Finite State Machine Based Optimization of Data Parallel Regular Domain Problems Applied in Low Level Image Processing

F.J. Seinstra*, D. Koelma, and A.D. Bagdanov
Intelligent Sensory Information Systems,
Faculty of Science, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
{fjseins, koelma, andrew}@science.uva.nl

*Corresponding author.

Abstract

A popular approach to providing non-experts in parallel computing with an easy-to-use programming model, is to design a software library consisting of a set of pre-parallelized routines, and hide the intricacies of parallelization behind the library's API. However, for regular domain problems (such as simple matrix manipulations or low level image processing applications — in which all elements in a regular subset of a dense data field are accessed in turn) speedup obtained with many such library-based parallelization tools is often sub-optimal. This is because inter-operation optimization (or: time-optimization of communication steps across library calls) is generally not incorporated in the library implementations.

This paper presents a simple, efficient, finite state machine-based approach for communication minimization of library-based data parallel regular domain problems. In the approach, referred to as *lazy parallelization*, a sequential program is parallelized automatically at run time by inserting communication primitives and memory management operations whenever necessary. Apart from being simple and cheap, lazy parallelization guarantees to generate legal, correct, and efficient parallel programs at all times.

The effectiveness of the approach is demonstrated by analyzing the performance characteristics of two typical regular domain problems obtained from the field of low level image processing. Experimental results show significant performance improvements over non-optimized parallel applications. Moreover, obtained communication behavior is found to be optimal with respect to the abstraction level of message passing programs.

Keywords

Parallel processing, data communications aspects, optimization, image processing software.

I. Introduction

A parallelization tool based on a software library of pre-parallelized routines can serve as a powerful programming aid to obtain high performance with relative ease. In the field of low (pixel) level image processing, for example, many such parallelization tools exist [10], [11], [13], [14], [21], [32], [33]. Such tools, however, generally restrict performance optimization to each library operation *in isolation*, and ignore communication minimization for full applications. For library implementations based on message passing primitives significant performance gains can be obtained, as it is often possible to remove many redundant communication steps, and to combine multiple messages in a single transfer.

Automatic optimization of communication overhead is not easy. First, this is because the optimization strategy must be able to determine which communication steps are essential, and which can be safely combined or removed. Also, it must guarantee that the

resulting parallel code is (1) *efficient*, preferably comparable to an optimal hand-coded implementation, (2) *legal*, such that the program is deterministic and can never end in deadlock, and (3) *correct*, such that it produces output identical to the original program.

This paper presents a new and surprisingly simple strategy for communication minimization in library-based data parallel regular domain problems [22], which adheres to all these requirements. In the approach, a *fully sequential* program is parallelized automatically *at run time* by inserting communication primitives and additional memory management operations whenever necessary. The approach, referred to as *lazy parallelization*, is based on a simple *finite state machine (fsm)* specification. One of two essential fsm ingredients is a set of states, each corresponding to a valid internal representation of a distributed data structure at run time. The other is a set of state transition functions, each of which defines how a valid data structure representation is transformed into another valid representation. This paper indicates how the fsm specification is applied in the process of obtaining legal, correct, and indeed efficient parallel code. Also, a compile-time extension is discussed, which is capable of producing the theoretically fastest parallel version of a program.

This paper is organized as follows. Section II describes the optimization problem. In Section III the finite state machine specification is presented. Section IV describes the fsm-based approach of lazy parallelization, and briefly presents a compile-time extension for additional optimization. An evaluation of measurements obtained for two example regular domain problems obtained from the field of low level image processing is presented in Section V. Section VI discusses related work. Concluding remarks are given in Section VII.

II. The Optimization Problem

The main objective in our research is to build a library-based software architecture that allows for *fully sequential* implementation of low level image processing applications executing in data parallel fashion [25], [26], [27], [29]. All parallelization and optimization issues are to be taken care of by the architecture itself, hidden from the user.

A. Parallelizable Patterns in Regular Domain Problems

For reasons of software maintainability and reuse, all library operations are implemented on the basis of a definition of so-called *parallelizable patterns* found in typical regular do-

main problems [29]. Each such pattern represents a generic description of a class of *sequential* algorithms with similar behavior in terms of data accesses to array-like structures. More specifically: a parallelizable pattern represents a generic operation that takes zero or more source structures as input and produces exactly one destination structure as output. It consists of n independent tasks, where a task specifies what data in any of the structures must be acquired in order to update the value of a single data point in the destination structure. As such, prior to *parallel* execution of a pattern, for all data structures on all processing units all data accesses are known. As all accesses are defined to be *local* to the processing unit executing the algorithm, all *non-local* data to be accessed must be communicated prior to execution. Given the precise definition of these *data access pattern types*, a default parallelization strategy with minimal communication overhead directly follows for *any* operation that maps onto one of the predefined parallelizable patterns [29]. Irrespective of the focus on low level image processing, due to the generic nature of parallelizable patterns this result naturally extends to other regular domain problems as well.

B. Abstract Function Specifications

As stated, in our software architecture all sequential image processing functionality is implemented on the basis of parallelizable patterns. For these operations we introduce a shorthand notation, presented in Table I. It includes (a.o.) unary and binary pixel

Create	(	OUT	dst	);					// create global structure
Delete	(	OUT	dst	);					// delete global structure
Import	(	OUT	dst	);					// import global structure from ext. device
Export	(	IN	src	);					// export global structure to ext. device
MemCopy	(	IN	src,	OUT	dst	);			// copy global structure
UnPixOp	(	IN	src,	OUT	dst	);			// unary pixel operation
BinPixOpV	(	IN	src,	OUT	dst,	IN	arg	);	// binary pixel operation (vector argument)
BinPixOpI	(	IN	src,	OUT	dst,	IN	arg	);	// binary pixel operation (image argument)
ReduceOp	(	IN	src,	OUT	dst	);			// global reduce operation
NeighOp	(	IN	src,	OUT	dst,	IN	ker	);	// generalized neighborhood operation
GenConvOp	(	IN	src,	OUT	dst,	IN	ker	);	// generalized convolution
RecGConvOp	(	IN	src,	OUT	dst,	IN	ker	);	// recursive generalized convolution
GeoMat	(	IN	src,	OUT	dst	);			// geometric transform. (matrix-based)
GeoRoi	(	IN	src,	OUT	dst	);			// geometric transform. (region of interest)

TABLE I
Abstract functions: sequential operation.

CreatLclPart	(	OUT	ldst	);			// create non-overlapping structure at all nodes
CreatLclFull	(	OUT	ldst	);			// create fully overlapping structure at all nodes
DelLcl	(	OUT	ldst	);			// delete local structure at all nodes
Broadcast	(	IN	gsrc,	OUT	ldst	);	// send global structure to all nodes
Scatter	(	IN	gsrc,	OUT	ldst	);	// divide global structure among all nodes
Gather	(	IN	lsrc,	OUT	gdst	);	// send each node's local structure to root
GatherAll	(	INOUT	lsrc,	INOUT	gdst	);	// send each node's local structure to all nodes
ReduceOne	(	INOUT	lsrc,	OUT	gdst	);	// global reduce across all nodes (result at root)
ReduceAll	(	INOUT	lsrc,	INOUT	gdst	);	// global reduce across all nodes (result at all)

TABLE II
Abstract functions: communication.

operations, (recursive) neighborhood operations, and geometric transformations.

Shorthand notation for all required inter-process communication is presented in Table II, and contains the common collective operations in MPI [17]. The additional CreatLclPart/Full and DelLcl functions constitute creators and destructors for *partial* data structures, each residing on a different processor at run time. Partial structures are referred to as *local* in the presented parameter lists (lsrc and ldst). The original structure from which the partial structures are obtained is referred to as *global* (gsrc and gdst). The importance of these abstractions is that for any application implemented using our architecture it is possible to derive an abstract operation stream comprising of functions from Tables I and II alone.

C. Default Algorithm Expansion

Because all functionality is implemented on the basis of parallelizable patterns, conversion of any sequential application into an equivalent parallel program is straightforward. The conversion process, referred to as *default algorithm expansion*, is illustrated in Listing 1. The sequential program, shown on the left, first imports image A, which is used as input to a unary pixel operation. Subsequently, resulting image B is used as input to a binary pixel operation. Finally, resulting image C is exported, and all images are destroyed.

The equivalent parallel program is shown on the right of Listing 1. First, a Scatter operation is inserted before the UnPixOp call. After the operation has finished, the resulting partial outputs are gathered to the single root node and all temporary partial structures are destroyed. Subsequently, the images which are passed as source and argument to the binary pixel operation are scattered throughout the parallel system. The partial outputs

```
Import( A );
UnPixOp( A, B );
BinPixOpI( B, C, A );
Export( C );
Delete( A );
Delete( B );
Delete( C );
```

(a) Sequential.

```
Import( A);
  Scatter( A, locA );
UnPixOp( locA, locB );
  Gather( locB, B );
  DelLcl( locA );
  DelLcl( locB );
  Scatter( A, locA );
  Scatter( B, locB );
BinPixOpI( locB, locC, locA);
  Gather( locC, C );
  DelLcl( locA );
  DelLcl( locB );
  DelLcl( locC );
Export( C );
Delete( A );
Delete( B );
Delete( C );
```

(b) Parallel (default).

Listing 1: *Abstract sequential application (a) and equivalent parallel program after default algorithm expansion (b).*

resulting from BinPixOp are gathered to the root, after which all partial structures are deleted. From this point onward, the program is identical to the original sequential version.

Default algorithm expansion always generates a legal and correct parallel version of any sequential program implemented on the basis of parallelizable patterns. This is because each abstract function call in the sequential code is replaced by an equivalent sequence of one or more (parallel) operations. The parallel code is not guaranteed to be time-optimal, however. Worse even, it can be expected to be *slower* than the original sequential program. Although other tools may have different implementations, all library-based tools suffer from the very same problem — and for improved performance a solution is essential.

D. The Problem: Inefficiencies from Default Algorithm Expansion

When considering the parallel code of Listing 1(b), it is clear that it contains several operations that could be removed without violating the program's correctness or legality. First, image locA, used as source structure for the unary pixel operation, is removed by DelLcl and subsequently recreated in the second occurrence of the Scatter(A, locA) call. For improved performance, both operations simply could be removed. The same holds for the sequence of instructions applied to the locB structure preceding the BinPixOpI call (i.e., Gather followed by DelLcl and Scatter). Listing 2(b) presents the optimized program obtained after removing the redundant communication steps from the parallel code. The remainder of this

```
Import( A );
UnPixOp( A, B );
BinPixOpI( B, C, A );
Export( C );
Delete( A );
Delete( B );
Delete( C );
```

(a) Sequential.

```
Import( A );
  Scatter( A, locA );
UnPixOp( locA, locB );
BinPixOpI( locB, locC, locA);
  Gather( locC, C );
  DelLcl( locA );
  DelLcl( locB );
  DelLcl( locC );
Export( C );
Delete( A );
Delete( B );
Delete( C );
```

(b) Parallel (optimized).

Listing 2: *Abstract sequential application (a) and equivalent parallel program after inter-operation optimization (b).*

paper indicates how execution of such redundant operations can be avoided automatically.

III. Finite State Machine Definition

Our solution to the problem of redundant communication avoidance is based on a *finite state machine (fsm)* specification. More specifically, we restrict ourselves to a *deterministic finite accepter (dfa)* [9], defined by the quintuple $M = (Q,\ \Sigma,\ \delta,\ q_0,\ F)$, where

Q is a finite set of *internal states*,

Σ is a finite set of symbols called the *input alphabet*,

$\delta : Q \times \Sigma \rightarrow Q$ is a *transition function*,

$q_0 \in Q$ is the *initial state*,

$F \subseteq Q$ is a set of *final states*.

A. Data Structure States and Lifespan

As described in [29], for parallel execution two types of data structure representations are used in our software architecture: *global* structures and *local* (or partial) structures. A global structure always resides at a single processing unit (the root), and contains all data for the complete domain of the structure it represents. Local structures, on the other hand, are the result of a scatter or broadcast operation performed on a global structure.

There is a strong relationship between a global structure and the set of derived local structures (or: *distributed data structure*). Clearly, at any time either the global structure itself or its derived distributed structure must contain all valid data. An abstract

representation of this relationship is given by the triple $q = (g,\ d,\ t)$, where

$$g \in G \text{ is the } \textit{state of the global structure},$$
$$d \in D \text{ is the } \textit{state of the derived distributed structure},$$
$$t \in T \text{ is the } \textit{distributed structure's distribution type},$$

and

$$G = \{ \text{ none, created, valid, invalid } \},$$
$$D = \{ \text{ none, valid, invalid } \},$$
$$T = \{ \text{ none, partial, full, not-reduced } \}.$$

In set G, none indicates that no space has been allocated for the global structure in the main memory of the root. Furthermore, created indicates that space for the global structure has been allocated by way of the Create function. In this state, the elements of the global structure do not contain values resulting from any calculation (yet). Finally, valid indicates that the global structure contains up-to-date values for all structure elements, and invalid indicates that at least one of the global structure's elements may contain an incorrect value. For distributed structures, the elements in set D are defined in a similar manner. The value created is not present in set D, however, simply because we do not need it.

In set T, none indicates that no distribution type information is available. In addition, partial indicates that the set of constituent local structures is the result of a Scatter operation, while full indicates that the structures are obtained in a Broadcast operation. Finally, not-reduced indicates that all elements of the constituent local structures yet have to be subjected to an element-wise ReduceOne or ReduceAll operation (see also [29]).

The set $R = G \times D \times T$ contains all possible representations of the relationship between a global structure and its derived distributed structure. However, many of these possible representations can not (or should not) occur. As an example, the representation $q =$ (invalid, invalid, full) should not occur in a program, as neither the global structure nor the distributed structure contains all correct values.

For the fsm, we have specified a restricted set of *valid internal states*, based on the relationship between global and distributed structures. It is defined by

$$Q = \{\ q_0,\ q_1, \cdots,\ q_8\ \} \subset G \times D \times T,$$

with

q_0 = (none, none, none), q_3 = (invalid, none, none), q_6 = (invalid, valid, partial),

q_1 = (created, none, none), q_4 = (valid, valid, partial), q_7 = (invalid, valid, full),

q_2 = (valid, none, none), q_5 = (valid, valid, full), q_8 = (invalid, invalid, not-reduced).

State q_0 is the *empty state*, and represents the state of the global-distributed structure combination before its initial creation and after its final destruction. State q_1 represents the state immediately after creation of the global structure. This is a special case of state q_2, as the global structure also could be designated as valid. State q_1 is still required, however, to avoid communication in case a distributed structure is to be derived from a global structure in this state. State q_2 indicates that a global structure's elements contain all up-to-date values, while a derived distributed structure is nonexistent. At first glance, q_3 seems to be a state that should never appear in a legal parallel program. However, this is the state obtained after performing a DelLcl operation in case the global-distributed structure combination is represented by states q_6, q_7, or q_8. In states q_4, q_5, q_6, and q_7, the distributed structure contains all correct values, while the related global structure is either consistent or inconsistent with these values. Finally, state q_8 occurs in parallel reduction operations. As long as the required reduction has not been performed on the distributed structure, all constituent local structures as well as the related global structure remain invalid.

At run time each global-distributed structure combination starts in the empty state q_0. From this point onward each state can be reached, depending on the operations performed on the structure combination. Also, it is possible for certain states to be reached multiple times. The *lifespan* of a global-distributed structure combination ends in case it returns to the empty state q_0. As such, state q_0 serves as the *initial state* of our finite state machine definition, as well as the single element in the set of *final states*.

B. State Transition Functions

For our purposes, the fsm *input alphabet* is formed by the operations of Tables I and II, with a concrete data structure reference for each formal parameter. Also, as the fsm is used to monitor state changes and lifespan of a *single* data structure only, monitoring the correctness and legality of a complete application involves *multiple* fsm's. This results in a *parallel view* of the states of all data structures in an application: at any moment

$\delta(q_0, (\text{Create}, -)) = q_1, \quad \delta(q_i, (\text{Delete}, -)) = q_0,$ $\delta(q_0, (\text{Import}, -)) = q_2, \quad \delta(q_j, (\text{Export}, -)) = q_j,$ with $i \in \{1,2,3\}, j \in \{1, 2, 4, 5\}$,
$\delta(q_0, (\text{op}, q_2)) = q_2, \quad \delta(q_0, (\text{op}, q_6)) = q_6,$ $\delta(q_0, (\text{op}, q_4)) = q_6, \quad \delta(q_0, (\text{op}, q_7)) = q_7,$ $\delta(q_0, (\text{op}, q_5)) = q_7, \quad \delta(q_i, (\text{op}, q_0)) = q_i,$ with $\text{op} \in \{\text{MemCopy, UnPixOp}\}, i \in \{2, 4, 5, 6, 7\}$,
$\delta(q_0, (\text{op}, q_2, q_2)) = q_2, \quad \delta(q_2, (\text{op}, q_0, q_2)) = q_2,$ $\delta(q_0, (\text{op}, q_4, q_i)) = q_6, \quad \delta(q_4, (\text{op}, q_0, q_i)) = q_4,$ $\delta(q_0, (\text{op}, q_5, q_i)) = q_7, \quad \delta(q_5, (\text{op}, q_0, q_j)) = q_5,$ $\delta(q_0, (\text{op}, q_6, q_i)) = q_6, \quad \delta(q_6, (\text{op}, q_0, q_i)) = q_6,$ $\delta(q_0, (\text{op}, q_7, q_i)) = q_7, \quad \delta(q_7, (\text{op}, q_0, q_j)) = q_7,$ with $\text{op} \in \{\text{BinPixOpV, NeighOp, GenConvOp, RecGConvOp}\}$, $i \in \{5, 7\}, j \in \{4, 5, 6, 7\}$,
$\delta(q_0, (\text{op}, q_2, q_2)) = q_2, \quad \delta(q_2, (\text{op}, q_0, q_2)) = q_2,$ $\delta(q_0, (\text{op}, q_i, q_j)) = q_6, \quad \delta(q_i, (\text{op}, q_0, q_j)) = q_i,$ $\delta(q_0, (\text{op}, q_k, q_l)) = q_7, \quad \delta(q_k, (\text{op}, q_0, q_l)) = q_k,$ with $\text{op} \in \{\text{BinPixOpI}\}, i, j \in \{4, 6\}, k, l \in \{5, 7\}$,
$\delta(q_0, (\text{ReduceOp}, q_2)) = q_2, \quad \delta(q_2, (\text{ReduceOp}, q_0)) = q_2,$ $\delta(q_0, (\text{ReduceOp}, q_i)) = q_8, \quad \delta(q_i, (\text{ReduceOp}, q_0)) = q_i,$ $\delta(q_0, (\text{ReduceOp}, q_j)) = q_7, \quad \delta(q_j, (\text{ReduceOp}, q_0)) = q_j,$ with $i \in \{4, 6\}, j \in \{5, 7\}$,
$\delta(q_0, (\text{op}, q_2)) = q_2, \quad \delta(q_2, (\text{op}, q_0)) = q_2,$ $\delta(q_0, (\text{op}, q_i)) = q_6, \quad \delta(q_i, (\text{op}, q_0)) = q_i,$ with $\text{op} \in \{\text{GeoMat, GeoRoi}\}, i \in \{5, 7\}$.

TABLE III
Transition functions: image operations.

during execution, several structures are 'alive' and their combined state is captured by their respective fsm's. As the states of multiple structures are not always *independent*, we assume that each fsm has a complete and up-to-date view of the states of all data structures in an application. Also, by way of the defined set of *state transition functions*, each fsm incorporates all knowledge regarding data structure *state dependencies*. To this end, the definition of state transition functions as presented before is extended as follows:

$$\delta : Q \times \Sigma_d \rightarrow Q,$$

where Σ_d is the input alphabet in which each function is annotated with a list of permitted state dependencies for all additional structures passed as parameter to that function (i.e., those structures for which the current fsm is not responsible). Here, we represent elements in Σ_d by a pair or triple, in which the first component is the name of the function, and the remainder represents the (possibly empty) list of state dependencies. For example, $\delta(q_0, (\text{BinPixOpV}, q_4, q_5)) = q_6$ represents a state transition function for the output structure produced by the BinPixOpV operation. This transition function changes the state of the output structure from q_0 to q_6, while the source and argument structures are expected to be in states q_4 and q_5 respectively. It should be noted, that the knowledge obtained with this parallel view also can be captured in a single *cross-product machine*, in which each dfa simulates, in parallel, the behavior of each component dfa [16]. For simplicity, however, in the remainder of this paper we keep to the parallel view of simple state machines.

Table III presents the transition functions for the image operations available in our library. In all cases, initial state q_0 refers to the state of the output structure produced by any of the operations. As can be seen, output structures are the only structures that actually move from one state to another. Input structures and argument structures never change state, as these are accessed only, and never updated. All transitions that cause a structure to be moved to state q_1 or q_2 always indicate sequential execution using global structures. All other transitions refer to parallel execution using distributed structures. State transition functions related to the additional communication functionality, and the memory management of local data structures, are presented in Table IV. In all of these the list of state dependencies is empty, as the functions work on a single data structure only.

$\delta(q_1, (\text{CreatLclPart}, -)) = q_4,$	$\delta(q_i, (\text{DelLcl}, -)) = q_2,$
$\delta(q_1, (\text{CreatLclFull}, -)) = q_5,$	$\delta(q_j, (\text{DelLcl}, -)) = q_3,$
with $i \in \{4, 5\}, j \in \{6, 7, 8\},$	
$\delta(q_2, (\text{Broadcast}, -)) = q_5,$	$\delta(q_8, (\text{ReduceOne}, -)) = q_2,$
$\delta(q_2, (\text{Scatter}, -)) = q_4,$	$\delta(q_8, (\text{ReduceAll}, -)) = q_5,$
$\delta(q_6, (\text{Gather}, -)) = q_4,$	$\delta(q_6, (\text{GatherAll}, -)) = q_5,$
$\delta(q_7, (\text{Gather}, -)) = q_5,$	

TABLE IV
Transition functions: communication.

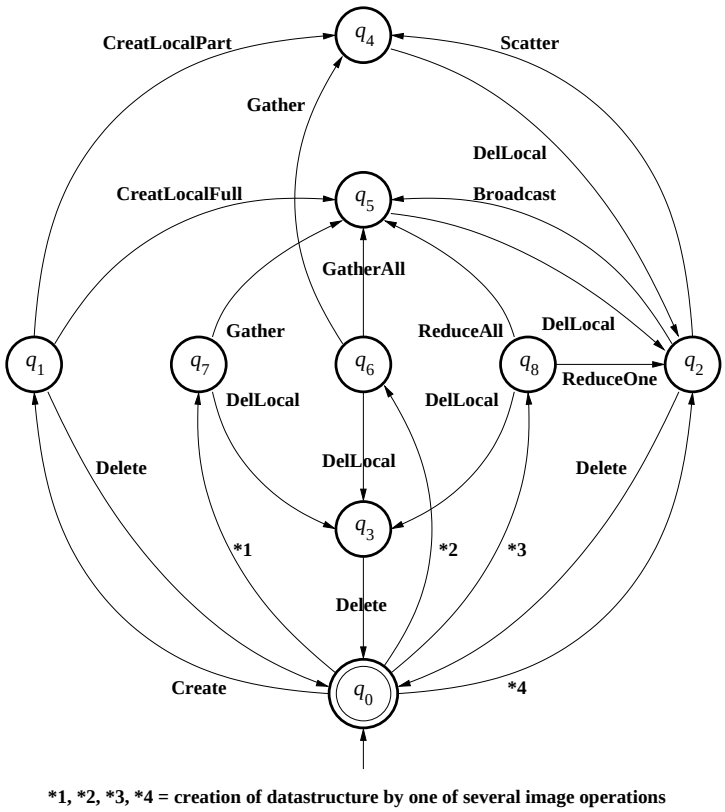


Fig. 1. *Reduced state transition graph.*

Figure 1 presents a reduced state transition graph for the fsm. For better readability, it contains only those operations that cause a structure to move from one state to another. As such, the graph incorporates the complete lifespan of a data structure, and covers any state a structure can reach at run time. Also, it is exactly these operations that are essential in the process of operation redundancy avoidance as presented in Section IV.

A program is *legal*, if it is accepted by *all* fsm's related to that program. In other words, in our architecture a program is legal if (1) it contains function calls from Tables I and II only, (2) it contains no data structure state inconsistencies, and (3) all structures start as well as end in state q_0. In case a user-provided sequential program is legal, default algorithm expansion always generates a legal and correct parallel program. This is because each sequence of (parallel) operations that replaces a sequential call generates exactly the same set of data structure state transitions at all times. The following section shows how the presented fsm is used to obtain legal and correct parallel code, which is optimized in that the execution of any redundant communication operations is avoided.

IV. Lazy Parallelization

In the approach of *lazy parallelization* it is assumed that *each* communication or memory management operation inserted by default algorithm expansion is redundant, unless proven otherwise. Stated differently, lazy parallelization causes an inserted operation to be executed only if its removal would introduce a data structure state inconsistency. Although the method can be applied on the fly at run time, for the moment we will present it as a compile time method. Conceptually, lazy parallelization consists of the following steps:

1. Apply the process of default algorithm expansion to the original sequential code.

2. Remove *all* communication operations, as well as *all* operations for the creation and destruction of partial data structures.

3. Apply *partial loop unrolling* by extracting the code for the first iteration of each loop, and placing it in front of the code for the remaining loop iterations.

4. Resolve data structure state inconsistencies by re-inserting operations removed in step 2.

5. Undo the loop unrolling by collapsing each separated loop into a single code block.

As stated, the parallel code obtained after the first step is legal, but non-optimal. The operation removal in the second step, however, introduces many state inconsistencies. As described below, these inconsistencies are resolved in step four. Steps 3 and 5 are present only to expose all data structure state inconsistencies that can possibly occur in a program.

Listing 3 gives a conceptual example of lazy parallelization. The programs obtained in the first three steps of the optimization process are straightforward, and will not be discussed. The re-insertion of code as applied in step 4 (see Listing 3(e)) is performed using the state transition functions of Section III-B (i.e., only those in the reduced state transition graph of Figure 1). The Broadcast(A, locA) operation in the first loop iteration is inserted because the Import operation causes its output structure to be moved to state q_2, while for parallel execution the subsequent GeoMat operation requires its input structure to be in state q_5 or q_7 (see Table III). The only operation that provides a resolution to this state inconsistency is Broadcast, as it moves a data structure from state q_2 to q_5. Similarly, Gather(locC, C) is inserted in the first loop iteration, as it moves C from q_6 to q_4, which is one of the allowed input states for the subsequent Export operation. The additional re-insertions work in a similar manner, and all further interpretation is left to the reader.

```
Import( A );
LOOP [1:N]
   GeoMat( A, B );
   GenConvOp( B, C, k );
   Export( C );
   Delete( C );
   Delete( B );
ENDLOOP
Delete( A );
```

(a) sequential code

```
Import( A );
LOOP [1:N]
   Broadcast( A, locA );
   GeoMat( locA, locB );
   Gather( locB, B );
   DelLcl( locB );
   DelLcl( locA );
   Scatter( B, locB );
   GenConvOp( locB, locC, k );
   Gather( locC, C );
   DelLcl( locC );
   DelLcl( locB );
   Export( C );
   Delete( C );
   Delete( B );
ENDLOOP
Delete( A );
```

(b) after step 1

```
Import( A );
LOOP [1:N]
   GeoMat( locA, locB );
   GenConvOp( locB, locC, k );
   Export( C );
   Delete( C );
   Delete( B );
ENDLOOP
Delete( A );
```

(c) after step 2

```
Import( A );
LOOP [1]
   GeoMat( locA, locB );
   GenConvOp( locB, locC, k );
   Export( C );
   Delete( C );
   Delete( B );
ENDLOOP
LOOP [2:N]
   GeoMat( locA, locB );
   GenConvOp( locB, locC, k );
   Export( C );
   Delete( C );
   Delete( B );
ENDLOOP
Delete( A );
```

(d) after step 3

```
Import( A );
LOOP [1]
   Broadcast( A, locA );
   GeoMat( locA, locB );
   GenConvOp( locB, locC, k );
   Gather( locC, C );
   Export( C );
   DelLcl( locC );
   Delete( C );
   DelLcl( locB );
   Delete( B );
ENDLOOP
LOOP [2:N]
   GeoMat( locA, locB );
   GenConvOp( locB, locC, k );
   Gather( locC, C );
   Export( C );
   DelLcl( locC );
   Delete( C );
   DelLcl( locB );
   Delete( B );
ENDLOOP
DelLcl( locA );
Delete( A );
```

(e) after step 4

```
Import( A );
LOOP [1:N]
   IF [1] Broadcast( A, locA);
   GeoMat( locA, locB );
   GenConvOp( locB, locC, k );
   Gather( locC, C );
   Export( C );
   DelLcl( locC );
   Delete( C );
   DelLcl( locB );
   Delete( B );
ENDLOOP
DelLcl( locA );
Delete( A );
```

(f) after step 5

Listing 3: *Example of optimization by lazy parallelization: (a) original code, (b) after default algorithm expansion, (c) after removal of 'redundant' operations, (d) after partial loop unrolling, (e) after default operation re-insertion, (f) optimized parallel code after loop recombination.*

A. Discussion

Lazy parallelization produces legal and correct parallel code at all times. This can be seen by considering the allowed states for all structures passed as parameters to the operations in Table I, and the resulting states for the produced output structures. As such, each operation has a set of *allowed input states* for each parameter, one of which is moved to a new *output state*. By exhaustion, it is easily shown that for each possible output state, a finite sequence of zero or more state transitions exists that moves a structure from that output state to one state in each set of allowed input states (see also [28]).

An important property of lazy parallelization is that it can be applied *on the fly at run time* (hence its name). As all data structure states are known for each operation, decisions regarding the execution of each communication step are deferred to as late as the actual moment of execution. Essentially, this means that all five steps as described above are reduced to a single step. This makes lazy parallelization *very easy to implement, and highly efficient* (i.e., without measurable run time overhead). An additional advantage is that *no prior knowledge regarding the behavior of loops and branches is required*. Finally, *run-time adaptation to data structure sizes is easily integrated*, by allowing flexibility in the applied number of processing units (or even by temporarily residing to sequential execution) [25].

Although lazy parallelization produces very efficient parallel code, it is still non-optimal. First, this is because it always applies the fastest communication step whenever message transfer is mandatory. This is a form of local performance optimization, however, as it may be better to insert a combined message transfer to avoid further communication steps at a later stage. Secondly, no knowledge is incorporated regarding the performance characteristics of the parallel machine at hand [26], [29]. To overcome these problems, we have also implemented an extension to the presented approach, which is capable of producing the (expected) fastest parallel version of a sequential program at compile time. The extended approach relies on the creation of an *application state transition graph (ASTG)*, incorporating all relevant performance optimization decisions that can be made at run time. Each decision is annotated with a cost estimation, such that the fastest implementation is represented by the cheapest branch in the graph. Drawback, however, is that it is often costly to actually obtain the cheapest branch. See [25] for more information.

B. Applicability

Although lazy parallelization was designed for data parallel imaging applications, it has a broader applicability. As stated in Section II, the approach will work (and generally be effective) for all regular domain problems in which the essential operations can be expressed in terms of parallelizable patterns. One obvious example is the domain of linear algebra applications. Clearly, for the approach to work in other application areas all references to image operations in the fsm specification should be altered, but this adaptation is only marginal. Also, the fact that operations in other areas may incorporate different data access pattern types does not challenge the validity of the proposed method in any way.

Essentially, lazy parallelization is applicable to irregular (even data driven) problems as well. For the approach to work, however, it is essential to have knowledge regarding the data access pattern types of operations to obtain the required communication sets on the fly at run time. For irregular applications this may not always be effective, especially in cases where nothing is known other than that n accesses are to be performed within a set of m elements, with $m \gg n$. When most elements in the set of size m are non-local, the communication set for each processor will be large. In such cases the performance obtained by lazy parallelization largely depends on the amount of overlap in the communication sets for sequences of operations. The more overlap, the more communication can be avoided.

In the problem of avoiding redundant communication steps the reader may see a relation to similar problems in other research areas. As a first example, there is an analogy to the generation of redundant instructions in the process of compilation. Here, a well-known problem is the avoidance of superfluous transfer of values between registers and (main) memory. As another example, there are similarities to cache coherency problems in the avoidance of unnecessary updates of stale data. Solutions to problems of this kind (e.g., peephole strategies for compilers, I/O address checking for cache accesses, etcetera) all require (often costly) look-ahead strategies to obtain knowledge regarding data accesses. Our solution to redundant communication avoidance is different in that it does not require any form of look-ahead at all. This property directly follows from the knowledge regarding data accesses contained in the definition of parallelizable patterns. As such, our solution to the redundancy problem does not easily transfer to the aforementioned

problems in other research areas. This is because it is often unfeasible or even impossible to incorporate a priori knowledge regarding data accesses in the general case. However, for certain domain-specific problems our approach is still applicable. It is possible, for example, to use compiler annotations in parallel languages such as HPF to obtain particularly efficient parallel code for certain regular domain problems. Specifying code segments as being implemented according to particular parallelizable patterns relieves the compiler of extensive dependency analysis, and allows for lazy parallelization to be incorporated. Currently, this approach is being considered for the SPAR parallel language [24], [31].

V. Measurements and Validation

To evaluate the approach of lazy parallelization, this section describes the implementation and parallel execution of two example image processing applications: (1) line detection, and (2) extraction of rectangular size distributions from document images. The actual code is available at `http://www.science.uva.nl/~fjseins/ParHorusCode/`.

The two applications have been tested on the 72-node Distributed ASCI Supercomputer 2 (DAS-2) located at the Vrije Universiteit in Amsterdam [2]. All nodes consist of two 1-Ghz Pentium-III CPUs, with 2 GByte of RAM, and are connected by a Myrinet-2000 network. At the time of measurement, the nodes ran the RedHat Linux 7.2 operating system. Our software architecture was compiled using `gcc` 2.96 (at highest level of optimization) and linked with MPICH-GM, which uses Myricom's GM as its message passing layer on Myrinet. As the DAS-2 system is heavily used for other research projects as well, measurement results are presented here for a system of up to 64 dual-CPU nodes only.

A. Curvilinear Structure Detection

As discussed in [8], the important problem of detecting lines and linear structures in images is solved by considering the second order directional derivative in the gradient direction, for each possible line direction. This is achieved by applying anisotropic Gaussian filters, parameterized by orientation θ, smoothing scale σ_u in the line direction, and differentiation scale σ_v perpendicular to the line, given by

$$r''(x, y, \sigma_u, \sigma_v, \theta) = \sigma_u \sigma_v \left| f_{vv}^{\sigma_u, \sigma_v, \theta} \right| \frac{1}{b^{\sigma_u, \sigma_v, \theta}},$$

with b the line brightness. When the filter is correctly aligned with a line in the image, and σ_u, σ_v are optimally tuned to capture the line, filter response is maximal. Hence, the per pixel maximum line contrast over the filter parameters yields line detection:

$$R(x,y) = \arg \max_{\sigma_u,\sigma_v,\theta} r''(x,y,\sigma_u,\sigma_v,\theta).$$

A.1 Sequential Implementations

The anisotropic Gaussian filtering problem can be implemented sequentially in many different ways. First, for each orientation θ it is possible to create a new filter based on σ_u and σ_v. Hence, a sequential implementation based on this approach (which we refer to as *Conv2D*) implies full 2-dimensional convolution for each filter.

The second approach (referred to as *ConvUV*) is to decompose the anisotropic Gaussian filter along the perpendicular axes u, v, and use bilinear interpolation to approximate the image intensity at the filter coordinates. Although comparable to the *Conv2D* approach, *ConvUV* is expected to be faster due to a reduced number of accesses to the image pixels.

Pseudo code for the *Conv2D* and *ConvUV* algorithms is presented in Listing 4. Filtering is performed in the inner loop by either a full two-dimensional convolution (*Conv2D*) or by a separable filter in the principle axes directions (*ConvUV*). On a state-of-the-art sequential machine either program may take from a few minutes up to several hours to complete, depending on the size of the input image and the extent of the chosen parameter subspace. Consequently, for the directional filtering problem parallel execution is highly desired.

```
FOR all orientations θ DO
  FOR all smoothing scales σ_u DO
    FOR all differentiation scales σ_v DO
      FiltIm1 = GenConvOp(OriginalIm, "func", σ_u, σ_v , 2, 0);
      FiltIm2 = GenConvOp(OriginalIm, "func", σ_u, σ_v , 0, 0);
      ContrastIm = BinPixOp(FiltIm1, "absdiv", FiltIm2);
      ContrastIm = BinPixOp(ContrastIm, "mul", σ_u × σ_v);
      ResultIm = BinPixOp(ResultIm, "max", ContrastIm);
    OD
  OD
OD
```

Listing 4: *Pseudo code for the Conv2D and ConvUV algorithms, with* `"func"` *either* `"gauss2D"` *or* `"gaussUV"`.

A.2 Parallel Execution

Execution of the parallel versions of the algorithms obtained by default algorithm expansion results in a huge amount of redundant communication overhead. This is because each image operation in the inner loop of the program now executes one or more `Scatter-Gather`-pairs similar to those presented in the example code of Listing 1(b).

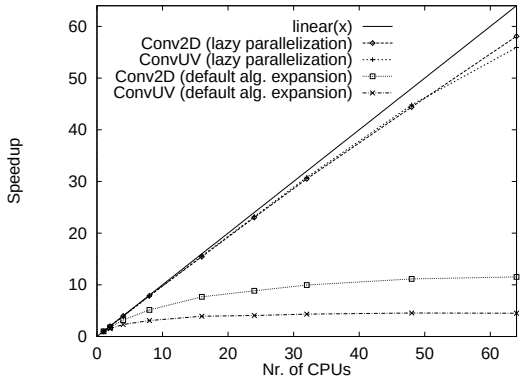
In contrast, applying lazy parallelization to the two algorithms results in minimal communication overhead. In the first loop iteration `OriginalIm` is scattered such that each node obtains a non-overlapping slice of the image's domain. Next, all subsequent operations are performed in parallel, only requiring border exchange communication in the convolutions (note: this is due to a sequential library design choice, see [25]). Finally, just before program termination, `ResultIm` is gathered to the root. In this manner, communication behavior is optimal with respect to the abstraction level of message passing programs.

A.3 Performance Evaluation

From the description, it is clear that the *Conv2D* algorithm is expected to be the slowest sequential implementation, due to the excessive accessing of image pixels in the 2-dimensional convolution operations. Figure 2(a) shows that this expectation indeed is confirmed by the measurements obtained on a single CPU. Although *Conv2D* has a slightly better speedup characteristic due to a better computation versus communication

# CPUs (1 CPU per node)	**Lazy Parallelization**		**Default Alg. Expansion**	
	Conv2D (s)	**ConvUV**(s)	**Conv2D** (s)	**ConvUV**(s)
1	425.115	185.889	425.115	185.889
2	213.358	93.824	237.450	124.169
4	107.470	47.462	133.273	79.847
8	54.025	23.765	82.781	60.158
16	27.527	11.927	55.399	47.407
24	18.464	8.016	48.022	45.724
32	13.939	6.035	42.730	43.050
48	9.576	4.149	38.164	40.944
64	7.318	3.325	36.851	41.265

(a)

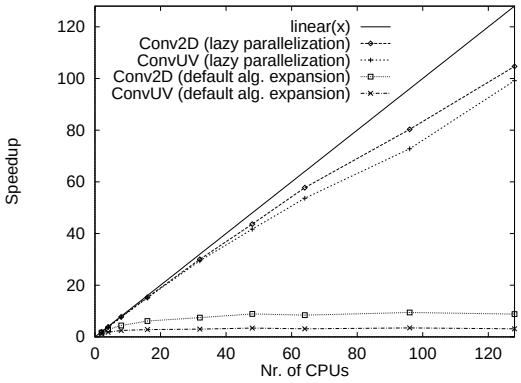


(b)

Fig. 2. *(a) Performance and (b) speedup characteristics for computing a typical orientation scale-space at 5° angular resolution (i.e., 36 orientations) and 8 (σ_u, σ_v) combinations. Scales computed are $\sigma_u \in \{3,5,7\}$ and $\sigma_v \in \{1,2,3\}$, ignoring the isotropic case $\sigma_{u,v} = \{3,3\}$. Image size is 512×512 (4-byte) pixels. Results obtained using 1 CPU per dual node.*

# CPUs (2 CPUs per node)	Lazy Parallelization		Default Alg. Expansion	
	Conv2D (s)	**ConvUV**(s)	**Conv2D** (s)	**ConvUV**(s)
2	217.366	99.587	256.425	149.575
4	109.710	50.233	148.766	101.974
8	54.801	24.550	96.134	75.084
16	27.802	12.297	69.378	65.595
32	14.150	6.313	57.032	61.984
48	9.737	4.460	47.884	54.447
64	7.363	3.464	50.529	59.649
96	5.294	2.553	45.025	53.610
128	4.062	1.875	48.148	59.169

(a)



(b)

Fig. 3. *(a) Performance and (b) speedup characteristics as in Figure 2. Results obtained using 2 CPUs per dual node.*

ratio, *ConvUV* always is the fastest implementation on any number of nodes.

The speedup graph of Figure 2(b) shows the importance of the lazy parallelization approach. Speedup values obtained on 64 nodes are 58.1 and 55.9 for *Conv2D* and *ConvUV* respectively, in case of lazy parallelization. These values drop to 11.5 and 4.5 in case of the original approach of default algorithm expansion.

Figure 3 shows similar results for measurements obtained in case both CPUs on each node are used in the execution. Even measurements for up to 128 CPUs deliver close to linear speedup. In this situation, however, performance is slightly degraded by the fact that two CPUs on a single node need to pass messages through the same communication port. Nonetheless, we can conclude that the application of lazy parallelization enables our software architecture to produce highly efficient parallel code for these implementations.

B. Rectangular Size Distributions

As discussed in [1], *rectangular size distributions* are an effective way to characterize visual similarities between document images. Here, the vertically and horizontally aligned regions of varying aspect ratios in a document image are characterized using multivariate, rectangular granulometries. A granulometry can be thought of as a morphological sieve, where objects not conforming to a particular size and shape are removed at each level of the sieving process. The rectangular granulometry, $\Psi_{x,y}$, of input image S is given by

$$\Psi_{x,y}(S) = S \circ (yV \oplus xH),$$

where H and V are the horizontal and vertical line segments of unit length centered at the origin, and x and y are independent scale parameters controlling the width and height of the rectangle used for filtering. Of most interest in describing the visual appearance are the *measurements* taken on the filtered images $\Psi_{x,y}(S)$. One useful measurement for granulometries is the rectangular size distribution. The rectangular size distribution induced by the granulometry $G = \{\Psi_{x,y}\}$ on image S is given by:

$$\Phi_G(x, y, S) = \frac{A(S) - A(\Psi_{x,y}(S)))}{A(S)},$$

$A(X)$ denoting the area of set X. As such, $\Phi_G(x, y, S)$ is the probability that an arbitrary pixel in S is filtered by a rectangle of size $x \times y$ or smaller.

B.1 Sequential Implementation

To obtain particularly efficient *sequential* code for generating rectangular size distributions, we have taken advantage of several properties of rectangular granulometries and size distributions. First, each rectangular filter is decomposed into 1-dimensional filters, eliminating the need to filter a document by rectangles of all sizes. Next, the need to use filters increasing linearly in size is removed by applying linear distance transforms for horizontal and vertical directions. These transforms are implemented by using recursive

```
calculateRectangularSizeDistribution(IMAGE inIm, INT w, INT h) {
  vertIm = verDist(inIm, 0);
  area = reduceOp(inIm, "sum");
  FOR (y=0; y≤h; y++) DO
    oy = (y/2h)*(inIm.height+1);
    vThreshIm = horDist(binPixOpC(vertIm, oy, "greaterthan"), 0);
    filtered = -1;
    FOR (x=0; x≤w; x++) DO
      IF (filtered != 1.0) THEN
        ox = (x/2w)*(inIm.width+1);
        hThreshIm = binPixOpC(vThreshIm, ox, "lessequal");
        hThreshIm = binPixOpC(verDist(hThreshIm, MAXVAL), oy, "greaterthan");
        hThreshIm = binPixOpC(horDist(hThreshIm, MAXVAL), ox, "lessequal");
        filtered = (area - reduceOp(hThreshIm, "sum")) / area;
      FI
      ... and save 'filtered' for current x,y combination ...
    OD
  OD
}
```

Listing 5: *Condensed pseudo code for fast calculation of rectangular size distributions; maximum size of calculated filters denoted by 'w' and 'h'. Functions 'horDist' and 'verDist' perform horizontal and vertical distance transforms, using recursive filter-pairs.*

forward/backward filter pairs. Lastly, the need to explore large, flat regions of the size distributions is eliminated by halting the filtering for the current filter when its properties guarantee that the filtered result will be identical.

Pseudo code for the presented problem is given in Listing 5. It should be noted that the use of recursive filters results in a implementation which is notoriously hard to parallelize (as is shown in the results provided in the remainder of this section). A less efficient sequential solution would be to use sieving without decomposition. This boils down to a morphological scale-space, and is comparable to the application of Section V-A.

B.2 Parallel Execution

As before, the sequential code of Listing 5 directly constitutes a parallel program as well. When applying default algorithm expansion for parallelization, the program suffers from the same problem as the application described in Section V-A: it results in execution of many costly `Scatter` and `Gather` operations. Lazy parallelization avoids all such redundant communication steps automatically, and again results in optimal communication behavior with respect to the abstraction level of message passing programs. In effect, the input image is scattered throughout the parallel system only once, and no additional communication steps are required for resolution of data structure state inconsistencies.

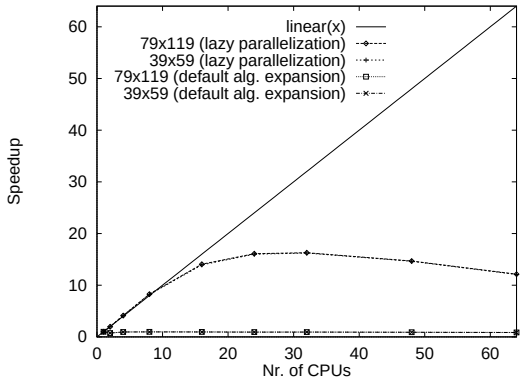
It should be noted, however, that speedup characteristics are not expected to be as good as those presented in Section V-A. This is because the applied recursive filter operations are hard to parallelize efficiently. In our library we apply a fast two-step redistribution of the partitioned image data to always match the horizontal and vertical filtering directions. Although this approach does result in fast parallel execution, we are aware of the fact that additional optimizations are possible (such as the application of a multi-partitioning technique [6]). This part of the pre-parallelized code is not affected by lazy parallelization, however, as data redistribution plays no role in the introduction or removal of data structure state inconsistencies.

B.3 Performance Evaluation

Measurement results for the two generated parallel versions of the presented algorithm are given in Figure 4. It should be noted that these results represent a lower bound on

# CPUs (1 CPU per node)	Lazy Parallelization		Default Alg. Expansion	
	'39x59' (s)	**'79x119'** (s)	**'39x59'** (s)	**'79x119'** (s)
1	41.975	157.439	41.975	157.439
2	21.297	80.279	55.955	209.964
4	10.097	38.174	44.157	166.163
8	5.109	19.029	43.441	160.874
16	3.014	11.198	44.235	163.865
24	2.621	9.778	45.462	167.792
32	2.587	9.673	45.319	167.566
48	2.870	10.732	47.201	174.986
64	3.476	12.984	49.283	183.054

(a)



(b)

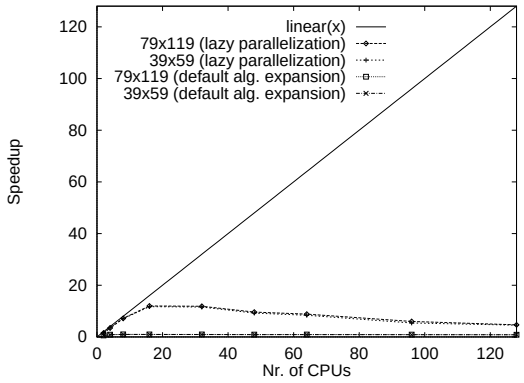
Fig. 4. *(a) Performance and (b) speedup for computing rectangular size distributions for document image of size 350×517 (2-byte) pixels. Maximum size of calculated filters either 39×59 or 79×119. Results obtained using 1 CPU per dual node. Note: speedup lines for either approach essentially coincide.*

the obtainable speedup for this application, as the size of the input images was reduced to 350×517 pixels only. As can be seen in Figure 4(a), lazy parallelization results in significant performance gains for any applied number of processors. In contrast, default algorithm expansion behaves badly, and even results in a performance drop at all times.

Figure 4(b) shows that the maximum number of nodes that can be used effectively for such a small-sized input image is about 32. Even though lazy parallelization has resulted in the removal of all redundant communication, the cost of the communication steps applied in the recursive filter operations is significant in case the number of processors becomes large. Still, the differences in the execution times for the two parallelization strategies are enormous, and clearly show the importance of redundant communication removal.

# CPUs (2 CPUs per node)	Lazy Parallelization		Default Alg. Expansion	
	'39x59' (s)	**'79x119'** (s)	**'39x59'** (s)	**'79x119'** (s)
2	28.040	104.443	74.211	272.451
4	12.066	45.055	48.145	179.605
8	5.933	21.898	43.330	159.686
16	3.578	13.122	43.163	161.686
32	3.627	13.267	43.969	164.093
48	4.536	16.375	46.358	171.133
64	5.008	17.839	45.871	167.999
96	7.769	26.295	47.397	173.023
128	9.207	33.589	50.003	183.948

(a)

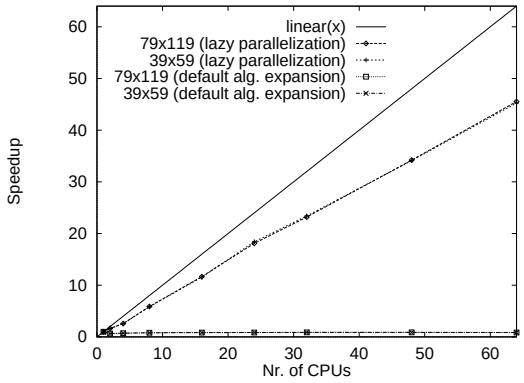


(b)

Fig. 5. *(a) Performance and (b) speedup as in Figure 4. Results obtained using 2 CPUs per dual node. Note: speedup lines for either approach essentially coincide.*

# CPUs (1 CPU per node)	Lazy Parallelization		Default Alg. Expansion	
	'39x59' (s)	**'79x119'** (s)	**'39x59'** (s)	**'79x119'** (s)
1	1263.277	4589.818	1263.277	4589.818
2	779.486	2821.391	1985.855	6941.262
4	497.145	1789.319	1749.873	6412.121
8	216.679	781.175	1551.239	5593.406
16	109.284	393.763	1505.077	5452.276
24	68.652	253.647	1478.558	5308.092
32	53.967	197.962	1453.866	5278.840
48	37.048	134.180	1454.710	5245.239
64	27.926	100.792	1489.291	5340.271

(a)



(b)

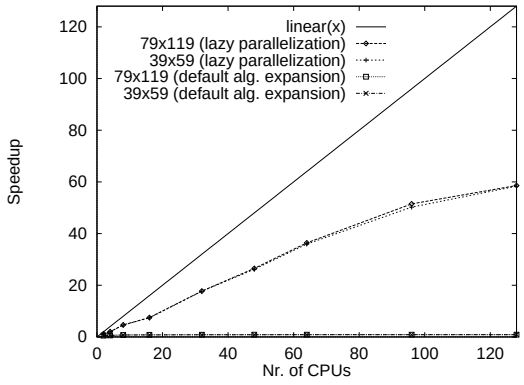
Fig. 6. *(a) Performance and (b) speedup for computing rectangular size distributions for document image of size 2325×3075 (2-byte) pixels. Maximum size of calculated filters either 39×59 or 79×119. Results obtained using 1 CPU per dual node. Note: speedup lines for either approach essentially coincide.*

Figure 5 shows similar results in case both CPUs on each node are used in the execution. As each dual node can communicate through one port only, communication overhead has increased in comparison to the results presented in Figure 4. As a result, the maximum number of processors that can be used effectively is now reduced to only 16.

Figure 6 shows that, for a much more realistic input image of size 2325×3075 pixels, lazy parallelization still provides very good speedup characteristics: 45.5 on 64 processors — an efficiency of 71.2%. As before, default algorithm expansion does not deliver any performance gains at all. Figure 7 shows similar results in case of using both CPUs on each node. Given these results, we conclude that lazy parallelization also generates efficient parallel code for the presented rectangular size distribution extraction algorithm.

# CPUs (2 CPUs per node)	Lazy Parallelization		Default Alg. Expansion	
	'39x59' (s)	**'79x119'** (s)	**'39x59'** (s)	**'79x119'** (s)
2	1038.741	3753.409	2480.137	9009.303
4	620.755	2278.433	2110.848	7735.314
8	275.947	986.406	1668.469	6056.997
16	170.621	613.643	1574.845	5691.093
32	71.430	258.796	1477.051	5357.002
48	48.265	173.008	1470.206	5301.129
64	35.155	126.047	1449.308	5227.423
96	25.145	89.133	1453.676	5196.682
128	21.655	78.356	1411.593	5160.945

(a)



(b)

Fig. 7. *(a) Performance and (b) speedup as in Figure 6. Results obtained using 2 CPUs per dual node. Note: speedup lines for either approach essentially coincide.*

C. Performance Comparison with Related Tools

In [27] we have made a performance comparison between our software architecture and several related tools described in the literature. The comparison is based on a well-known stereo vision application which — in its parallel behavior — is comparable to the line detection application of Section V-A. The following briefly presents the main results.

First, a comparison is made with results obtained for the stereo vision application written in a specialized parallel programming language (SPAR [24]), which was executed on the same parallel machine as used in the above evaluation. Also, the codes generated by the SPAR front-end and that of our own architecture were compiled in an identical manner. Measurements showed our architecture to provide superior sequential performance of about a factor 5, and better speedup — clearly indicating that the overhead from our lazy parallelization approach is much smaller than that of the SPAR run time system.

Second, a comparison is made with results obtained for an implementation in the Adapt parallel image processing language [34]. A true comparison with this work turned out to difficult, however, as the results were obtained on a significantly different machine (i.e., a collection of iWarp processors, with a better potential for obtaining high speedup than our DAS cluster). Even so, our software architecture showed superior performance (of about a factor 2) with comparable speedup characteristics over a large range of processors.

Most interesting, however, is the comparison with *Easy-PIPE* [20], a library-based software environment for parallel image processing similar to ours. The most distinctive feature of this architecture is that it incorporates a mechanism for combining data and task parallelism. Also, *Easy-PIPE* does not shield *all* parallelism from the application programmer. As a consequence from these differences, *Easy-PIPE* has the potential of outperforming our architecture, which is fully user transparent, and strictly data parallel. However, performance and speedup characteristics for the stereo vision application obtained on the very same DAS cluster show that our implementations far better exploit the available parallelism than *Easy-PIPE*. Part of the difference is accounted for by the fact that *Easy-PIPE* does not incorporate an explicit inter-operation optimization mechanism for removal of redundant communication. In addition, the run time parallelization overhead of *Easy-PIPE* turned out to be much higher than that of our software architecture.

VI. Related Work

For obtaining efficient library-based parallel image processing applications, the importance of inter-operation optimization has been acknowledged before. Morrow et al. [19] describe an environment for data parallel image processing similar to ours. One of the important features of this environment is its *self-optimizing class library*, which is extended automatically with optimized parallel operations. During program execution, a syntax graph is constructed for each statement in the program, and evaluated only when an assignment operator is met. At first execution of a program, each syntax graph is traversed, and an instruction stream is generated and executed. In addition, any syntax graph for combinations of primitive instructions is written out for later consideration by an off-line optimizer. On subsequent runs of the program a check is made to decide if an optimized routine is available for a given sequence of library calls. In comparison with lazy parallelization, this optimization approach has several drawbacks. First, the optimization process is performed at compile-time only, and has inherent problems with data-dependent conditionals and loop constructs. Next, optimized performance is obtained only for runs following the initial execution of a program. Finally, the approach may guarantee optimal performance of sequences of library routines, but not necessarily of complete programs. It should be noted that the approach of Lee et al. [15] is quite similar to that of Morrow et al.; as a consequence it suffers from the very same problems as well.

A related approach to obtaining efficient code for library-based scientific applications is the concept of Telescoping Languages introduced by Kennedy et al. [12]. In this approach, high performance for full applications is achieved by exhaustively analyzing and precompiling a given library — which is annotated with domain-specific optimizations that should not be discovered unaided — to produce a processor that recognizes and optimizes library operations as primitives in a domain-specific language. The goal of precompilation is to specialize different versions of each library routine for sets of conditions that hold when the routine is invoked. The entire set of specialized routines is collected in a database that permits efficient code selection and inlining when full applications are compiled. Although many other forms of optimization are incorporated (a.o.: self-tuning for portability, which is comparable to our ASTG-approach referred to in Section IV-A), of most relevance to this

paper is the fact that the Telescoping Languages approach also considers combinations of library operations on data structures for multiple distribution types. In comparison to lazy parallelization, however, the presented approach has several disadvantages. First, as in the approach of Morrow et al. described above, optimization is performed at compile-time only, resulting in difficulties with data-dependent conditionals and loops. Moreover, the required precompilation can be extremely time-consuming, and results in a large database of operations from which only a few routines will generally be invoked at run time. Also, to be able to deal with different shapes and sizes of data structures (which generally remain unknown until run time), the database of alternative implementations is extended even further. Although it has not been emphasized so much before, lazy parallelization can easily deal with this problem by remaining flexible in the number of nodes to be used, and by allowing for run time selection of a single state transition from a set of multiple alternatives, depending on a structure's size and shape. As indicated in [25], this solution has been integrated cleanly and elegantly, and without measurable run time overhead.

To our knowledge, usage of fsm specifications is new in the field of library-based parallelization tools. Moreover, the application of an fsm definition seems not to have been considered at all in the field of parallel image processing. In related research areas of parallel computation, however, fsm definitions have been applied before. For example, Chatterjee et al. [4] apply a finite state machine for the generation of optimal communication sets in distributed-memory implementations of data-parallel languages such as HPF. As in our case, results indicate that the fsm approach requires very little run time overhead. For ad-hoc optimization of specific algorithms and applications fsm definitions have been applied successfully as well [5], [18].

Interestingly, our approach to finding optimal performance of operations as well as complete applications is related to several projects in other domains. The SPIRAL project [23], [30], for example, is aimed at the design of a system to generate efficient libraries for digital signal processing algorithms. SPIRAL generates efficient implementations of algorithms expressed in a domain-specific language, called SPL, by a systematic search through the space of possible implementations. Other efforts in automatically generating efficient implementations of programs include FFTW [7] for adaptively generating

time-optimal FFT algorithms, and the ATLAS project [35] for deriving efficient implementations of basic linear algebra routines.

Finally, our work shares common goals with that of Baumgartner et. al. [3], in the search of an optimal data partitioning strategy with minimal communication overhead for applications in the field of quantum chemistry and physics. As in our extended approach not discussed here, an operator tree is generated, in which multiple data partitioning and communication strategies are incorporated. This approach is also entirely static, however, and includes no possibility for partial optimization performed at run time.

VII. Conclusions

In this paper we have presented a finite state machine based approach for communication minimization of data parallel regular domain problems. The approach, referred to as lazy parallelization, considers a sequential program, which is parallelized automatically by inserting communication operations and local memory management operations whenever necessary. The approach always generates a legal, correct, and efficient parallel version of any sequential program implemented on the basis of so-called *parallelizable patterns*, where each such pattern represents a generic description of a class of sequential algorithms with similar behavior in terms of data accesses to array-like structures.

The main advantage of the optimization approach is that it can be applied on the fly at run time. As all required data accesses are defined for each operation, decisions regarding inter-process communication can be deferred to the actual moment of intended execution. As such, lazy parallelization is very easy to implement, and performs without measurable run-time overhead. In comparison with other methods described in the literature, lazy parallelization requires no prior knowledge regarding the behavior of loops and branches, and run-time adaptation to data structure shapes and sizes is easily integrated [25].

In conclusion, lazy parallelization on the basis of a finite state machine specification has proven to constitute a surprisingly simple, yet effective method for global optimization of data parallel regular domain problems. Essentially, the simplicity stems from the knowledge contained in the definition of parallelizable patterns, and from the high level abstractions incorporated in the finite state machine definition. Consequently, we feel that the applicability of the approach extends beyond the domain of library-based low

level image processing applications. This is particulary true for the domains of signal processing and linear algebra applications, which include similar patterns of communication and calculation.

References

[1] A.D. Bagdanov and M. Worring. Multi-scale Document Description using Rectangular Granulometries. In *Document Analysis Systems V, LNCS 2423*, pages 445–456, August 2002.

[2] H.E. Bal et al. The Distributed ASCI Supercomputer Project. *Operating Systems Review*, 34(4):76–96, October 2000.

[3] G. Baumgartner et al. A High-Level Approach to Synthesis of High-Performance Codes for Quantum Chemistry. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 1–10, Baltimore, Maryland, USA, November 2002.

[4] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating Local Addresses and Communication Sets for Data Parallel Programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, April 1995.

[5] J.M. Constantin, M.W. Berry, and B.T. Vander Zanden. Parallelization of the Hoshen-Kopelman Algorithm Using a Finite State Machine. *International Journal of Supercomputer Applications and High Performance Computing*, 11(1):31–45, Spring 1997.

[6] A. Darte, D. Chavarría-Miranda, R. Fowler, and J. Mellor-Crummey. Generalized Multipartitioning for Multi-dimensional Arrays. In *Proceedings of the 16th International Parallel & Distributed Processing Symposium*, Fort Lauderdale, Florida, USA, April 2002.

[7] M. Frigo and S.G. Johnson. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. International Conference on Acoustics, Speech, and Signal Processing*, pages 1381–1384, Seattle, USA, May 1998.

[8] J.M. Geusebroek, A.W.M. Smeulders, and H. Geerts. A Minimum Cost Approach for Segmenting Networks of Lines. *Int. Journal of Computer Vision*, 43(2):99–111, July 2001.

[9] J.E. Hopcroft, R. Motwani, and J.D. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, 2000.

[10] L.H. Jamieson, E.J. Delp, C.-C. Wang, J. Li, and F.J. Weil. A Software Environment for Parallel Computer Vision. *IEEE Computer*, 25(2):73–75, February 1992.

[11] Z. Juhasz and D. Crookes. A PVM Implementation of a Portable Parallel Image Processing Library. In *Proceedings of EuroPVM'96*, pages 188–196, Munich, Germany, October 1996.

[12] K. Kennedy et al. Telescoping Languages: A Strategy for Automatic Generation of Scientific Problem-Solving Systems from Annotated Libraries. *Journal of Parallel and Distributed Computing*, 61:1803–1826, 2001.

[13] D. Koelma, P.P. Jonker, and H.J. Sips. A Software Architecture for Application Driven High Performance Image Processing. In *Parallel and Distributed Methods for Image Processing, Proceedings of SPIE*, volume 3166, pages 340–351, San Diego, California, USA, July 1997.

[14] C. Lee and M. Hamdi. Parallel Image Processing Applications on a Network of Workstations. *Parallel Computing*, 21(1):137–160, January 1995.

[15] C. Lee, Y.-F. Wang, and T. Yang. Global Optimization for Mapping Parallel Image Processing Tasks on Distributed Memory Machines. *Journal of Parallel and Distributed Computing*, 45(1):29–45, August 1997.

[16] P. Maurer. Logic Simulation Using Networks of State Machines. In *Proceedings of Design, Automation and Test in Europe Conference 2000 (DATE 2000)*, pages 674–678, Paris, France, March 2000.

[17] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard (version 1.1). Technical report, University of Tennessee, Knoxville, Tennessee, June 1995. Available at `http://www.mpi-forum.org`.

[18] D. Milicev and Z. Jovanovic. A Finite State Machine Based Formal Model of Software Pipelined Loops with Conditions. *International Journal of Computer Research*, 10(1):11–20, 2001.

[19] P.J. Morrow, D. Crookes, J. Brown, G. McAleese, D. Roantree, and I. Spence. Efficient Implementation of a Portable Parallel Programming Model for Image Processing. *Concurrency: Practice and Experience*, 11:671–685, September 1999.

[20] C. Nicolescu and P. Jonker. EASY-PIPE - An Easy to Use Parallel Image Processing Environment Based on Algorithmic Skeletons. In *Proceedings of the 15th International Parallel & Distributed Processing Symposium*, San Francisco, California, USA, April 2001.

[21] C. Nicolescu and P. Jonker. A Data and Task Parallel Image Processing Environment. *Parallel Computing*, 28(7–8):945–965, August 2002.

[22] M. Prieto, I.M. Llorente, and F. Tirado. Data Locality Exploitation in the Decomposition of Regular Domain Problems. *IEEE Transactions on Parallel and Distributed Systems*, 11(11):1141–1149, November 2000.

[23] M. Püschel, B. Singer, M. Veloso, and J. Moura. Fast Automatic Generation of DSP Algorithms. In *Proceedings of the International Conference on Computational Science, LNCS 2073*, pages 97–106, 2001.

[24] C. van Reeuwijk, A.J.C. van Gemund, and H.J. Sips. Spar: A Programming Language for Semi-Automatic Compilation of Parallel Programs. *Concurrency: Practice and Experience*, 9(11):1193–1205, November 1997.

[25] F.J. Seinstra. *User Transparent Parallel Image Processing.* PhD thesis, Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, The Netherlands, May 2003.

[26] F.J. Seinstra and D. Koelma. P-3PC: A Point-to-Point Communication Model for Automatic and Optimal Decomposition of Regular Domain Problems. *IEEE Transactions on Parallel and Distributed Systems*, 13(7):758–768, July 2002.

[27] F.J. Seinstra and D. Koelma. User Transparency: A Fully Sequential Programming Model for Efficient Data Parallel Image Processing. *Concurrency and Computation: Pract. Exper.*, 16(6):611–644, May 2004.

[28] F.J. Seinstra, D. Koelma, and A.D. Bagdanov. On the Correctness of Lazy Parallelization. Technical Report Series, Vol. 2004-01, Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, The Netherlands, March 2004.

[29] F.J. Seinstra, D. Koelma, and J.M. Geusebroek. A Software Architecture for User Transparent Parallel Image Processing. *Parallel Computing*, 28(7–8):967–993, August 2002.

[30] B. Singer and M. Veloso. Learning to Construct Fast Signal Processing Implementations. *Journal of Machine Learning Research*, 3:887–919, December 2002.

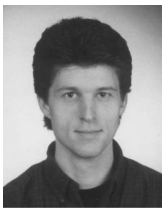
[31] C. Soviany. *Embedding Data and Task Parallelism in Image Processing Applications.* PhD thesis, Delft University of Technology, The Netherlands, May 2003.

[32] J.M. Squyres, A. Lumsdaine, and R.L. Stevenson. A Toolkit for Parallel Image Processing. In *Parallel and Distributed Methods for Image Processing II, Proc. SPIE*, volume 3452, San Diego, USA, July 1998.

[33] R. Taniguchi et al. Software Platform for Parallel Image Processing and Computer Vision. In *Parallel and Distributed Methods for Image Processing, Proc. SPIE*, volume 3166, pages 2–10, San Diego, USA, July 1997.

[34] J.A. Webb. Implementation and Performance of Fast Parallel Multi-Baseline Stereo Vision. In *Proceedings of the 1993 DARPA Image Understanding Workshop*, pages 1005–1010, April 1993.

[35] R.C. Whaley, A. Petitet, and J.J. Dongarra. Automated Empirical Optimization of Software and the ATLAS Project. *Parallel Computing*, 27(1–2):3–25, January 2001.



Frank Seinstra received the MS degree in Computer Science from the Vrije Universiteit in Amsterdam in 1996, and the PhD degree in Computer Science from the University of Amsterdam in 2003. The subject of his PhD thesis is "User Transparent Parallel Image Processing". His research interests include parallel and distributed programming, automatic parallelization, performance modeling, and scheduling, especially in the application area of image and video processing.

Dennis Koelma received the MS and PhD degrees in Computer Science from the University of Amsterdam in 1989 and 1996, respectively. The subject of his PhD thesis is "A Software Environment for Image Interpretation". Currently, he is working on Horus: a software architecture for research in accessing the content of digital images. His research interests include image and video processing, software architectures, parallel programming, databases, graphical user interfaces, and image information systems.

Andrew Bagdanov received the BS and MS degrees in Mathematics and Computer Science from the University of Nevada, Las Vegas, where he was a member of the Information Science Research Institute. He is currently finishing his PhD thesis in Computer Science (title: "Style Characterization of Machine Printed Texts") at the University of Amsterdam. His research interests include document understanding, pattern recognition, image processing, and functional programming languages.