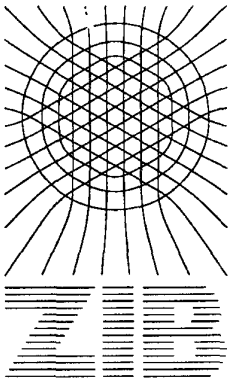
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Str. 10, D-1000 Berlin 31



Hans-Christian Hege Hinnerk Stüben*

Vectorization and Parallelization of Irregular Problems via Graph Coloring

* Freie Universität Berlin
Institut für Physik
Arnimallee 14
D-1000 Berlin 33

Preprint SC 90-16 (December 1990)

Herausgegeben vom
Konrad-Zuse-Zentrum für Informationstechnik Berlin
Heilbronner Str. 10
1000 Berlin 31
Verantwortlich: Dr. Klaus André
Umschlagsatz und Druck: Rabe KG Buch-und Offsetdruck Berlin

ISSN 0933-7911

Abstract

Efficient implementations of irregular problems on vector and parallel architectures generally are hard to realize. An important class of irregular problems are Gauß-Seidel iteration schemes applied to irregular data sets. The unstructured data dependences arising there prevent restructuring compilers from generating efficient code for vector or parallel machines. It is shown, how to structure the data dependences by decomposing the data set using graph coloring techniques and by specifying a particular execution order already on the algorithm level. Methods to master the irregularities originating from different types of tasks are proposed. An example of application is given and possible future developments are mentioned.

Contents

1 Introduction

There is a threefold way for shifting the complexity barrier in large scale computing: development of fast architectures, design of fast algorithms suited to various architectures, and invention of advanced languages, methods and tools for expressing algorithms and mapping these to machines. Because of physical limitations, future increases of computer power will be possible mainly by relying on parallel architectures. Therefore, exploitation of parallelism becomes the major task from the algorithmic and software point of view.

Parallelism appears at three different levels: the algorithm level, the program level and the machine level [1]. *Algorithm level parallelism* can be characterized by the number of variables, the functional dependences between the variables, the complexity of basic algorithmic operations, and the precedence constraints on the order of operations. *Program level parallelism* can be characterized by the control and data dependence constraints, imposed by the programmers choice of data structures and control structures and their semantics. *Machine level parallelism* can be characterized by the use of the various machine features enabling pipelined and concurrent execution.

The major hindrances to parallel execution are data dependences, memory access, and communication delays: an instruction cannot begin execution until its operands are available. This is valid for today's as well as future architectures. We concentrate upon *data dependences*. Control dependences also cause problems, but at least in numerical problems, to which we focus here, these are of secondary importance. Therefore, the main goals are the design of algorithms containing less, and more regular data dependences, the development of methods for mapping algorithms to program and machine level without introducing too many additional data dependences, and architectures handling data dependences efficiently.

Common programming techniques, like reassignment of variables, introduce additional data dependences. The lack or limited availability of parallel constructs in currently most wide-spread languages often force to serialize concurrent parts of algorithms. With the advent of powerful data dependence analysis and program restructuring techniques [2–13] and their implementation into several commercial and experimental restructuring compilers[1] this displeasing situation is partially remedied. Modern translators analyse data dependences, remove some 'artificial' data dependences, and restructure the code, generating specific control and data structures better suited for the target machines. Although many features, like interprocedural dependence analysis, are still missing in commercial compilers and despite the use of un-

[1]Examples are the VAST (Pacific Sierra Research), KAP (Kuck & Associates), PTRAN (IBM), PFC and PTOOL (Rice University), Parafrase and Parafrase-2 (University of Illinois).

suitable programming languages, one may hope that future compilers will uncover and utilize parallelism.

Today's compilers are able to detect and exploit *structured parallelism*, represented by regular data dependence patterns, for example shift-invariant iteration space dependence graphs. A more difficult problem is the exploitation of *unstructured parallelism* with different instruction and/or data streams in single tasks. This kind of parallelism is very common in scientific computing, like in computations on irregular or even adaptive grids, or in sparse matrix calculations with unpredictable fill-in. Modern algorithms, that adapt themselves more flexibly to the structure of the problem, lead to further unstructured dependences. Mostly, these cannot be inferred from the information available at compile-time. One approach therefore is run-time dependence checking (see [7, 9] and references therein). This is essentially the main idea behind the data flow concept [16]. For run-time dependence checking the compiler generates extra code to resolve dependences at run-time that are not amenable to analysis at compile-time. This might possibly be supported by specialized hardware components. Up to the data flow efforts such techniques are rarely employed yet. Real problems presently solved on concurrent processors usually are rather regular [14, 15]. Sophisticated adaptive algorithms are seldom employed.

The object of this paper is to demonstrate how an important class of algorithms in scientific computing—the *iterative algorithms* employing a *Gauß-Seidel update scheme* on *irregular data structures*—can be vectorized and parallelized. This is achieved by choosing a particular order of execution and thereby structuring the data dependences already on the algorithm level. A set of possible orders of execution is constructed by coloring the nodes of a given or imagined underlying graph. The set of tasks can be partitioned into equal-sized microtasks, that are particularly well suited for processing on shared memory vector or fine grained SIMD machines, or into coarser grains that are favorable on distributed memory machines.

2 Preliminaries

2.1 Graphs

Before we start with the actual subject we establish our basic notation.

A *graph* $G(V, E)$ is a pair of a finite set V and a set E of two elementary subsets of V. The elements of V are called *vertices* and those of E *edges*. The elements of an edge $e = \{u, v\} \in E$ are called *endpoints* of e; u and v are called *adjacent* or *connected*. The *neighborhood* $N(v)$ of a vertex v is the set of vertices that are adjacent to v. A *directed graph* is defined similar to a graph, except that the elements of E are now ordered pairs $e = (a, b) \in E$,

$a, b \in V$. These pairs are called *arcs*. Let $X \subseteq V$ and let $E \mid X$ denote the set of all edges having both endpoints in X. Then the graph $(X, E \mid X)$ is called *induced subgraph* of $G(V, E)$.

The cardinality of a set M is written as $|M|$. The *degree* $\deg v$ of a vertex v is the cardinality of its neighborhood. We will use $n = n(G)$ to denote the cardinality of V. $\delta(G) = \min_{v \in V} \deg v$ and $\Delta(G) = \max_{v \in V} \deg v$ denote the minimal or maximal degree of all vertices.

A subset $X \subset V$ is called *independent* if not any two elements of X are adjacent. A *k-coloring* of a graph G is a partition of G into k independent subsets. G is *k-colorable* if a k-coloring of G exists. The *chromatic number* $\chi(G)$ of G is the minimal k for which G is k-colorable.

2.2 Data Dependences

To define data dependences in a program, we need the following terms.

A *program* is a sequence of $l \in N$ statements. S_m denotes the m-th statement in the program, counted in lexicographic order. If a statement S_m can be executed multiply, e. g., if S_m is enclosed in d nested loops, we conceive it as explicitely or implicitely controlled by a multi-index like $(I_1, I_2, \ldots, I_d)$. We write $S_m(i_1, i_2, \ldots, i_d)$ to refer to the *instance of* S_m during the particular iteration step[2] when $I_1 = i_1, I_2 = i_2, \ldots, I_d = i_d$. As program model we take a Fortran or C loop, denoted as a **for**-loop. **for**-loops are supposed to be executed sequentially. This imposes constraints on the order of memory references. The *execution order of statements* is a relation $\leq$ defined between pairs of instances. We write $S_m \leq S_n$ for non-indexed statements, if S_m can be executed before S_n, and for indexed statements $S_m(i_1, \ldots, i_d) \leq S_n(j_1, \ldots, j_d)$, if instance $S_m(i_1, \ldots, i_d)$ can be executed before instance $S_n(j_1, \ldots, j_d)$.

By $\mathrm{IN}(S_m)$ and $\mathrm{OUT}(S_m)$ we denote the set of variables that are read and written by the non-indexed statement S_m and by $\mathrm{IN}(S_m(i_1, \ldots, i_d))$ and $\mathrm{OUT}(S_m(i_1, \ldots, i_d))$ the set of variable instances defined or used by the statement instance $S_m(i_1, \ldots, i_d)$.

Data dependences arise if IN and OUT sets of two statement instances intersect. One defines [2]:

- S_n is *data flow-dependent* on S_m, written $S_m \delta S_n$, iff there exist index values $(i_1, \ldots, i_d)$ and $(j_1, \ldots, j_d)$, such that $S_m(i_1, \ldots, i_d) \leq S_n(j_1, \ldots, j_d)$, and $\mathrm{OUT}(S_m(i_1, \ldots, i_d)) \cap \mathrm{IN}(S_n(j_1, \ldots, j_d)) \neq \emptyset$.

- S_n is *data anti-dependent* on S_m, written $S_m \bar{\delta} S_n$, iff there exist index values $(i_1, \ldots, i_d)$ and $(j_1, \ldots, j_d)$, such that $S_m(i_1, \ldots, i_d) \leq S_n(j_1, \ldots, j_d)$, and $\mathrm{IN}(S_m(i_1, \ldots, i_d)) \cap \mathrm{OUT}(S_n(j_1, \ldots, j_d)) \neq \emptyset$.

[2]This term collides with the same used in numerical mathematics. For the later we will introduce the notion of a *sweep* in section 5.1.

- S_n is *data output-dependent* on S_m, written $S_m \delta^\circ S_n$, iff there exist index values $(i_1, \ldots, i_d)$ and $(j_1, \ldots, j_d)$, such that $S_m(i_1, \ldots, i_d) \leq S_n(j_1, \ldots, j_d)$, and $\mathrm{OUT}(S_m(i_1, \ldots, i_d)) \cap \mathrm{OUT}(S_n(j_1, \ldots, j_d)) \neq \emptyset$.

The dependences of a program P are comprised in the *program data dependence graph* $G_P(V, E)$; it is a directed labeled graph with the set of nodes $V = \{S_1, S_2, \ldots, S_l\}$ corresponding to the statements of the program, and the set of labeled arcs $E = \{(S_m, S_n)_{\hat{\delta}} | S_n \hat{\delta} S_m\}$, where the label $\hat{\delta}$ is one of the relations δ, $\bar{\delta}$ or δ°.

Focusing on loops, it is worthwhile to display the dependences in more detail by looking at single instances. The *iteration space* of d nested loops is a set $I \subset \boldsymbol{Z}^d$, containing all possible indices $(i_1, i_2, \ldots, i_d)$. Unrolling such a loop nest completely, i. e., transforming the loop to a set of sequential statements with each instance written out explicitely, the resulting data dependence graph is called *iteration space dependence graph.*

The execution order and the intersection sets of two statements may depend on the data values used in a particular run. On compile-time, testing for execution order and set intersection must be conservative, i. e., dependence relations have to be assumed if the opposite cannot be proven.

It is not necessary to execute statements in the 'normal' execution order defined by the source program. All execution sequences not violating the dependences lead to the same result (up to rounding errors in floating point operations). The dependences only fix a partial order on the execution of program statements. This is widely exploited in restructuring, vectorizing, or parallelizing compilers [2, 3, 4, 9].

But this freedom is not always sufficient to utilize the inherent parallelism. Considering different execution orders already on the algorithm level, it is possible to structure the data dependences more flexibly.

3 Iteration Schemes

The general setting of iteration schemes is as follows. A set of variables $(x_i)_{i \in I}$, $I = \{1, 2, \ldots, n\}$ is updated according to a functional rule

$$x_i \leftarrow F_i((x_k)_{k \in K_i}), \ \forall i \in I$$

with operators F_i mapping variables $(x_k)_{k \in K_i}$, $K_i \subseteq I$ to variables x_i. The index sets K_i, $i \in I$ may have different cardinalities $|K_i|$, what constitutes part of the irregularity. Some index sets K_i may be empty and some operators F_i may be identical maps. For simplicity we restrict ourselves to cases where the index sets K_i and operators F_i do not vary during the whole iteration process.

Starting with initial values $(x_i^0)_{i \in I}$, invocation of *primitive tasks* $(F_i)_{i \in I}$ generates a set of values $\{x_i^l \,|\, i \in I, \ l = 1, 2, \ldots\}$. We assume that at any

point of time at most one task is working for an element $i \in I$. Therefore the values x_j^l can be labeled according to their time order. The value x_j^l is called the *l-th update* of variable x_j. To completely define the tasks, their arguments must be specified. Denoting the sets K_i as $\{k(i,1), k(i,2), \ldots, k(i,|K_i|)\}$, we write the task for the l-th update of x_i as

$$x_i^l \leftarrow F_i(x_{k(i,1)}^{l-d(l,i,1)}, x_{k(i,2)}^{l-d(l,i,2)}, \ldots, x_{k(i,|K_i|)}^{l-d(l,i,|K_i|)})$$

for all $i \in I$. The terms $d(l,i,k) \in \boldsymbol{Z}$ are referred to as *delays*. Specification of all delays and all conditions on the update order defines an *iteration scheme.*

An iteration scheme that imposes weak or no conditions on the delays is *asynchronous relaxation* or 'chaotic relaxation' [18]. It is characterized by the absence of most or all synchronizations between the tasks and therefore well adapted to deficiences of some concurrent architectures, like unpredictable communication overhead. The delays, which may originate from very different sources, like varying computing times, communication delays or memory access times, may be unpredictable and may take pretty large values. Since synchronization is a major source of performance degradation on concurrent architectures, it would be worthwhile to investigate the competitive power of asynchronous algorithms. Imposing rather weak conditions on the distribution of the delays and the update order, the correctness of such algorithms has been proven for important classes of problems [19, 20, 23].

But if the delays take too large values, the convergence rates seem to be questionable. The most important (partially) synchronous iteration schemes are:

- The *Jacobi iteration scheme*, defined by $d(l,i,k) = 1$, $\forall l \in \boldsymbol{N}$, $\forall i \in I$, $\forall k \in K_i$.
- The *Gauß-Seidel iteration scheme*, defined by the requirement that two tasks F_i and F_j may not be performed simultaneously if $i \in K_j$ or $j \in K_i$. The delays of a Gauß-Seidel iteration scheme are

$$d(l,i,k) = \left\{ \begin{array}{ll} 0 & \forall k \in M_i^l \\ 1 & \forall k \in K_i \backslash M_i^l \end{array} \right\}, \ \forall l \in \boldsymbol{N}, \ \forall i \in I$$

  where $M_i^l = \{k(i,m) \in K_i \mid \text{update value } x_{k(i,m)}^l \text{ has already been computed when starting the computation of } x_i^l\}$.

Although asynchronous relaxations seem to be a natural choice on some concurrent architectures, particularly for irregular problems, there are arguments for synchronous relaxations. Some of these are reliable convergence rates, the possibility to efficiently use synchronous architectures like pipelined vector processors, and—sometimes—lacking correctness in asychronous relaxations.

Implementation of a Jacobi iteration scheme on vector and parallel processors is straightforward, since it imposes no sequential order on updates of the values $(x_i^l)_{i\in I}$ for each l. Hence it can be carried out in parallel for each iteration step l. Unfortunately it is of less practical importance because it converges relatively slowly. Methods employing the Gauß-Seidel iteration scheme generally show much better convergence rates. This is due to the fact that Gauß-Seidel iteration schemes always use the newest available update values as input arguments, whereas Jacobi iterations exclusively use values of the previous update step.

4 Gauß-Seidel Iterations for Regular Problems

Gauß-Seidel iteration schemes are widely used. One significant Gauß-Seidel type method is successive over-relaxation (SOR) for solving large linear systems, that appear, for example, in finite difference methods for solving partial differential equations. Similar algorithms are employable on semi-rings for solving 'algebraic' path problems, which represent a large variety of problems in computer science (for a review see [17]). When simulating Markov-chains, the Markov property even enforces to use a Gauß-Seidel type update scheme.

The Gauß-Seidel iteration scheme is very easily implemented as a sequential update: choosing any permutation π_l of the indices $(1, 2, \ldots, n)$ as update sequence, i. e., computing updates in the order

$$x_{\pi_1(1)}, x_{\pi_1(2)}, \ldots, x_{\pi_1(n)}, x_{\pi_2(1)}, x_{\pi_2(2)}, \ldots, x_{\pi_2(n)}, \cdots ,$$

use of correct input arguments is simply achieved by physically storing all update values of a variable at the same memory location.

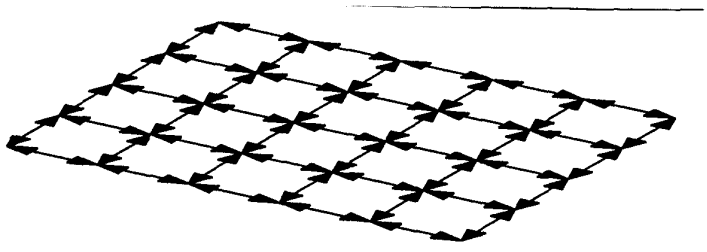


Figure 1: Functional dependences (represented by arrows) between variables on a square grid, defined by a five point stencil.

The implementation on vector and parallel processors is rendered more difficult by the requirement of non-adjacent concurrent updating and the condition to use the newest update values available. Let us illustrate this on a simple example: imagine the variables $(x_i)_{i\in I}$ associated with the nodes $i \in I$ of a regular square grid, and the sets K_i to contain the indices of the nearest neighbors on the grid. Ignoring boundary effects, any variable functionally depends only on its four nearest neighbors (illustrated by arrows in Fig. 1).

For simplicity we replace the index $i \in I$ by two indices for x- and y-direction, running from 1 to n_x and 1 to n_y. Choosing a definite update order by writing the sequential loops (see Fig. 2)

for $l = 1, 2, \ldots$
  for $j = 2$ **to** $n_y - 1$
    for $i = 2$ **to** $n_x - 1$
      $x_{ij} = F(x^l_{i-1,j}, x^{l-1}_{i+1,j}, x^l_{i,j-1}, x^{l-1}_{i,j+1})$

we get data flow-dependences between different instances of the innermost loop body. Part of the corresponding iteration space dependence graph is shown in Fig. 3.

The data flow-dependences in the $(l+1)$-plane prohibit naive vectorization or parallelization of the inner loop. A way out is to choose a different update order. A method that respects the data dependences originating from the update order in the above code is the hyperplane or wave-front method [21].

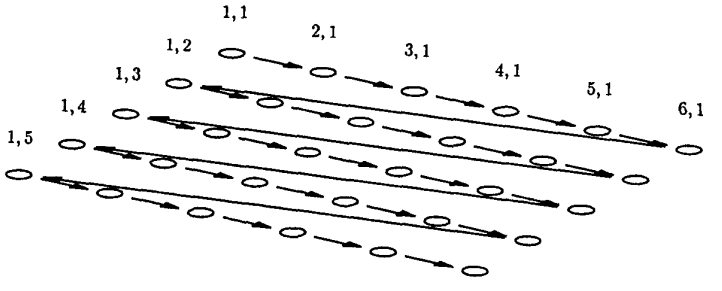


Figure 2: A possible update order on the square lattice, defined by sequential loops.

Using this prescription, nodes lying on a 'hyperplane' (see Fig. 4) are updated in parallel:

for $l = 1, 2, \ldots$
  for $i = 4$ **to** $n_x + n_y - 2$
    for $j = \max(2, i - n_x + 1)$ **to** $\min(n_y - 1, i - 2)$
      $x^l_{i-j,j} = F(x^l_{i-j-1,j}, x^{l-1}_{i-j+1,j}, x^l_{i-j,j-1}, x^{l-1}_{i-j,j+1})$

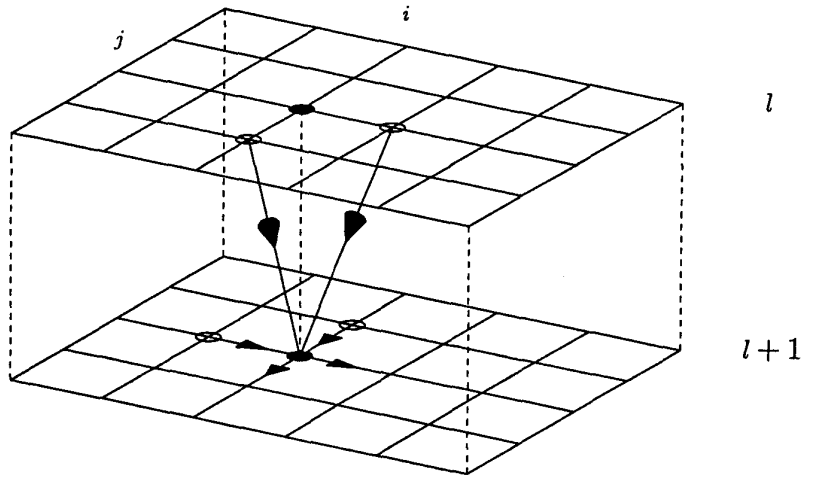


Figure 3: Data flow-dependences between statement instances arising from the functional dependences illustrated in Fig. 1 and the update order shown in Fig. 2. The dependences are $S_{i-1,j,l+1}\delta S_{i,j,l+1}$, $S_{i,j-1,l+1}\delta S_{i,j,l+1}$ between instances of sweep $l+1$, and $S_{i+1,j,l}\delta S_{i,j,l+1}$, $S_{i,j+1,l}\delta S_{i,j,l+1}$ between instances of successive sweeps.

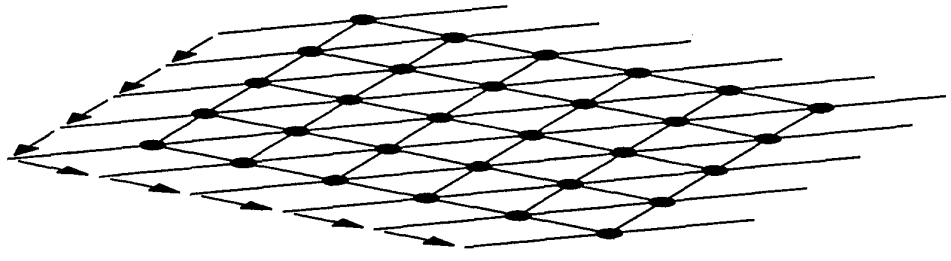


Figure 4: Update ordering according to the hyperplane prescription. In the case of a two-dimensional square lattice the hyperplanes are diagonals.

This transformation may be managed by a restructuring compiler using 'loop skewing' and 'loop interchanging' [11]. However, the resulting code exploits only part of the potential parallelism. This is due to the short inner loop at the beginning and end of the j-loop. Furthermore, a generalization to irregular grids seems to be rather unwieldly.

Restructuring compilers have to respect the data dependences that arise from the particular order of execution defined by the source program. Hence they can examine only a subset of the set of all possible orders. For instance, the hyperplane transformation respects the data dependences displayed in Fig. 3 of the update order shown in Fig. 2. To leave the class of execution orders that can be examined by such a compiler, one has to choose a different update order already on the algorithm level.

In our example a well-known update order is the so-called red-black or checkerboard scheme [22]. Looking at the fine-grained solution (Fig. 5), suitable especially for vector processing, the utilizable parallelism is obvious: half of the points can be updated simultaneously or computed in a vectorized loop. The scheme can also be employed to coarse-grain the underlying data domain (see Fig. 6), where all grains of the same color can be processed concurrently. Each grain is processed sequentially. If the processing elements are vector processors, the red-black decomposition can be applied recursively in each grain.

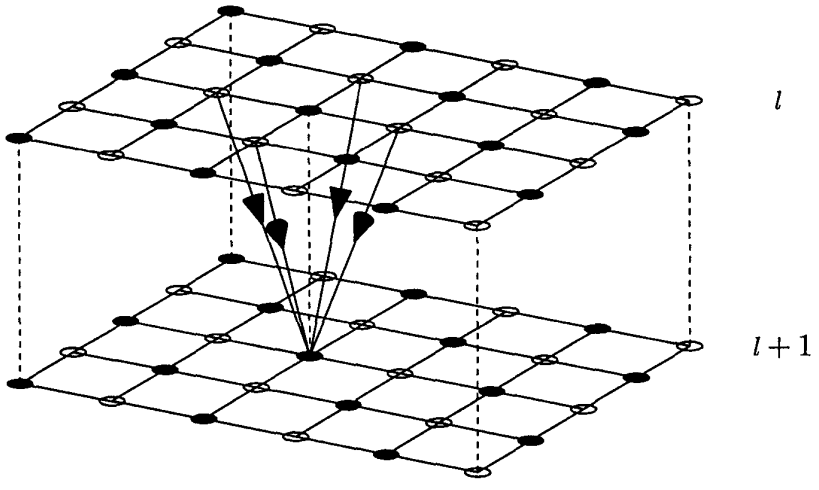


Figure 5: Using the fine grained red-black scheme, data flow-dependences occur between consecutive sweeps only.

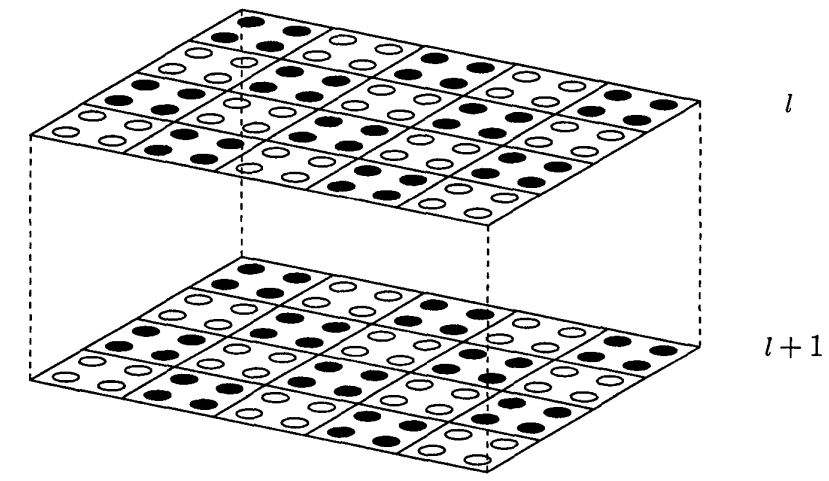


Figure 6: Decomposition by a coarse grained red-black scheme, that moves the data dependences on the boundaries of the colored squares to the outer l-loop. Coloring recursively renders execution with multiple vector processors possible.

5 Gauß-Seidel Iterations for Irregular Problems

5.1 General Procedure

To generalize the red-black scheme to arbitrary graphs we look for a partition of the set I into disjoint subsets $I_1 \cup I_2 \cup \ldots \cup I_c = I$. Due to the Gauß-Seidel condition these sets must be independent sets. Therefore a partition $(I_1, I_2, \ldots, I_c)$ must be a c-coloring of a graph $G(V, E)$. This graph $G(V, E)$ is defined by $V = I$ and $E = \{\{i, j\} | i \in K_j \text{ or } j \in K_i\}$. Since $G(V, E)$ represents the functional dependences[3] we call it *functional graph*. It also displays the connectivity of the underlying 'grid', 'mesh', or 'lattice', if such a geometrical object is used in the application. Otherwise it is to be conceived as an abstract object associated with the iteration problem. Notice that $G(V, E)$ is not a program data dependence graph. This is not defined until a sequence of operations is fixed.

The conditions for a Gauß-Seidel iteration scheme, given in Sect. 3, allow arbitrary update orders, up to the condition that two variables x_i and x_j

[3]If there are pairs (i, j) with $i \in K_j$ but $j \notin K_i$, a 'sharp' *directed* graph $g(V, E)$ with $E = \{(i, j) | i \in K_j\}$ represents the functional dependences more precisely. For our purposes it is sufficient to consider $G(V, E)$.

must not be updated concurrently if i and j are adjacent. We do not want to exhaust the whole set of possible update orders and specialize to synchronized sweeps. A *sweep* is an update of each variable once. Sweeps are *synchronized* if before starting computation of any update value x_j^{l+1}, all values $(x_i^l)_{i \in I}$ must have been computed.

Suppose a partition $I_1 \cup I_2 \cup \ldots \cup I_c = I$ has been found, then a sweep can be formulated as two nested loops:

for $m = 1$ **to** c
  forall $j \in I_m$
    $x_j \leftarrow F_j((x_k)_{k \in K_j})$

The inner loop is free of data dependences, since the iteration space dependence graph for fixed m contains no arcs between instances of the assignment $x_j \leftarrow F_j((x_k)_{k \in K_j})$. Therefore this loop can be executed in arbitrary order. This is emphasized by the keyword **forall**. All data flow-dependences now appear between different iterations of the outer loop.

5.2 Implementation on Vector and Concurrent Architectures

On vector and SIMD architectures efficient implementation of the inner loop

forall $j \in I_m$
  $x_j \leftarrow F_j((x_k)_{k \in K_j})$

obviously requires that the functional form of F_j is identical[4] for all $j \in I_m$. This is a strong restriction. Of course on vector architectures the functions F_j must be vectorizable. Assuming these conditions are fulfilled, we are left with the problem of varying cardinalities $(|K_j|)_{j \in I_m}$, that require further control structures. For the present assume that the lower and upper bound (δ and Δ) of the cardinalities of the sets K_i are known at compile-time. This allows to write the operators F_j explicitely for each cardinality of K_j without additional control structures. Then we propose two solutions to the problem of varying cardinalities:

1. Introduction of Dummy Variables

   To get an equal number of operands in each task F_j, one may add some dummy elements $x_{n+1}, x_{n+2}, \ldots$ to the set of variables $(x_i)_{i \in I_m}$. The usage of dummy elements must result in neutral operations like addition of zero or multiplication by one. The index sets $(K_i)_{i \in I_m}$ are supplemented by the indices $n+1, n+2, \ldots$, such that $|K_i^{(\text{new})}| =$

[4] If there are not too many different types of tasks, each group of identical tasks can be treated separately with the methods described in this section.

$\max_{j \in I_m} |K_j^{(\text{old})}| := d$, for all $i \in I$. Then F_j has $d \leq \Delta$ operands. The new indices do not enter the graph $G(V, E)$ and therefore do not influence its coloring.

The advantage of this method is, that the number of colors does not increase. Since it may be difficult or impossible to find neutral elements, e. g., if $x_j \leftarrow \sum_{k \in K_j} \exp(x_k)$, this procedure is not generally applicable.

2. Subdivision of I

   Another solution is to sort the tasks according to their number μ of operands before coloring. This means that I is partitioned into $I = I^\delta \cup I^{\delta+1} \cup \ldots \cup I^\Delta$, where $I^\mu := \{i \in I \,|\, |K_i| = \mu\}$ and $\delta \leq \mu \leq \Delta$. Now each subgraph corresponding to I^μ is colored separately. Suppose c^μ colors are needed, implying a second partition $I^\mu = I_1^\mu \cup I_2^\mu \cup \ldots \cup I_{c^\mu}^\mu$. Then nested loops are invoked for each cardinality μ:

   for $\mu = \delta$ **to** Δ
   for $\nu = 1$ **to** c^μ
   forall $j \in I_\nu^\mu$
   $x_j \leftarrow F_j((x_k)_{k \in K_j})$

   Due to the double decomposition of I into $\sum_{\mu=\delta}^{\Delta} c^\mu$ subsets, the inner loop has become shorter. This may be disadvantageous for small $|I|$, for large variations in the cardinalities $|K_j|$, or for large c^μ. But in contrast to the use of dummy variables this method is generally applicable.

According to our assumption that the operators F_j are written out explicitely for each μ, the μ-loop has a symbolic meaning here. There may be another formulation if the statement $x_j \leftarrow F_j((x_k)_{k \in K_j})$ can be written as a loop itself:

for $\kappa = 1$ **to** $|K_j|$
$x_j \leftarrow f(x_{k(j,\kappa)})$

where $\bigcup_{\kappa=1}^{|K_j|} \{k(j, \kappa)\} = K_j$. Provided the κ- and j-loop can be exchanged, the complete loop-nest is:

for $\mu = \delta$ **to** Δ
for $\nu = 1$ **to** c^μ
for $\kappa = 1$ **to** μ
forall $j \in I_\nu^\mu$
$x_j \leftarrow f(x_{k(j,\kappa)})$

We have put $|K_j| = \mu$ according to method 2. This formulation does not require that δ and Δ are known at compile-time.

In all loop nests mentioned in this subsection, the statements of the innermost loop can be executed in any order. Hence it is parallelizable, or vectorizable if the operators are representable by vector functions. F_j may especially be the operator "take the weighted sum over nearest neigbors", what makes the method usable for many problems in scientific computing.

Up to now we have neglected the important aspect of communication delays on concurrent architectures. To reduce data transfers, coarse grain solutions are favorable. Coarse grains are comprised of several primitive tasks F_j. They are represented by induced subgraphs of the functional graph $G(V, E)$. The communication costs are proportional to the number of edges that connect vertices of different subgraphs. A good decomposition strategy has to take into consideration the target architecture and the problem to be solved. The aim generally should be finding subgraphs with few edges to other subgraphs and with computation costs being equally distributed among the associated grains. Each subgraph can be considered as a vertex of a 'coarse' graph. Two vertices of this graph are connected if the corresponding subgraphs are connected. For parallel processing the coarse graph has to be colored. Multi-vector processing is rendered possible by additionally coloring each subgraph.

In contrast to the usual checkerboard decompositions, this general procedure has the advantage of not imposing restrictions on the shape and topology of the underlying mesh and grains—if such a geometrical interpretation is associated with the application at all. Furthermore the variability of individual grain sizes allows flexible load balancing.

Concerning the execution order that is chosen after coloring there remains a freedom, that has not been used up to now. All **for**-loops in Sects. 5.1 and 5.2 must be executed sequentially, but this may happen in arbitrary order. Choosing a specific execution order in the Gauß-Seidel iteration scheme does not affect the correctness of the algorithm, but may influence the convergence rate. The freedom of choosing any permutation of the indices of the **for**-loops therefore may be exploited to improve the convergence of the iterative algorithm.

6 Graph Coloring

We saw that structuring data dependences in Gauß-Seidel iterations is feasible by graph coloring. The choice of the coloring algorithm is determined only by practical considerations. Among these are the *performance* of the algorithm, i. e., the ratio of the number of colors being used and $\chi(G)$, and its *efficiency*, i. e., the time needed to color the graph depending on the complexity of the graph. A $\chi(G)$-coloring might be the optimal solution for the given problem. But the determination of $\chi(G)$ is known to be a NP-

complete problem for general graphs. Practically, one is interested in finding good approximations for $\chi(G)$ in polynomial time or in figuring out special kinds of graphs for which $\chi(G)$ can be found in polynomial time.

At least for perfect graphs, $\chi(G)$ can be computed in polynamial time [24]; but the authors of [24] appraise their algorithm as practially too slow. Another interesting class of graphs are planar graphs, for which $\chi(G)$ is known to be 4. There are linear time 5-coloring algorithms for planar graphs (see references in [25]) and algorithms for 4-coloring perfect planar graphs that take $O(n \log n)$ time [25]. For general graphs an algorithm with good performance was given in [26]. It takes $O(n^3)$ time. The parallelization of coloring itself is also a subject of research [25, 27].

The references just mentioned present optimal solutions to the coloring problem, but mostly deal with special graphs. If one is interested in finding a practical way to overcome the problem of unstructured data dependences, one can try to use a simple coloring procedure, which is known as *sequential coloring* [28] and proceeds as follows:

chose an order $v_1, v_2, \ldots, v_n$ of the vertices of G
for $i = 1$ **to** n
    give v_i the smallest color that has not been used in
    $N(v_i) \cap \{v_1, v_2, \ldots, v_{i-1}\}$

To analyse performance and efficiency of this procedure we remark that it maximally uses $\Delta(G) + 1$ colors. This is true because the worst case for giving a colour occurs when all neighbors of a vertex with maximal degree have already been colored with different colors. Sequential coloring takes $O(n \log n)$ time if one assumes that $\Delta(G) \propto \log n(G)$, what is supposed to be an upper bound for meshs of discretized surfaces or discretized three dimensional objects.

If coloring has the status of a preprocessing step for a long production run, the expense of coloring is neglectible. In general, efficiency of the coloring program has to be judged by the gain achieved by vectorization or parallelization. The demands on the performance of a coloring algorithm among other things depend on the specific details of the target architecture. For instance, on vector computers with vector registers it is favorable to achieve loop lengths being greater than or equal to the vector register length; short loops might be made longer by re-coloring.

7 Example of Application

The authors encountered the problem of vectorizing an irregular problem when investigating a model from quantum field theory by a Monte-Carlo

method[5]. The model was studied on spheres and on tori [29, 30], which have been discretized by Poissonian Delaunay lattices [31], called *random lattices* in physics. The irregularity of the problem is caused by the varying degrees of the vertices.

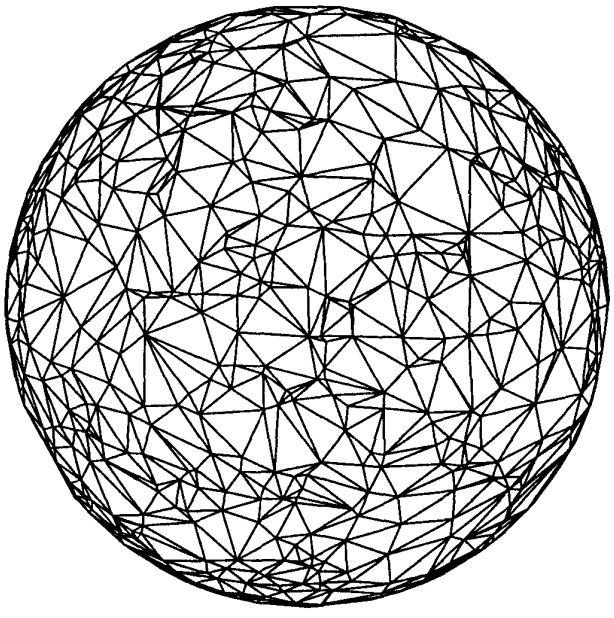


Figure 7: A spherical random lattice (Poissonian Delaunay lattice) consisting of 1000 vertices.

The Monte-Carlo method that has been employed generates a Markov chain. To satisfy the Markov condition the Gauß-Seidel iteration scheme has to be used. In the simulation program the operator F_i is a function f of the sum of the nearest neighbor variables of x_i. According to our notation, the indices of the nearest neighbors are the elements of the sets K_i. The function f involves random numbers, trigonometric, logarithmic and exponential functions. Since the neighbors only enter the sum, and the sum is only a small part of the update prescription, dummy variables have been introduced—more than one, to avoid bank conflicts. The resulting dummy operations cause only little overhead. The loop for calculating the sum of Δ neighbor terms was automatically generated by a preprocessor.

A coloring algorithm specific to random lattices was devised and implemented. The algorithm is based on some heuristic assumptions taking into

[5]Here, a random variable is given by a lattice configuration. If the memory is large enough to store many configurations, these could be sampled simultaneously. Hence, in the special case of such Monte-Carlo methods, vectorization might also be possible in the configuration index.

account that the random lattices describe surfaces. It also tries to use colors equally often. After the coloring process the program re-colors, such that all colors are approximately used the same number of times.

The coloring program usually needs $\Delta/2$ colors. Typical values for a lattice with about 10000 sites are $\delta \sim 3$ and $\Delta \sim 15$.

Without using coloring, a correct implementation of the Monte-Carlo algorithm would have led to purely scalar execution. Utilizing coloring, the simulation program has been fully vectorized. There has been one gather operation per iteration. The corresponding scatter operation has been eliminated by relabeling the variables.

The program has been implemented on a CRAY X-MP/24. The whole preprocessing including coloring took O(seconds) time, while the production runs took O(10 hours). The time needed for coloring could be neglected and the gain was the speed-up achieved by vectorization.

8 Concluding Remarks

The data dependences in irregular problems hinder parallel or vector execution. Irregular structures often appear in modern adaptive algorithms, e. g., by spatial adaption in a PDE solver that refines the grid near singularities. It has been argued that efficient processing of such problems can be achieved by finding execution orders, that imply more regular dependence structures. This idea has been elaborated for Gauß-Seidel iteration schemes. The general problem consists of two parts:

- The class of execution orders accessible to restructuring compilers usually does not contain an order that fully exploits the parallelism.
- Different types of tasks cause irregularities: the types of operations and/or numbers of operands vary among the tasks.

Concerning the execution order it has been shown that for Gauß-Seidel iteration schemes coloring the functional graph provides a decomposition into sets of tasks that can be executed in parallel. Two methods have been proposed to master on SIMD and vector machines different types of tasks. Coarse grained decomposition for distributed memory machines has been shortly discussed. A short survey of graph coloring algorithms has been given. The practical applicability has been illustrated by an example in which Gauß-Seidel update is imperative.

The problem has been presented from the perspective of data dependences and execution orders. This unveils the source of difficulties when vectorizing and parallelizing irregular problems: *structureless data dependences by unfavorable execution orders*. Efficient concurrent or vector execution of irregular

problems can be achieved by finding instruction sequences that give rise to more regular dependences.

Any level above the instruction issue level potentially can contribute to a solution. One could imagine specialized hardware units that monitor instruction and data streams to support run-time dependence checking and operation re-ordering—without completely complying with the data flow concept. Since on the hardware level the possibilities of intelligent operation reordering probably are rather restricted, accompanying activities on the software level are necessary. These could include generation of additional code for steering run-time dependence checking and instruction reordering. Independently, low level compiler optimizations and restructuring program transformations are applicable to unstructured problems (for examples see [9]). Another possibility is to let compilers ignore constraints arising from data dependences, e. g., to oblige compilers by user-specified directives to ignore data anti-dependences in specific loops. On the algorithm level one should devise algorithms that do have less, or more structured dependences, as well as methods that find suitable operation sequences from given functional dependences and constraints on the execution order.

Some of the problems we have encountered arise from the use of programming languages that are unsuitable for concurrent programming: they do not allow to specify problems without implying an execution order—at least partially. This unnecessarily restricts the space of search of possible execution sequences being accessible to compilers. It is desirable to overcome this language barrier by using languages, that allow to formulate the *functional dependences*—without implying any execution order—and to separately specify *constraints on the execution order*. Additionally, powerful methods on the algorithm, software and machine level are required, that transform these representations into efficient execution sequences.

References

[1] K.-Y. Wang, D. Gannon, "Applying AI Techniques to Program Optimization for Parallel Computers". In K. Hwang, D. DeGroot (eds.). Parallel Processing for Supercomputers and Artificial Intelligence, pp. 441–485, McGraw-Hill, New York, 1989.

[2] D. J. Kuck, R. H. Kuhn, B. Leasure and M. Wolfe, "Dependence Graphs and Compiler Optimizations", Proc. of the 8th ACM Symp. of Programming Languages, Williamsburg, VA, pp. 207–218, Jan. 1981.

[3] M. J. Wolfe, Optimizing Supercompilers for Supercomputers, Ph. D. Thesis, Univ. of Illinois, Urbana, IL, Dept. of Comp. Sci. Rpt. No. 82–1105, Oct. 1982.

[4] J. R. Allen and K. Kennedy, "Automatic translation of Fortran Programs to Vector Form", ACM Transactions on Programming Languages and Systems, Vol. 9, No. 4, pp. 491–542, Oct. 1987.

[5] M. Burke and R. Cytron, "Interprocedural Dependence Analysis and Parallelization", Proc. of the SIGPLAN '86 Symposium on Compiler Construction, Palo Alto, CA, pp. 162–175, June 1986.

[6] C. D. Polychronopoulos, "Loop Coalescing: A Compiler Transformation for Parallel Machines", Proc. of the 1987 Int. Conf. on Parallel Processing, St. Charles, IL, Sahni (ed.), Penn State Univ. Press, University Park, PA, pp. 235–242, Aug. 1987.

[7] C. D. Polychronopoulos, "Compiler Optimization for Enhancing Parallelism and their impact on Architecture Design", IEEE Trans. Comput., Vol. 37, No. 8, pp. 991–1004, Aug. 1988.

[8] J.-K. Peir, R. Cytron, "Minimum Distance for Partitioning Recurrence for Multiprocessors", IEEE Trans. Comput., Vol. 38, No. 8, pp. 1203–1211, Aug. 1989.

[9] C. D. Polychronopoulos, Parallel Programming and Compilers, Kluwer Academic Publishers, Boston, 1988.

[10] U. Banerjee, Dependence Analysis for Supercomputing, Kluwer Academic Publishers, Boston, 1988.

[11] M. Wolfe, Optimizing Supercompilers for Supercomputers, Pitman, London, 1989.

[12] M. Burke, "An Interval-Based Approach to Exhaustive and Incremental Interprocedural Data-Flow Analysis", ACM Trans. Program. Lang. Syst. 12, 3 (July 1990), pp. 341–395.

[13] C. D. Polychronopoulos, M. Girkar, M. R. Haghigat, C. L. Lee, B. Leung and D. Schouten, "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing and Scheduling Programs on Multiprocessors", Int. J. High Speed Comput. 1, 1 (1989), pp. 45–72.

[14] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, D. Walker, Solving Problems on Concurrent Processors, Vol. 1 (General Techniques and Regular Problems), Prentice Hall International, London, 1989.

[15] I. G. Angus, G. C. Fox, J. S. Kim, D. W. Walker, Solving Problems on Concurrent Processors, Vol. 2 (Software for Concurrent Processors), Prentice Hall International, London, 1989.

[16] Arvind and R. S. Nikhil, "Executing a program on the MIT tagged-token data flow architecture", in Proc. PARLE Conf. Eindhoven, The Netherlands, Springer-Verlag, Lecture Notes in Computer Science 259, June 1987.

[17] G. Rote, "Path Problems in Graphs", in G. Tinhofer, E. Noltemeier, M. Syslo (eds.), Computational Graph Theory, Springer-Verlag, Computing Suppl. 7, Wien, 1990, pp. 155–198.

[18] D. Chazan, W. Miranker, "Chaotic Relaxation", Linear Algebra Appl. 2 (1969), pp. 199–222.

[19] D. Mitra, "Asynchronous Relaxations for the Numerical Solution of Differential Equations by Parallel Processors", SIAM J. Sci. Stat. Comput. 8 (Jan. 1987), pp. 43–58.

[20] A. Üresin, M. Dubois, "Parallel Asynchronous Algorithms for Discrete Data", J. ACM 37, 3 (July 1990), pp. 588–606.

[21] L. Lamport, "The Parallel Execution of DO Loops", Comm. of the ACM, (Febr. 1974), pp. 83–93.

[22] J. M. Ortega, R. V. Vogt, "Solution of Partial Differential Equations on Vector and Parallel Computers", SIAM Review 27, 2 (June 1985), pp. 149–240.

[23] G. M. Baudet, "Asynchronous Iterative Methods for Multiprocessors", J. ACM 25, 2 (Apr. 1978), pp. 226–244.

[24] M. Grötschel, L. Lovász and A. Schrijver, "Polynomial Algorithms for Perfect Graphs", Annals of Discrete Mathematics 21, (1984), pp. 325–356.

[25] Xin He, "Efficient Parallel and Sequential Algorithms for 4-coloring Perfect Planar Graphs", Algorihmica 5, (1990), pp. 545–559.

[26] B. Berger and J. Rompel, "A Better Performance Guarantee for Approximate Graph Coloring", Algorithmica 5, (1990), pp. 459–466.

[27] J. Žerovnik, "A Parallel Variant of a Heuristical Algorithm for Graph Coloring", Parallel Computing 13, (1990), pp. 95–100.

[28] D. de Werra, "Heuristics for Graph Coloring", Computing Suppl. 7, (1990), pp. 191–208.

[29] H. Stüben, "Monte-Carlo Untersuchungen der topologischen Suszeptibilität im zweidimensionalen nichtlinearen O(3)-σ-Modell", Thesis, Freie Universität Berlin, 1989.

[30] H. Stüben, H.-C. Hege, A. Nakamura, "The Nonlinear O(3)-σ-Model on Random Lattices with Different Topology", Phys. Lett. 244B, (1990), pp. 473–478.

[31] H.-C. Hege, "Geometrie der Zufallsgitter, ihre Realisierung und ihr Verhalten als physikalische Systeme am Beispiel des Z(2)-Spin-Modells", Thesis, Freie Universität Berlin, 1984.