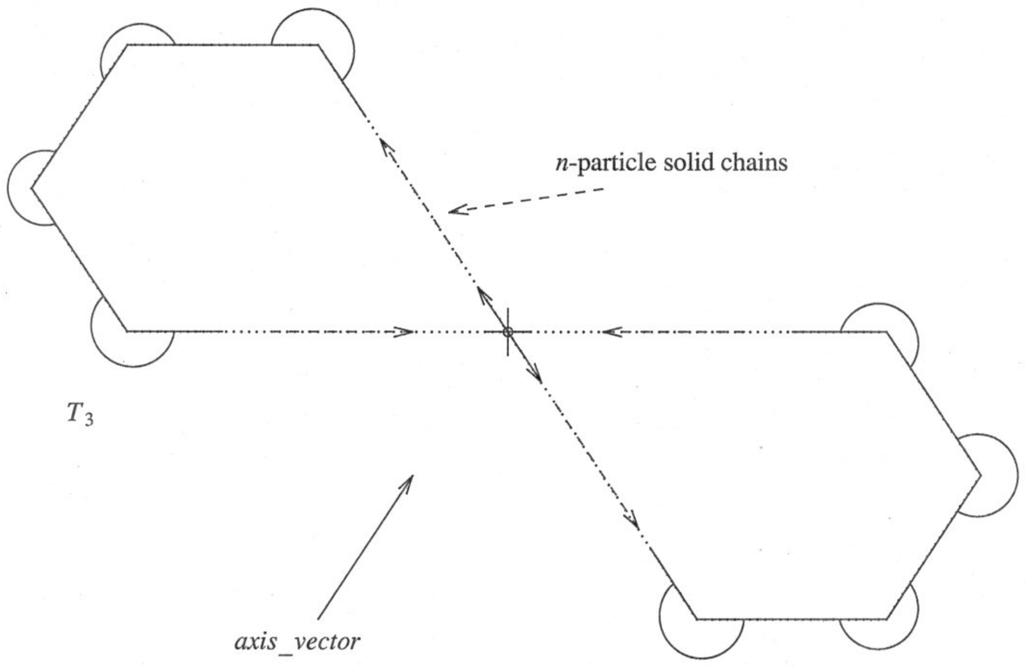
B(*origin*, *axis_vector*, *n*)



B_2 is identical to B_1 except the rest-particle is missing.

C_1.

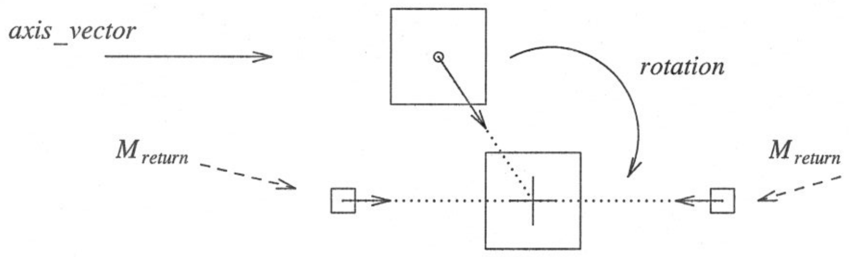
C_1(*origin*, *axis_vector*, *n*)



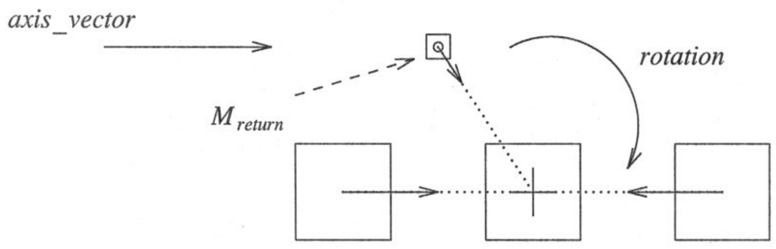
C_2 is the same, but has a rest-particle at the origin.

D_1.

D_1(*origin*, *axis_vector*, *rotation*, *n*)



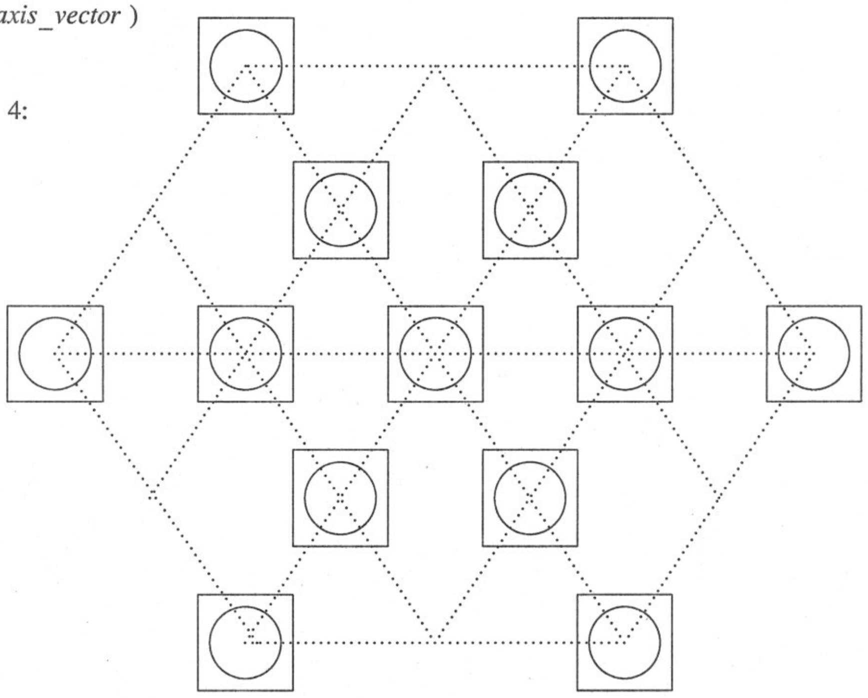
D_2.



E_i.

$E_i($ *origin*, *axis_vector* $)$

$i = 4$:



The *E* patterns are unlike the rest of the patterns in that there are no moving particles. The background is the same in most patterns: every site not occupied by a pattern element contains a rest-particle. The pattern elements here are layed out in a star shape. Any changes in the pattern show up in the alteration of the star pattern or in gross changes throughout the pattern. The test sites contain the following for each of the different *E* versions:

E_1: empty lattice sites

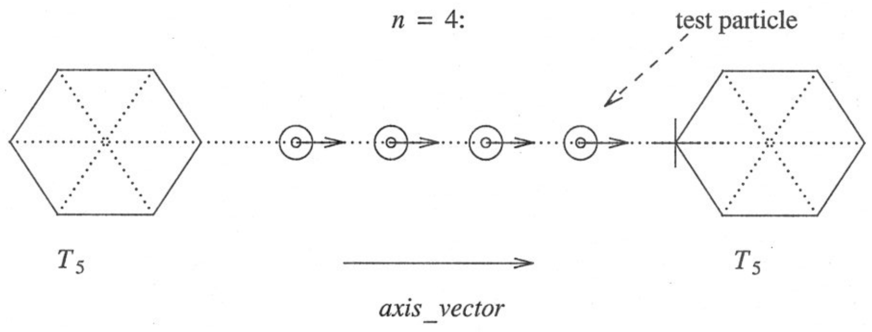
E_2: barriers

E_3: rest-particles

E_4: rest-particle / barrier (shown above)

H_2.

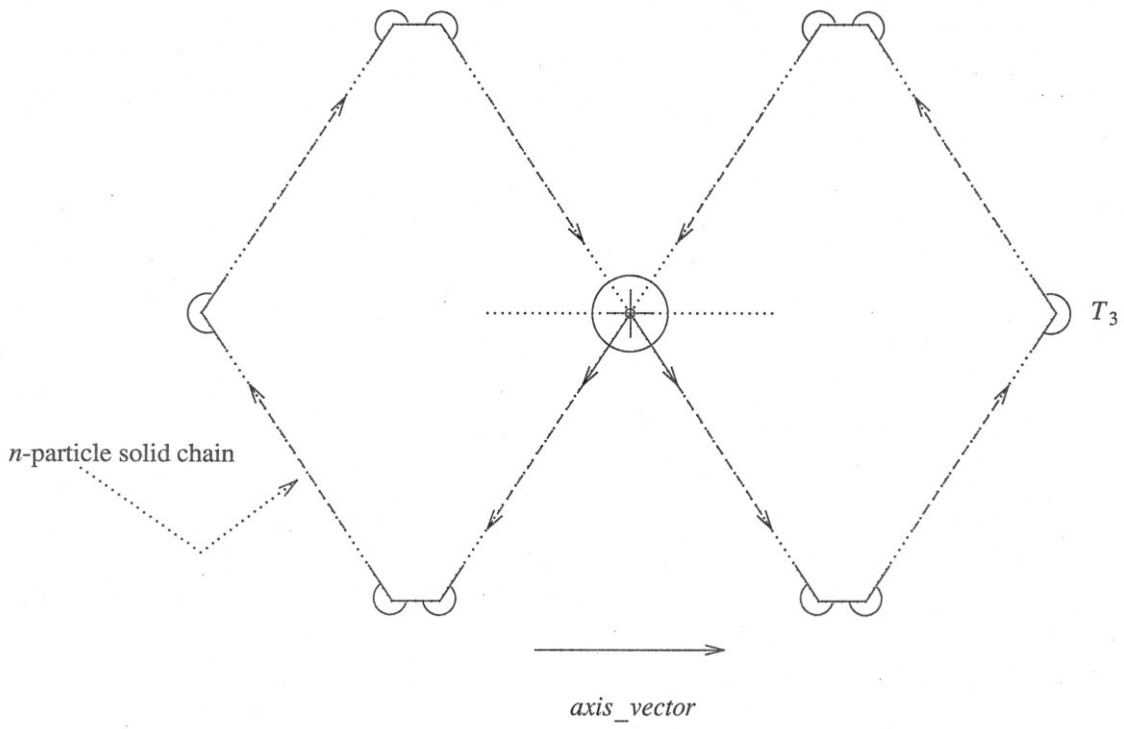
H_2(*origin*, *axis_vector*, *n*)



n is the number of test particles in the pattern. For H_2 the test particles are rest-particle / single-particle combinations. H_1 is the same except the rest particles are missing.

F_1.

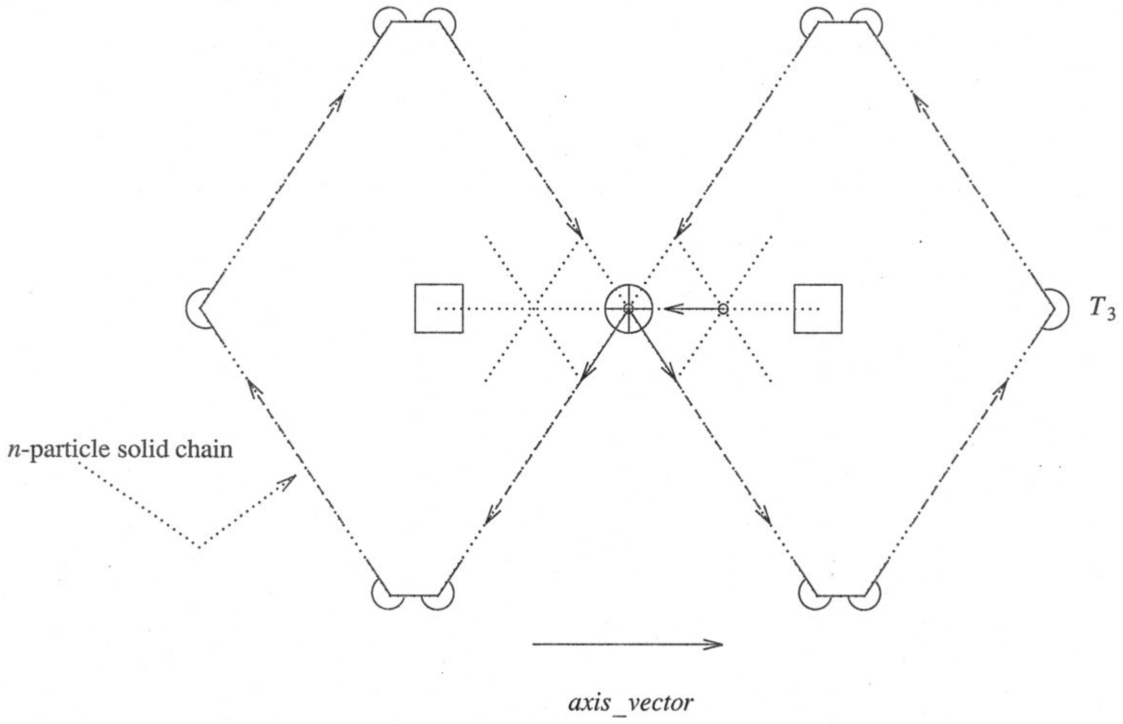
F_1(*origin*, *axis_vector*, *n*)



F_3 is identical except the rest-particle is missing from the origin.

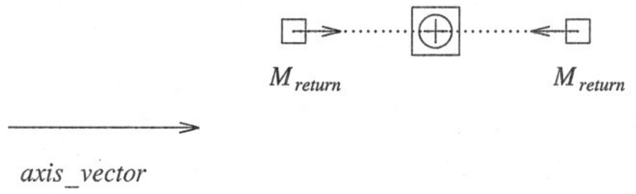
F_2.

F_2(*origin*, *axis_vector*, n)

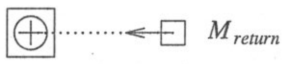


G_1.

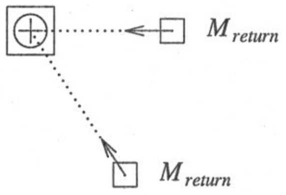
G_1(*origin*, *axis_vector*)



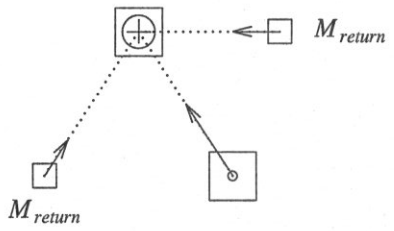
G_2.



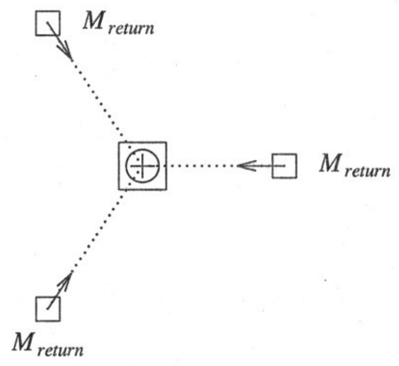
G_3.



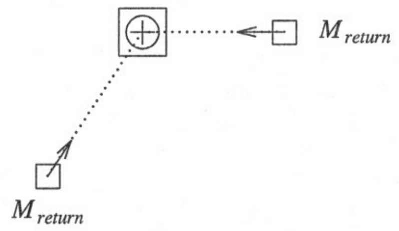
G_4.



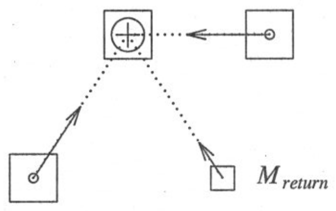
G_5.



G_6.

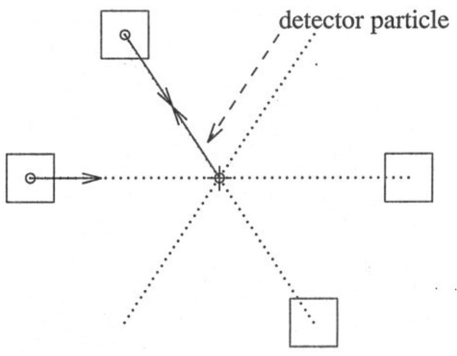


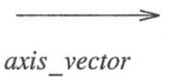
G_7.



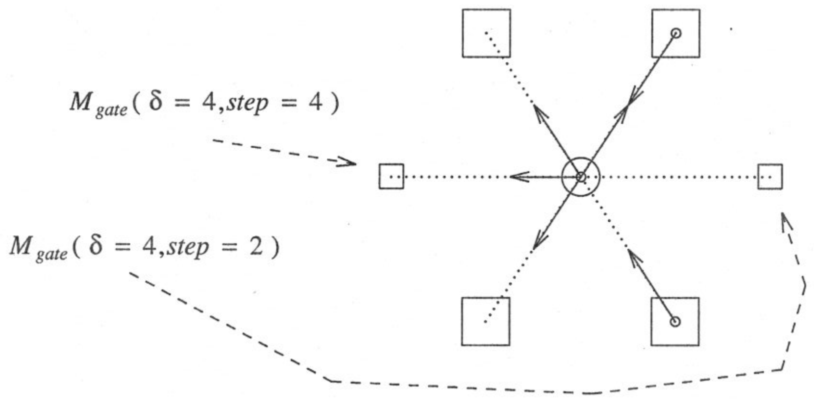
K_1.

K_1(*origin*, *axis_vector*)



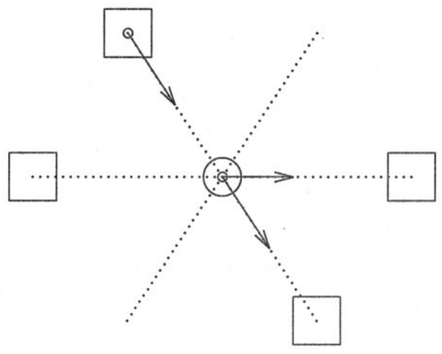


K_2.

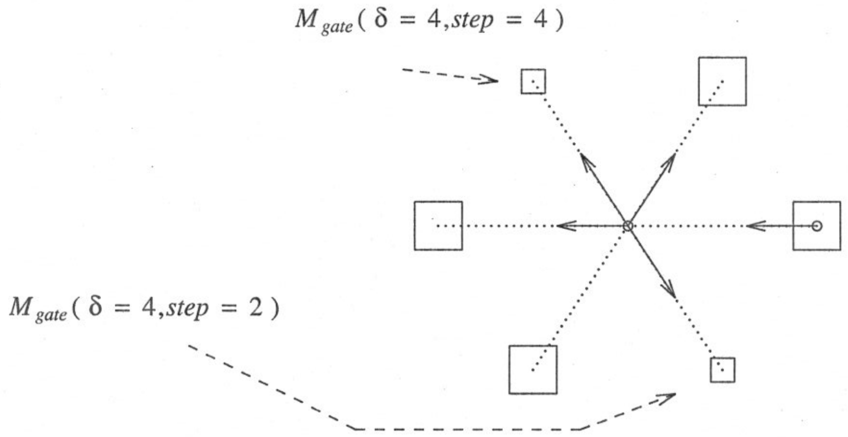


$M_{gate}(\delta = 4, step = 4)$
$M_{gate}(\delta = 4, step = 2)$

K_3.

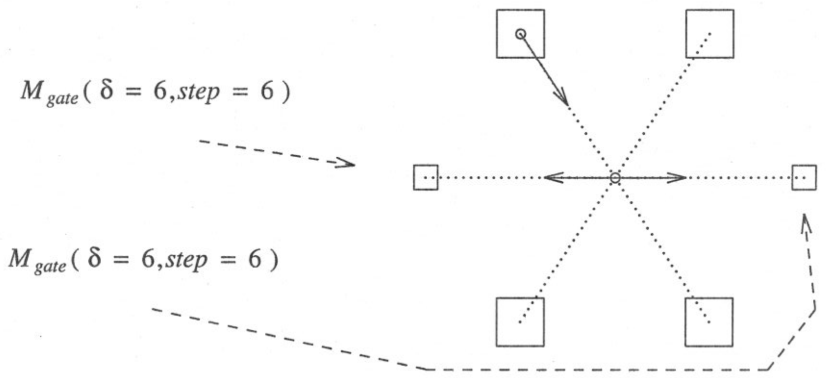


K_4.



$M_{gate}(\delta = 4, step = 4)$
$M_{gate}(\delta = 4, step = 2)$

K_5.



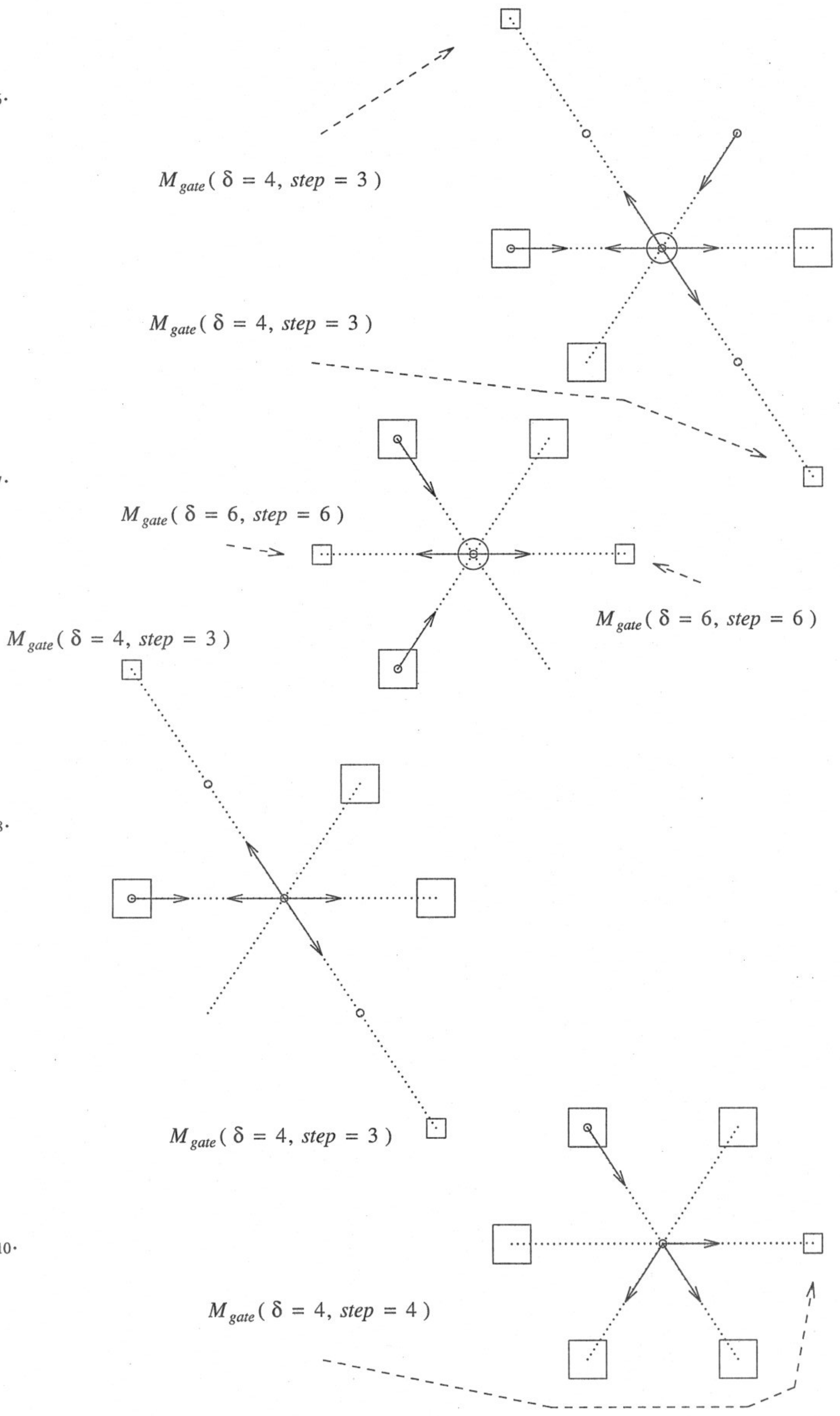
$M_{gate}(\delta = 6, step = 6)$
$M_{gate}(\delta = 6, step = 6)$

K_6.

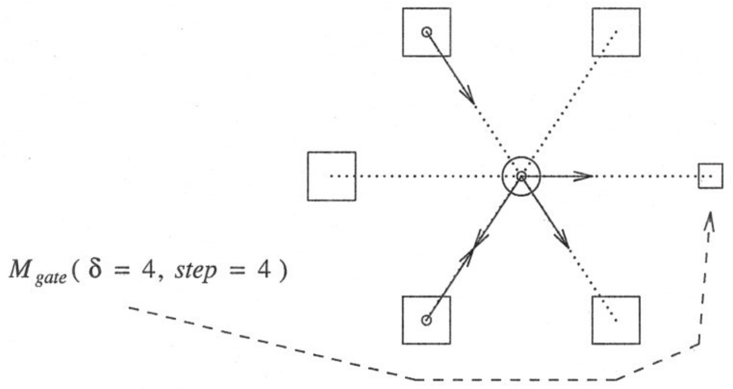
K_7.

K_8.

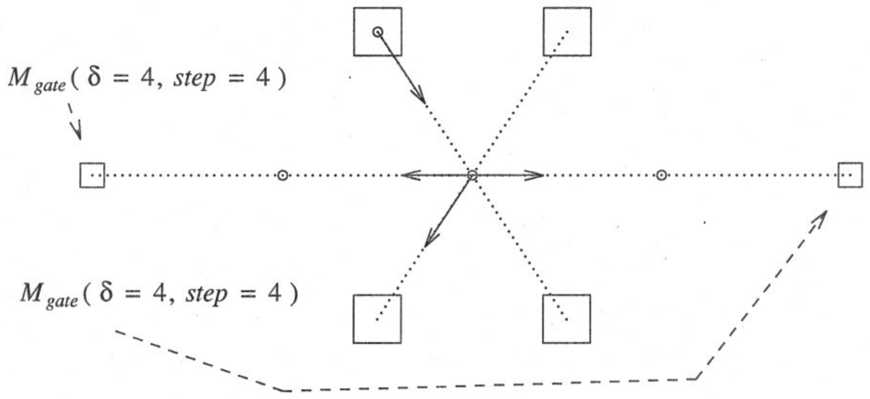
K_{10}.


$M_{gate}(\delta = 4, step = 3)$
$M_{gate}(\delta = 4, step = 3)$
$M_{gate}(\delta = 6, step = 6)$
$M_{gate}(\delta = 6, step = 6)$
$M_{gate}(\delta = 4, step = 3)$
$M_{gate}(\delta = 4, step = 3)$
$M_{gate}(\delta = 4, step = 4)$

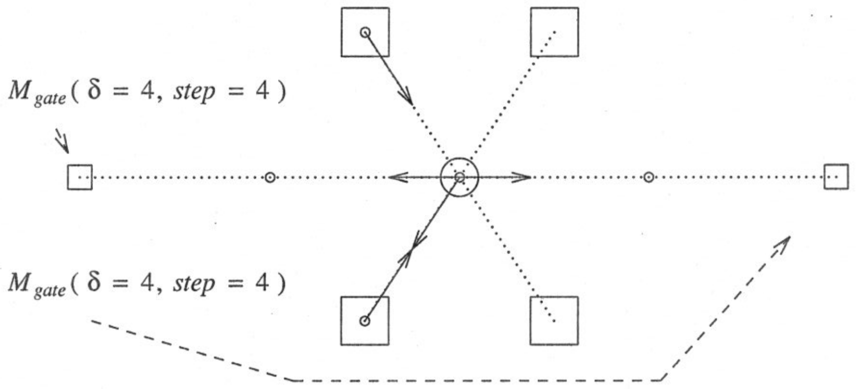
K_{11}.



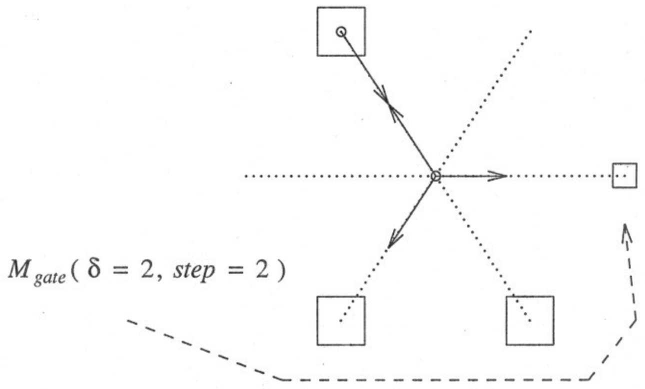
K_{12}.


$M_{gate}(\ \delta = 4,\ step = 4\ )$
$M_{gate}(\ \delta = 4,\ step = 4\ )$

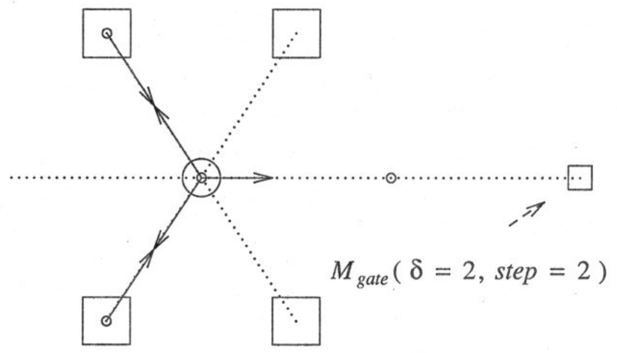
K_{13}.


$M_{gate}(\ \delta = 4,\ step = 4\ )$
$M_{gate}(\ \delta = 4,\ step = 4\ )$

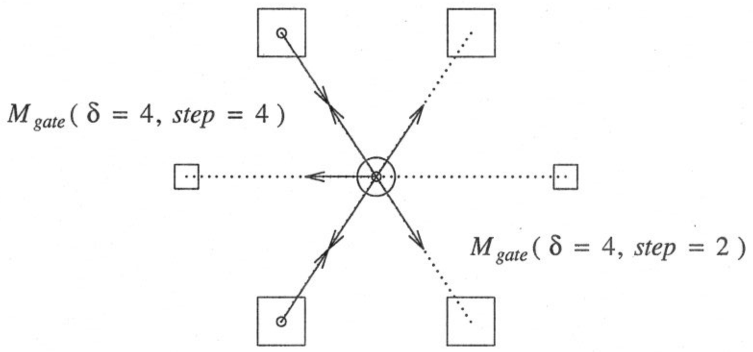
K_{14}.



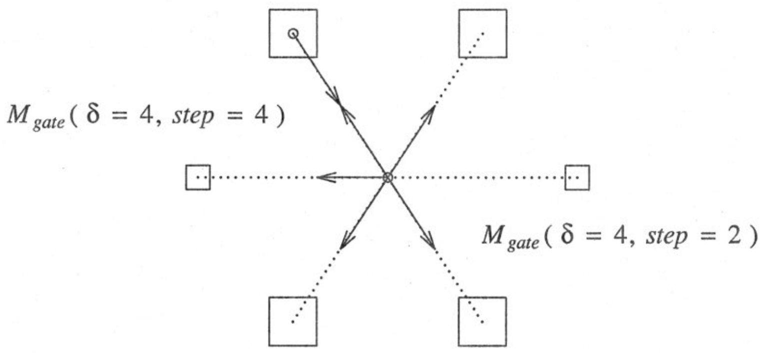
K_{15}.



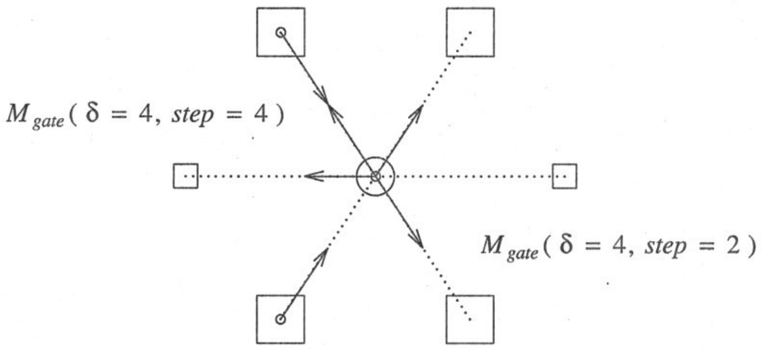
K_{16}.


M_{gate}(δ = 4, *step* = 4)
M_{gate}(δ = 4, *step* = 2)

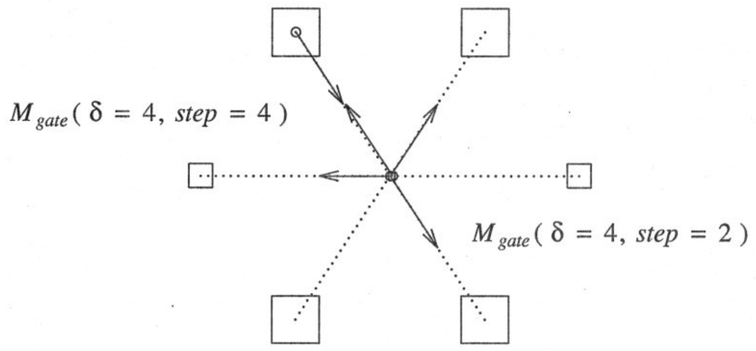
K_{17}.


M_{gate}(δ = 4, *step* = 4)
M_{gate}(δ = 4, *step* = 2)

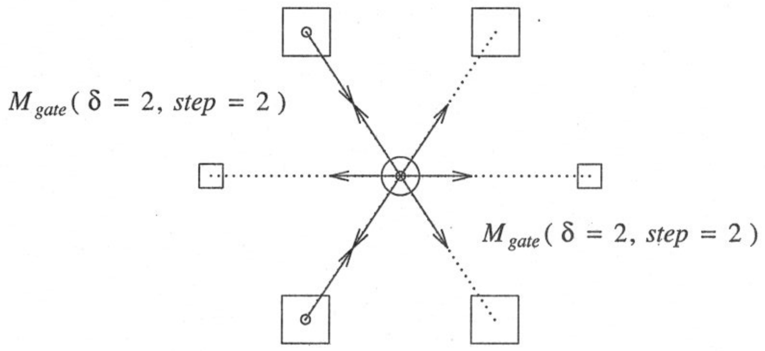
K_{18}.


M_{gate}(δ = 4, *step* = 4)
M_{gate}(δ = 4, *step* = 2)

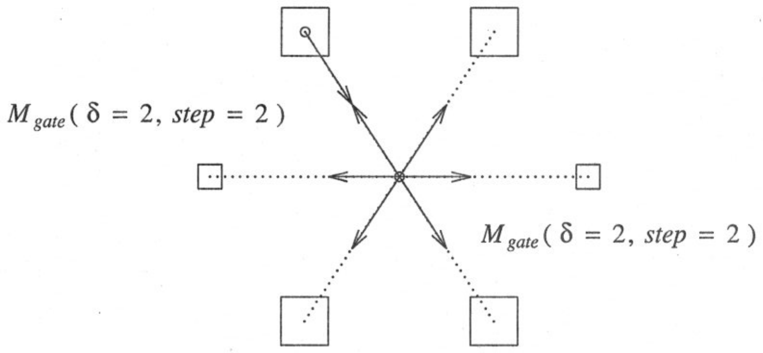
K_{19}.


$M_{gate}(\delta = 4, step = 4)$
$M_{gate}(\delta = 4, step = 2)$

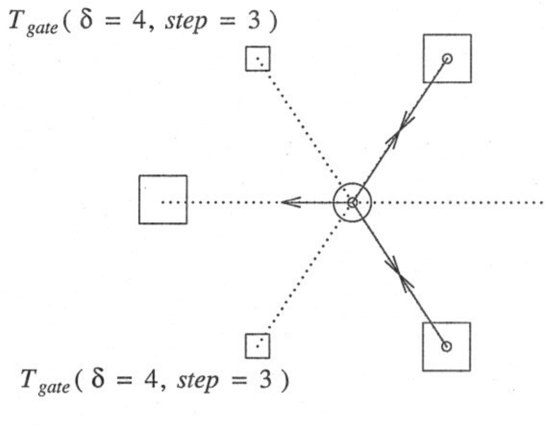
K_{20}.


$M_{gate}(\delta = 2, step = 2)$
$M_{gate}(\delta = 2, step = 2)$

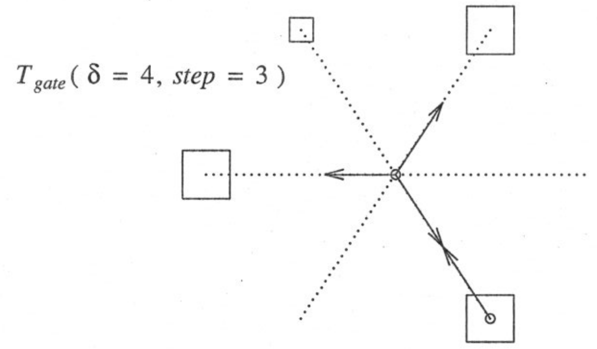
K_{21}.


$M_{gate}(\delta = 2, step = 2)$
$M_{gate}(\delta = 2, step = 2)$

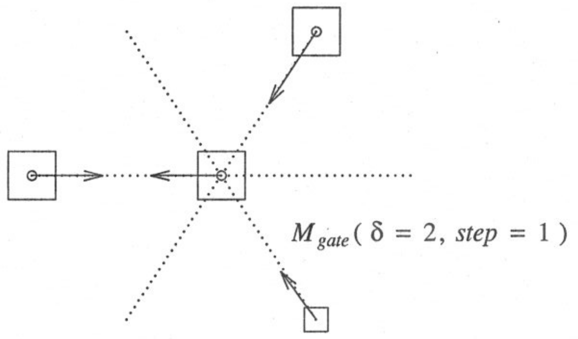
K_{22}.


$T_{gate}(\delta = 4, step = 3)$
$T_{gate}(\delta = 4, step = 3)$

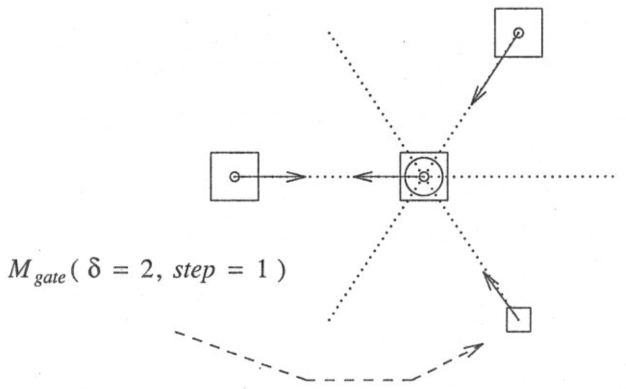
K_{23}.



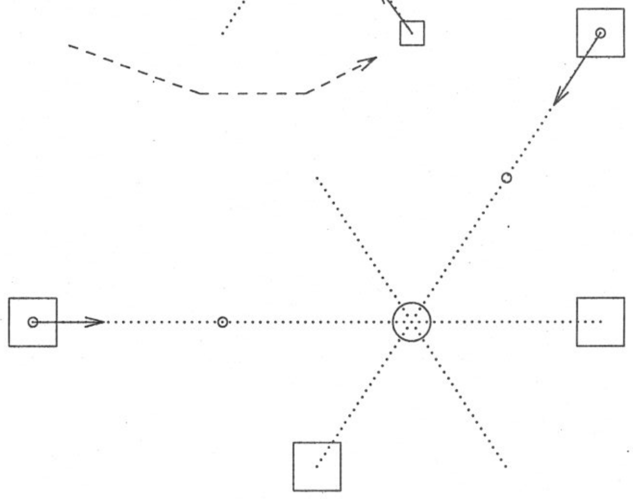
K_{24}.



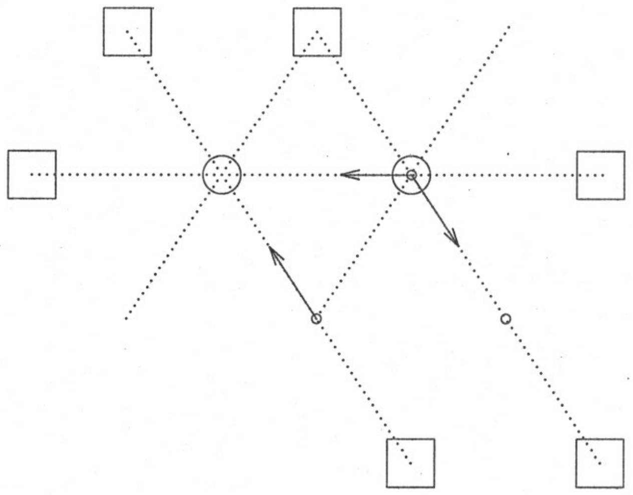
K_{25}.



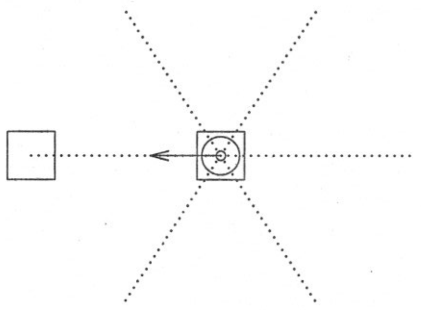
K_{26}.



K_{27}.



K_{28}.



References

[1] P. W. Kasteleyn, ''Graph Theory and Crystal Physics,'' in *Graph Theory and Theoretical Physics*, ed. F. Harary, Academic Press, New York, 1967.

[2] J. Hardy, O. de Pazzis, and Y. Pomeau, ''Molecular dynamics of a classical lattice gas: Transport properties and time correlation functions,'' *Physical Review A*, vol. 13, no. 5, pp. 1949-1961, May 1976.

[3] U. Frisch, D. d'Humiéres, B. Hasslacher, P. Lallemand, Y. Pomeau, and J. P. Rivet, ''Lattice Gas Hydrodynamics in Two and Three Dimensions,'' *Complex Systems*, vol. 1, pp. 649-707, 1987.

[4] John von Neumann, ''Theory of Natural and Artificial Automata,'' in *Papers of John von Neumann on Computing and Computer Theory*, ed. William Aspray and Arthur Burks, Charles Babbage Institute Reprint Series for the History of Computing, vol. 12, pp. 363-491, MIT Press, Cambridge, Massachusetts, 1987.

[5] T. Toffoli and N. Margolus, *Cellular Automata Machines: A New Environment for Modeling,* MIT Press, Cambridge, Massachusetts, 1987.

[6] Kendall Preston, Jr. and Michael J. B. Duff, *Modern Cellular Automata: Theory and Applications,* Plenum Press, New York, 1984.

[7] *Proceedings of the Int. Conf. on Application Specific Array Processors,* IEEE Computer Society Press, Los Alamitos, California, September 5-7, 1990.

[8] A. Clouqueur and D. d'Humiéres, ''RAP1, a Cellular Automata Machine for Fluid Dynamics,'' *Complex Systems*, vol. 1, pp. 585-597, 1987.

[9] W. Daniel Hillis, ''The Connection Machine: A Computer Architecture Based on Cellular Automata,'' in *Physica 10D*, pp. 213-228, North-Holland, Amsterdam, 1984.

[10] S. D. Kugelmass and K. Steiglitz, ''A Scalable Architecture for Lattice-Gas Simulations,'' *J. Comp. Phys.*, vol. 84, pp. 311-325, 1989.

[11] Derek L. Eager, John Zahorjan, and Edward D. Lazowska, ''Speedup Versus Efficiency in Parallel Systems,'' Technical Report 86-08-01, Department of Computer Science, University of Washington, Seattle, WA 98195, August, 1986.

[12] Alok Aggarwal, Bowen Alpern, Ashok K. Chandra, and Mark Snir, ''A Model for Hierarchical Memory,'' *Proc. 19-th Ann. ACM Symp. on Theory of Computing*, pp. 305-314, New York, NY, 25-27 May, 1987.

[13] Alok Aggarwal, Ashok K. Chandra, and Marc Snir, ''Hierarchical Memory with Block Transfer,'' *28-th Ann. Symp. on the Foundations of Computer Science*, IEEE, 12-14 October 1987.

[14] Alok Aggarwal and Jeffrey Scott Vitter, ''The Input/Output Complexity of Sorting and Related Problems,'' Technical Report CS-87-10, Brown University, Department of Computer Science, Providence, RI 02912, August, 1987.

[15] Christos H. Papadimitriou and Jeffrey D. Ullman, ''A Communication-Time Tradeoff,'' *SIAM J. Comput.*, vol. 16, no. 4, pp. 639-646, August, 1987.

[16] László Lovász and Michael Saks, ''Lattices, Mõbius Functions and Communication Complexity,'' *29-th Ann. Symp. on the Foundations of Computer Science*, IEEE, 24-26 October 1988.

[17] T. Lengauer and R. E. Tarjan, ''Upper and Lower Bounds on Time-Space Tradeoffs,'' *ACM Symp. on the Theory of Computing*, pp. 262-277, Atlanta, GA, 1979.

[18] Thomas Lengauer and Robert E. Tarjan, ''Asymptotically Tight Bounds on Time-Space Trade-offs in a Pebble Game,'' *Journ. of the ACM*, vol. 29, no. 4, pp. 1087-1130, Association for Computing Machinery, October 1982.

[19] N. Pippenger, ''Pebbling,'' *Proc. 5th IBM Symposium on Mathematical Foundations of Computer Science*, Academic and Scientific Programs, IBM Japan, May 1980.

[20] H.T. Kung, ''Why Systolic Architectures?,'' *Computer*, vol. 15, no. 1, pp. 37-46, IEEE, Jan. 1982.

[21] Jia-Wei Hong and H. T. Kung, ''I/O Complexity: The Red-Blue Pebble Game,'' *Proceedings of ACM Sym. Theory of Computing*, pp. 326-333, 1981.

[22] John E. Savage and Jeffrey Scott Vitter, ''Parallelism in Space-Time Tradeoffs,'' in *VLSI: Algorithms and Architectures*, ed. F. Luccio, pp. 49-58, Elsevier Science Publishers B.V. (North Holland), 1985.

[23] Steven D. Kugelmass, Richard Squier, and Kenneth Steiglitz, ''Performance of VLSI Engines for Lattice Computations,'' *Complex Systems*, pp. 939-965, 1987.

[24] Mark Nodine and Daneil P. Lopresti, ''I/O Overhead in Parallel VLSI Architectures for ''Life'','' in *unpublished manuscript*, Department of Computer Science, Brown University, Providence, Rhode Island, 1988.

[25] Martin Gardner, ''On Cellular Automata, Self-Reproduction, the Garden of Eden and the Game of `Life´,'' *Scientific American*, vol. 224, no. 2, pp. 112-117, 1971.

[26] S. Fortune and J. Wyllie, ''Parallelism in Random Access Machines,'' *Proc. 10th Annual ACM Symp. on the Theory of Computing*, San Diego, CA, 1978.

[27] Béla Bollobás and Imre Leader, ''Isoperimetric Inequalities and Fractional Set Systems,'' *J. of Combinatorial Theory (A)*, to appear.

[28] Béla Bollobás and Imre Leader, ''An Isoperimetric Inequality on the Discrete Torus,'' *SIAM J. Discrete Math*, vol. 3, pp. 32-37, (1990).

[29] D. L. Wang and P. Wang, ''Discrete Isoperimetric Problems,'' *SIAM J. Appl. Math.*, vol. 32, pp. 860-870, (1977).

[30] D. L. Wang and P. Wang, ''Extremal Configurations on a Discrete Torus and a Generalization of the Generalized Macaulay Theorem,'' *SIAM J. Appl. Math.*, vol. 33, pp. 55-59, (1977).

[31] Jeane E. Taylor, ''Crystalline Variational Problems,'' *Bulletin of the Am. Math. Soc.*, vol. 84, no. 4, pp. 568-588, July 1978.

[32] Frank Morgan, *Geometric Measure Theory, A Beginner's Guide,* Academic Press, Inc., San Diego, California, 1988.

[33] Lawrence Snyder, ''Type Architectures, Shared Memory, and the Corollary of Modest Potential,'' *Ann. Rev. Comput. Sci.*, no. 1, pp. 289-317, Annual Reviews Inc., 1986.

[34] S. Y. Kung, *VLSI Array Processors,* Prentice Hall, Englewood Cliffs, New Jersey, 1988.

[35] Juan J. Navarro, José M. Llaberia, and Mateo Valero, ''Partitioning: An Essential Step in Mapping Algorithms Into Systolic Array Processors,'' *Computer*, vol. 20, no. 7, pp. 77-90, July 1987.

[36] Jaime H. Moreno and Tomás Lang, ''Matrix Computations on Systolic-Type Meshes: An Introduction to the Multimesh Graph Method,'' *Computer*, vol. 23, no. 4, pp. 32-52, April 1990.

[37] Alan V. Oppenheim and Ronald W. Schafer, *Digital Signal Processing,* pp. 113-114, Prentice Hall, Englewood Cliffs, New Jersey, 1975.

[38] Takao Nishitani et al, ''Parallel Video Signal Processor Configuration based on Overlap-Save Technique and its LSI Processor Element: VISP,'' *J. of VLSI Signal Processing*, vol. 1, no. 1, pp. 25-35, Kluwer Academic Publishers, August, 1989.

[39] Tien Chi Chen, ''Parallism, Pipelining, and Computer Efficiency,'' *Computer Design*, pp. 69-74, January, 1971.

[40] Stanley R. Sternberg, ''Pipeline Architectures For Image Processing,'' in *Multicomputers and Image Processing, Algorithms and Programs*, ed. Leonard Uhr, pp. 291-305, Academic Press, 1982.

[41] C. V. Ramamoorthy and H. F. Li, ''Pipeline Architecture,'' *ACM Computing Surveys*, vol. 9, no. 1, pp. 61-102, March, 1977.

[42] John L. Hennessy and David A. Patterson, *Computer Architecture: a quantitative approach,* pp. 52-66, Morgan Kaufmann Publishers, Inc., San Mateo, California, 1990.

[43] U. Frisch, B. Hasslacher, and Y. Pomeau, ''Lattice-gas Automata for the Navier-Stokes Equation,'' *Phys. Rev. Lett.*, vol. 56, pp. 1505-1508, 1986.

[44] D. d'Humiéres, P. Lallemand, and U. Frisch, ''Lattice Gas Models for 3D Hydrodynamics,'' *Europhysics Letters*, vol. 4, no. 2, 15 August 1986.

[45] D. d'Humiéres, P. Lallemand, and T. Shimomura, ''Lattice Gas Cellular Automata, A New Experimental Tool For Hydrodynamics,'' Preprint LA-UR-85-4051, Los Alamos National Laboratory, Los Alamos, New Mexico, 1985.

[46] S. Salem, J. Salem, and S. Wolfram, ''Thermodynamics and Hydrodynamics with Cellular Automata,'' in *Theory and Applications of Cellular Automata*, vol. 1, World Scientific, Singapore, 1987.

[47] F. Hayot, M. Mandal, and P. Sadayappan, ''Implementation and Performance of a Binary Lattice Gas Algorithm on Parallel Processor Systems,'' *J. Comp. Phys.*, vol. 80, no. 2, February 1989.

[48] T. Shimomura, G. D. Doolen, B. Hasslacher, and C. Fu, ''Calculations Using Lattice Gas Techniques,'' *Los Alamos Science*, vol. 15, pp. 201-210, 1987.

[49] D. d'Humiéres and P. Lallemand, ''Numerical Simulations of Hydrodynamics with Lattice Gas Automata in Two Dimensions,'' *Complex Systems*, vol. 1, pp. 599-632, 1987.

[50] L. Kadanoff, G. McNamara, and G. Zanetti, ''A Poiseuille Viscometer for Lattice Gas Automata,'' *Complex Systems*, vol. 1, pp. 791-803, 1987.

[51] H. Lim, ''Lattice Gas Automata of Fluid Dynamics for Unsteady Flow,'' *Complex Systems*, vol. 2, pp. 45-58, 1988.

[52] H. Chen, S. Chen, G. Doolen, and Y. C. Lee, ''Simple Lattice Gas Models for Waves,'' *Complex Systems*, vol. 2, pp. 259-267, 1988.

[53] R. Benzi and S. Succi, ''Bifurcations of a Lattice Gas Flow under External Forcing,'' *J. Stat. Phys.*, vol. 56, no. 1/2, 1989.

[54] P. Binder, ''Abnormal Diffusion in Wind-tree Lattice Gasses,'' *Complex Systems*, vol. 3, pp. 1-7, 1989.

[55] N. Margolus and T. Toffoli, ''Cellular Automata Machines,'' *Complex Systems*, vol. 1, pp. 967-993, 1987.

[56] Thomas J. Ostrand and Marc J. Balcer, ''The Category-Partition Method for Specifying and Generating Functional Tests,'' *Communications of the ACM*, vol. 31, no. 6, pp. 676-686, June 1988.

[57] David Gelperin and Bill Hetzel, ''The Growth of Software Testing,'' *Communications of the ACM*, vol. 31, no. 6, June 1988.

[58] Richard Squier and Kenneth Steiglitz, ''Testing Parallel Simulators for Two-Dimensional Lattice-Gas Automata,'' Tech. Report CS-TR-269-90, Princeton University, Computer Science Department, 35 Olden Street, Princeton, NJ 08544-2087, June 1990.

[59] David P. Dobkin and Eleftherios Koutsofios, ''The Cheyenne Graphics Library,'' *unpublished internal documentation*, Computer Science Department, Princeton University.

[60] Steven A. Orszag and Victor Yakhot, ''Reynolds Number Scaling of Cellular Automaton Hydrodynamics,'' *Physical Review Letters*, vol. 56, no. 16, pp. 1691-1693, April 21, 1986.

[61] Richard Squier and Kenneth Steiglitz, ''A Practical Runtime Test Method for Parallel Lattice-Gas Automata,'' in *Proceedings of the International Conference on Application Specific Array Processors (ASAP90)*, IEEE, Princeton, New Jersey, September 5-7, 1990.