Lazy Parallelization: A Finite State Machine Based Optimization Approach for Data Parallel Image Processing Applications

F.J. Seinstra and D. Koelma
Intelligent Sensory Information Systems,
Faculty of Science, University of Amsterdam,
Kruislaan 403, 1098 SJ Amsterdam, The Netherlands
(fjseins, koelma)@science.uva.nl

Abstract

Performance obtained with existing library-based parallelization tools for implementing high performance image processing applications is often sub-optimal. This is because inter-operation optimization (or: optimization across library calls) is often not incorporated in the library implementations. This paper presents a simple, efficient, finite state machine-based method for global performance optimization, called 'lazy parallelization'. Experimental results based on this approach show significant performance improvements over non-optimized parallel implementations.

1. Introduction

A parallelization tool based on a software library of parallel routines can serve as a powerful programming aid to obtain high performance in image processing research. Due to the relative ease of implementation, many such parallelization tools exist [6, 7, 8, 9, 13, 20, 21]. These tools, however, generally restrict performance optimization to each library operation *in isolation*, and ignore global optimization for full applications. For library implementations based on message passing primitives, global optimization can provide significant performance gains as it is often possible to (1) remove many redundant communication steps, and (2) combine multiple messages in a single transfer.

Automatic optimization of communication overhead is not easy. 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 that of the original program.

In this paper we present a new, and surprisingly simple strategy for global performance optimization, which adheres to these requirements. In the approach, a *fully sequential* program is parallelized automatically by inserting communication 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. In this paper it is shown how to apply the fsm specification in order to obtain legal, correct, and indeed efficient parallel code.

This paper is organized as follows. Section 2 describes the optimization problem. Section 3 introduces the fsm definition. The fsm-based optimization strategy of lazy parallelization is described in Section 4, and results for a simple application are presented. In Section 5 related work is discussed. Conclusions are given in Section 6.

2 The Performance Optimization Problem

In [18] we have described our library-based software architecture for implementing data parallel image processing applications. In this architecture, *all* parallelization and optimization is shielded, which leaves the programmer with a fully sequential programming model [17]. For performance optimization, we have implemented all library routines on the basis of a high level *abstract parallel image processing machine* specification, or APIPM [18]. As a result, any imaging application is composed of a sequence of APIPM instructions. For global performance optimization it is not necessary to individually consider each of the instructions in such a sequence. Specific combinations of APIPM instructions often appear together, and are identical for sequential operation as well as for parallel execution.

Create	(	OUT	dst	);				
Delete	(	OUT	dst	);				
MemCopy	(	IN	src,	OUT	dst	);		
UnPixOp	(	IN	src,	OUT	dst	);		
BinPixOpV	(	IN	src,	OUT	dst,	IN	arg	);
BinPixOpI	(	IN	src,	OUT	dst,	IN	arg	);
ReduceOp	(	IN	src,	OUT	dst	);		
NeighOp	(	IN	src,	OUT	dst,	IN	ker	);
GenConvOp	(	IN	src,	OUT	dst,	IN	ker	);
GeoMat	(	IN	src,	OUT	dst	);		
GeoRoi	(	IN	src,	OUT	dst	);		
Import	(	OUT	dst	);				
Export	(	IN	src	);				

Table 1. *Abstract functions: sequential operation.*

CreatLclPart	(	OUT	ldst	);		
CreatLclFull	(	OUT	ldst	);		
DelLcl	(	OUT	ldst	);		
Broadcast	(	IN	gsrc,	OUT	ldst	);
Scatter	(	IN	gsrc,	OUT	ldst	);
Gather	(	IN	lsrc,	OUT	gdst	);
GatherAll	(	INOUT	lsrc,	INOUT	gdst	);
RduceOne	(	INOUT	lsrc,	OUT	gdst	);
RduceAll	(	INOUT	lsrc,	INOUT	gdst	);

Table 2. *Abstract functions: communication.*

For such 'unbreakable' APIPM instruction sequences relating to sequential processing, we have introduced a shorthand notation, presented in Table 1. These abstractions include (a.o.) unary and binary pixel operations, neighborhood operations, and geometric transformations. Notation for unbreakable instruction streams relating to interprocess communication is presented in Table 2, and contains abstractions similar to operations in MPI [11]. The additional CreatLclPart/Full and DelLcl functions constitute creators and destructors for *partial* data structures (i.e., the constituent components of a distributed data structure, each residing on a different node in the parallel machine 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 1 and 2 alone.

2.1 Default Algorithm Expansion

In our architecture we have implemented a default parallelization strategy for each library operation. Consequently, (run time) conversion of a sequential application into an equivalent parallel program is straightforward. The conversion process, referred to as *default algorithm expansion*, is illustrated by the simple example code of Listing 1. The abstract sequential program, shown on the left, first imports an image structure A, which is used as input to a unary pixel operation. Subsequently, the resulting output structure B is used as input to a binary pixel operation. Finally, the resulting image C is exported, and all images are destroyed.

The equivalent parallel program, obtained after default algorithm expansion, 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 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 is guaranteed to produce a legal and correct parallel version of any sequential program implemented using our software architecture. This is because each abstract function call in the sequential code is replaced by an equivalent sequence of one or more (parallel) operations. The resulting code is not guaranteed to be time-optimal, however. In fact, in most situations the expansion process will not even produce the fastest parallel implementation at all. Worse even, the resulting code often can be expected to be *slower* than the original sequential program. Although other tools may be implemented differently, all library-based tools suffer from the very same problem — and for improved performance a solution is essential.

```
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).*

```
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).*

When considering the parallel code of Listing 1(b), it is clear that it contains several function calls that could be removed without violating the program's correctness or legality. First, image structure 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 paper indicates how execution of such redundant operations can be avoided automatically.

3 Finite State Machine Definition

To guide the process of redundant communication avoidance we have defined a *finite state machine* (or *fsm*) which is used for operation redundancy detection, the monitoring of the lifespan of (distributed) data structures, and the resolution of data structure inconsistencies. In this paper, we restrict ourselves to so-called *deterministic finite accepters*, which have no temporary storage and which can not produce strings of output [5]. A deterministic finite accepter (*dfa*) is 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*.

3.1 Data Structure States and Lifespan

As described in [18], 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 (referred to as a *distributed data structure*). Clearly, at any time during execution of a parallel program either the global structure itself or the distributed structure derived from that global structure must contain up-to-date values for all structure elements. An abstract representation of the relationship between these structures is given by the three-tuple $q = (g,\ d,\ t)$, where

$g \in G$ is the *state of the global structure*,

$d \in D$ is the *state of the derived distributed structure*,

$t \in T$ is the *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 RduceOne or RduceAll operation.

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 finite state machine, we have specified a restricted set of *valid internal states*, based on the presented relationship between global and distributed structures. The selected set of valid internal fsm states is defined by

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

with

q_0 = (none, none, none),
q_1 = (created, none, none),
q_2 = (valid, none, none),
q_3 = (invalid, none, none),
q_4 = (valid, valid, partial),
q_5 = (valid, valid, full),
q_6 = (invalid, valid, partial),
q_7 = (invalid, 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. Essentially, 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 simply indicates that a global structure's elements contain all correct and 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 yet 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*.

3.2 State Transition Functions

The fsm *input alphabet* is formed by the abstract functions of Tables 1 and 2, 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* finite state machines. The presence of multiple state machines results in a *parallel view* of the states of all data structures in an application. At any moment during execution, several data structures are 'alive' and their combined state is captured by their respective finite state machines. As the states of multiple data 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 state machine 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 (abstract) function is annotated with a list of permitted state dependencies for all additional data structures passed as parameter to that function (i.e., those structures for which the current fsm is not responsible). Here, we represent elements in

$\delta(q_0, (\text{Create}, -)) = q_1$, $\quad \delta(q_i, (\text{Delete}, -)) = q_0$, $\delta(q_0, (\text{Import}, -)) = q_2$, $\quad \delta(q_j, (\text{Export}, -)) = q_j$, 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 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 op $\in \{\text{BinPixOpV, NeighOp, GenConvOp}\}$, $i \in \{5, 7\}, j \in \{4, 5, 6, 7\}$,
$\delta(q_0, (\text{op}, q_2, q_2)) = q_2$, $\quad \delta(q_2, (\text{op}, q_0, q_2)) = q_2$, $\delta(q_0, (\text{op}, q_i, q_j)) = q_6$, $\quad \delta(q_i, (\text{op}, q_0, q_j)) = q_i$, $\delta(q_0, (\text{op}, q_k, q_l)) = q_7$, $\quad \delta(q_k, (\text{op}, q_0, q_l)) = q_k$, with 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 op $\in \{\text{GeoMat, GeoRoi}\}, i \in \{5, 7\}$.

Table 3. *Transition functions: image operations.*

Σ_d by a two- or three-tuple, in which the first component is the name of the abstract 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 of state machines also could have been captured in a single *cross-product machine*, in which each deterministic finite automaton simulates, in parallel, the behavior of each component dfa (e.g., see [10]). For simplicity, however, in the remainder of this paper we keep to the parallel view of simple state machines.

Table 3 presents the transition functions for the image operations available in our library. The overview is complete in the sense that our implementations allow no state transitions other than the ones presented here. In all cases, initial state q_0 refers to the state of the output structure produced by any of the operations (represented by an OUT parameter in Table 1). 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 transition functions that cause a structure to be moved to state q_2 indicate sequential execution using global structures. All other transition functions 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 4. In all of these the list of state dependencies is empty, as the functions work on a single data structure only.

Figure 1 presents a reduced state transition graph for our fsm definition. For better readability, the graph 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 data structure can reach at run time. Also, it is exactly these operations that play an essential role in the process of operation redundancy avoidance as presented in Section 4.

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

Table 4. *Transition functions: communication.*

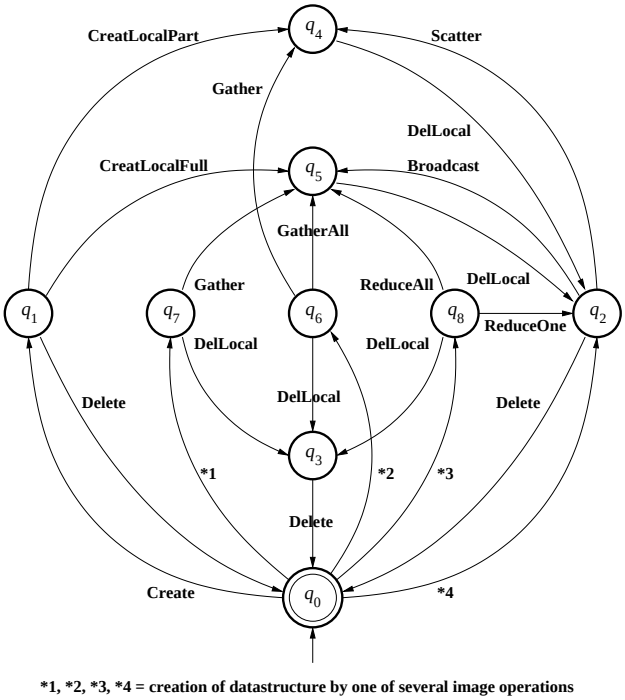

Figure 1. *Reduced state transition graph.*

A program is a *legal program*, if it is accepted by *all* finite state machines related to that program. In other words, a program is legal if (1) it consists of a sequence of abstract function calls from Tables 1 and 2 only, (2) it contains no data structure state inconsistencies, and (3) all data structures start as well as end in the empty state q_0. In case a user-provided sequential program is accepted as a legal program, the process of 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 finite state machine is used to obtain legal and correct parallel code, which is optimized in that the execution of any redundant communication operations is avoided.

4 Lazy Parallelization

In the approach of *lazy parallelization* it is simply assumed that *each* communication or memory management operation inserted by default algorithm expansion is redundant, unless proven otherwise. Stated differently, lazy parallelization causes a default communication or memory management operation to be executed only, in case its removal would introduce an (immediate) data structure state inconsistency. Although lazy parallelization can be applied on the fly at run time, for the moment we will present it as a com-

pile time method. Conceptually, lazy parallelization consists of the following parallelization and optimization steps:

1. Apply the process of default algorithm expansion to the original sequential code.
2. Scan the expanded code, and 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 all introduced data structure state inconsistencies by re-inserting operations removed in step 2.
5. Undo the partial loop unrolling by replacing all separated loops by a single combined code block.

As stated, the code obtained after the first step consists of legal, but non-optimal parallel code. The operation removal in the second step, however, introduces many state inconsistencies. These are resolved in step four. As will be described below, in this step any illegal parallel code is transformed to legal code by (re-)inserting operations to resolve data structure state inconsistencies. Steps 3 and 5 are present only to deal with loop constructs which may be present in the user-provided code. The extraction of the first iteration of a loop (*partial loop unrolling*) exposes all data structure state inconsistencies that can possibly occur in a program. More specifically, loop unrolling makes it possible to compare (1) the data structure states reached after execution of the pre-loop code with the states required in the first loop iteration, (2) the states reached after execution of the n-th loop iteration with the states required in iteration $n + 1$, and (3) the states reached after execution of the last loop iteration with the states required in the post-loop code.

Listing 3 gives an example of the application of lazy parallelization. The abstract code for a simple example program is shown in the Listing 3(a). The programs obtained in the first three steps of the optimization process are all straightforward, and will not be explained any further.

The re-insertion of code as applied in step 4 (see Listing 3(e)) is performed using the state transition functions of Section 3.2 (i.e., only those incorporated 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 3). The only available operation that provides a resolution to this state inconsistency is the Broadcast operation, 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 operation re-insertions work in a similar manner, and all further interpretation of Listing 3 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) example 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: *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.*

4.1 Discussion

Lazy parallelization produces legal and correct parallel code at all times. This can be seen by considering the allowed states for all data structures passed as parameters to the operations in Table 1, and the resulting states for the output structures produced by these operations. As such, each operation has a set of *allowed input states* for each of its parameters, and one of these is moved to a new *output state*.

By exhaustion, it is easily shown that for each possible output state, a sequence of zero or more state transitions exists that moves a data structure from that particular output state to one state in each set of allowed input states.

An important property of the approach is that it can be applied on the fly at run time (hence its name). Because the required data structure states are known for each operation, it is possible to defer decisions regarding the execution of each default communication operation to as late as the actual moment of intended execution. Essentially, this means that all five steps as described above, are reduced to a single step. As such, lazy parallelization is unrestrictive and highly efficient, as no prior knowledge regarding the behavior of loops and branches is required. This knowledge is simply obtained during execution of the application, and is not required any sooner.

It should be noted that, although lazy parallelization works well in many situations, it does not guarantee to always produce the *fastest possible* version of a program under consideration. This is because the approach always applies the fastest communication step whenever message transfer is mandatory. This is still a form of *local* performance optimization, however, as it may be better to insert a *combined* message transfer to avoid further communication steps to be executed at a later stage. To overcome this problem, we have also implemented an extension to the approach of lazy parallelization, which is indeed capable of producing the (expected) fastest parallel version of any sequential program. The extended approach relies on the creation of an *application state transition graph (ASTG)*, which incorporates all performance optimization decisions which can possibly made at application run time. Each decision is annotated with a run time cost estimation, such that the fastest version of the program is represented by the cheapest branch in the graph. Drawback of the extended approach, however, is that it is often costly to actually obtain the cheapest branch. More detailed information related to these issues is presented in [15, 16, 18].

```
FOR i=0:NrImages-1 DO
  InputIm = ReadFile(...);
  SqrdInputIm = BinPixOp(InputIm, ”mul”, InputIm);
  FOR j=0:NrSymbols-1 DO
    IF (i==0) THEN
      weights[j] = ReadFile(...);
      symbols[j] = ReadFile(...);
      symbols[j] = BinPixOp(symbols[j], ”mul”, weights[j]);
    FI
    FiltIm1 = GenConvOp(SqrdInputIm, ”mult”, ”add”, weights[j]);
    FiltIm2 = GenConvOp(InPutIm, ”mult”, ”add”, symbols[j]);
    FiltIm2 = BinPixOp(FiltIm2, ”mult”, 2);
    ErrorIm = UnPixOp(FiltIm1, ”sub”, FiltIm2);
    WriteFile(ErrorIm);
  OD
OD
```

Listing 4: *Pseudo code for template matching.*

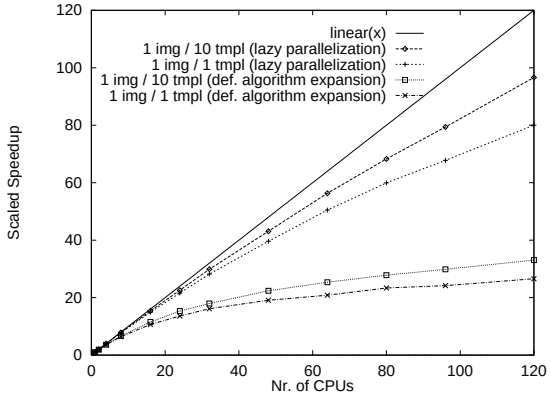

Figure 2. *Speedup for template matching: input image of 1093×649 pixels, templates of size 41×35.*

4.2 Performance Evaluation

We have applied the approach of lazy parallelization to the simple template matching program presented in Listing 4. In this application, a set of electrical engineering drawings is matched against a set of templates representing electrical components, such as diodes, etc. A detailed description of the application, as well as a thorough discussion of its parallel execution is presented in [17]. Here we will only compare the measurements as obtained for this application. As is shown in Figure 2, speedup obtained for the version of the program optimized by lazy parallelization is dramatically improved over the version obtained by default algorithm expansion. Moreover, the performance of the optimized program is even similar to that of a reasonable hand-code version [17], hence indicating the importance of the approach. More performance results for other, more complex applications are presented in [17].

5 Related Work

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. [2] 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 runtime overhead. For ad-hoc optimization of specific algorithms and applications fsm definitions have been applied successfully as well [3, 12].

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 [14, 19], 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 [4] for adaptively generating time-optimal FFT algorithms, and the ATLAS project [22] for deriving efficient implementations of basic linear algebra routines.

Finally, our work shares common goals with that of Baumgartner et. al. [1], 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 entirely static, however, and includes no possibility for partial optimization performed at run time.

6 Conclusions

In this paper we have presented a finite state machine based approach for global optimization of data parallel image processing applications. 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 generates legal, correct, and efficient parallel programs, given any sequential program implemented using our library-based software architecture for data parallel image processing.

The main advantage of the optimization approach is that it can be applied on the fly at run time. As a result, the primary importance of lazy parallelization over other approaches described in the literature, lies in the fact that it requires no a priori knowledge regarding the branching behavior of the application at hand. An additional advantage of lazy parallelization is that it requires very little runtime overhead. Also, in our software architecture it proved to be possible to implement the approach elegantly [15].

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 image processing applications. Essentially, the simplicity stems from the high level abstractions incorporated in the finite state machine definition. Consequently, we feel that a similar approach could be applicable in other library-based architectures as well. Apart from the many environments for parallel image processing (such as our own), this is also true for environments for linear algebra operations, which include similar patterns of communication and calculation.

References

[1] G. Baumgartner et al. A Performance Optimization Framework for Compilation of Tensor Contraction Expressions into Parallel Programs. In *Proc. 16th IPDPS*, Apr. 2002.

[2] S. Chatterjee et al. Generating Local Addresses and Communication Sets for Data Parallel Programs. *J. Par. Dist. Comp.*, 26(1):72–84, 1995.

[3] J. Constantin et al. Parallelization of the Hoshen-Kopelman Algorithm Using a Finite State Machine. *Int. J. Supercomp. Appl. and High Perf. Comp.*, 11(1):31–45, 1997.

[4] M. Frigo et al. FFTW: An Adaptive Software Architecture for the FFT. In *Proc. Int. Conf. on Acoustics, Speech, and Signal Processing*, pages 1381–1384, May 1998.

[5] J. Hopcroft et al. *Introduction to Automata Theory, Languages, and Computation (2nd Ed.)*. Addison Wesley, 2000.

[6] L. Jamieson et al. A Software Environment for Parallel Computer Vision. *IEEE Computer*, 25(2):73–75, Feb. 1992.

[7] Z. Juhasz et al. A PVM Implementation of a Portable Parallel Image Processing Library. In *EuroPVM'96*, 1996.

[8] D. Koelma et al. A Software Architecture for Application Driven High Performance Image Processing. In *Par. Dist. Methods for Image Processing*, pages 340–351, 1997.

[9] C. Lee et al. Parallel Image Processing Applications on a Network of Workstations. *Par. Comp.*, 21(1):137–160, 1995.

[10] P. Maurer. Logic Simulation Using Networks of State Machines. In *Proc. DATE 2000*, pages 674–678, Mar. 2000.

[11] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. Technical report, June 1995.

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

[13] C. Nicolescu et al. A Data and Task Parallel Image Processing Environment. *Par. Comp.*, 28(7-8):945–965, 2002.

[14] M. Püschel et al. Fast Automatic Generation of DSP Algorithms. In *Proc. ICCS'01*, pages 97–106, 2001.

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

[16] F. 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.

[17] F. Seinstra and D. Koelma. User Transparency: A Fully Sequential Programming Model for Efficient Data Parallel Image Processing. Technical Report Series, Vol. 2002-06, Intelligent Sensory Information Systems, Faculty of Science, University of Amsterdam, The Netherlands, Oct. 2002.

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

[19] B. Singer et al. Learning to Generate Fast Signal Processing Implementations. In *Proc. ICML'01*, pages 529–536, 2001.

[20] J. Squyres et al. A Toolkit for Parallel Image Processing. In *Par. Dist. Methods for Image Processing II*, July 1998.

[21] R. Taniguchi et al. Software Platform for Parallel Image Processing and Computer Vision. In *Par. Dist. Methods for Image Processing, Proc. SPIE*, pages 2–10, 1997.

[22] R. Whaley et al. Automated Empirical Optimization of Software and the ATLAS Project. *Par.Comp.*, 27(1):3–25, 2001.