Chapter 6

A Finite State Machine for Global Optimization of Application Performance*

"Speed is good only when wisdom leads the way."

James Poe (1921-1980)

In the previous chapters we have shown how to implement parallel versions of many common image processing operations in a sustainable manner. Also, we have shown how to accurately model the run time performance of such operations. To obtain high performance for complete applications, however, it is not sufficient to consider parallelization and optimization of the operations in isolation. This is because parallel code consisting of a sequence of optimized parallel routines often contains many redundant communication steps. Also, in many situations it is possible to further reduce communication overhead by combining multiple messages in a single transfer.

Automatic optimization of communication overhead is not easy. First, this is because the applied optimization strategy must be able to determine which communication steps are essential, and which can be safely combined or removed. In addition, the approach must guarantee that the resulting parallel code is:

- **efficient**, preferably comparable to an optimal hand-coded implementation,
- **legal**, in the sense that the program is deterministic (i.e., always produces the same result) and can never end in deadlock, and
- **correct**, such that it produces output identical to that of the original program.

*This chapter is an extended version of our paper published in the 17th International Parallel & Distributed Processing Symposium (IPDPS 2003) [145].
In this chapter we propose a new, and surprisingly simple strategy for global performance optimization that adheres to the stated list of requirements. In the approach, a *fully sequential program* is parallelized automatically by inserting communication operations whenever necessary. The approach, which is 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 essential ingredient is a set of state transition functions, each of which defines how a valid internal data structure representation is transformed into another valid representation.

Although it is shown that lazy parallelization works well in many situations, the approach does not guarantee to always produce the *fastest possible* version of a program. First, this is because the approach 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 additional communication at a later stage. Also, the approach does not incorporate knowledge obtained from our APIPM-based performance models (see Chapter 4).

To overcome these problems, this chapter also proposes an extended technique, which requires an *application state transition graph (ASTG)* to be generated for the program under consideration. An ASTG incorporates all optimization decisions that can possibly be made at run time. Each decision is annotated with a cost estimation, such that the fastest program is represented by the 'cheapest' branch in the graph. A drawback of this approach, however, is that it is often costly to obtain the cheapest branch. This is because the ASTG is generally large, even for applications of moderate size. Therefore we also define additional heuristics for search space reduction.

Hence, the primary research issue addressed in this chapter is formulated as follows: How to automatically convert a legal sequential image processing application into a legal, correct, and efficient (preferably even time-optimal) parallel version of the same program? As this issue is the central, most essential problem our software architecture for user transparent parallel image processing is confronted with, the proposed solution incorporates all results obtained in Chapters 3, 4, and 5.

This chapter is organized as follows. Section 6.1 describes the optimization problem. Section 6.2 introduces the finite state machine (fsm) definition. The fsm-based optimization strategy of lazy parallelization is described in Section 6.3. Section 6.4 presents a short description of the ASTG, and some heuristics for search space reduction. In Section 6.5 related work is discussed. Conclusions are given in Section 6.6.

### 6.1 The Performance Optimization Problem

In Chapter 3 we have defined a default parallelization strategy for each library routine. For operations executed *in isolation*, this default strategy is optimal. This is because communication overhead is minimized, while — for the given parallelization granularity — the available parallelism is fully exploited. When several optimized parallel routines are executed in sequence, however, communication overhead is generally far from optimal. This section explains the problem in more detail.
6.1.1 Abstract Function Specifications

As described in Section 4.3, each application implemented using our software architecture is composed of a sequence of instructions from the APIPM instruction set. 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. For such 'unbreakable' APIPM instruction sequences relating to sequential processing, we have introduced a shorthand notation, presented in Table 6.1.

Notation for unbreakable instruction streams relating to interprocess communication is given in Table 6.2. It contains abstractions similar to operations in MPI [104]. The additional CreatLocalPart/Full and DelLocal functions constitute creators and destructors for partial data structures (see Section 3.3.2). The BorderExchange function is as described in Section 5.3. Finally, the Redistribute function is included for completeness only, and implements a remapping of a distributed data structure onto a newly defined logical processor grid.

Partial structures are referred to as local in Table 6.2 (locsrc and locdst). The original data structure from which the partial data structures are obtained is referred to as global (globsrc and globdst). As an example, the Scatter operation requires a global source data structure as input, and produces the local (partial) destination structures as output, each of which is transferred to the node with the appropriate responsibility (see also Section 3.2).

For any application implemented using our software architecture it is possible to derive an abstract operation stream comprising of functions from Tables 6.1 and 6.2 alone. Consequently, in the remainder of this chapter we restrict our attention to abstract operation streams, and ignore the lower level APIPM instructions altogether.

<table>
<thead>
<tr>
<th>Function</th>
<th>Arguments (Input, Output)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>( OUT dst )</td>
</tr>
<tr>
<td>Delete</td>
<td>( OUT dst )</td>
</tr>
<tr>
<td>MemCopy</td>
<td>( IN src, OUT dst )</td>
</tr>
<tr>
<td>UnPixOp</td>
<td>( IN src, OUT dst, IN arg )</td>
</tr>
<tr>
<td>BinPixOpV</td>
<td>( IN src, OUT dst, IN arg )</td>
</tr>
<tr>
<td>BinPixOpI</td>
<td>( IN src, OUT dst, IN arg )</td>
</tr>
<tr>
<td>ReduceOp</td>
<td>( IN src, OUT dst )</td>
</tr>
<tr>
<td>NeighOp</td>
<td>( IN src, OUT dst, IN ker )</td>
</tr>
<tr>
<td>GenConvOp</td>
<td>( IN src, OUT dst, IN ker )</td>
</tr>
<tr>
<td>GeoMat</td>
<td>( IN src, OUT dst )</td>
</tr>
<tr>
<td>GeoRoi</td>
<td>( IN src, OUT dst )</td>
</tr>
<tr>
<td>Import</td>
<td>( OUT dst )</td>
</tr>
<tr>
<td>Export</td>
<td>( IN src )</td>
</tr>
</tbody>
</table>

Table 6.1: Abstract function specifications for sequential operation (see Tables 4A.1 and 4A.2 for comparison).
6.1.2 Default Algorithm Expansion

In Section 3.4.2 we have indicated that all data structures applied in our library operations have a predefined data access pattern type. Each such type determines how accesses to non-local partial data structures are resolved with minimal communication overhead. From this information, a default approach for parallel execution directly follows for each library operation. The availability of a default parallelization strategy for each individual operation makes for a straightforward conversion of a complete sequential image processing application into an equivalent parallel program.

The conversion process, referred to as default algorithm expansion, is illustrated by the simple example code of Listing 6.1. The abstract sequential program, shown

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Parallel (default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Import( ImA );</td>
<td>Import( ImA );</td>
</tr>
<tr>
<td>UnPixOp( ImA, ImB );</td>
<td>Scatter( ImA, locImA );</td>
</tr>
<tr>
<td>BinPixOp( ImB, ImC, ImA );</td>
<td>UnPixOp( locImA, locImB );</td>
</tr>
<tr>
<td>Export( ImC );</td>
<td>Gather( locImB, ImB );</td>
</tr>
<tr>
<td>Delete( ImA );</td>
<td>DelLocal( locImA );</td>
</tr>
<tr>
<td>Delete( ImB );</td>
<td>DelLocal( locImB );</td>
</tr>
<tr>
<td>Delete( ImC );</td>
<td>DelLocal( locImC );</td>
</tr>
<tr>
<td>Import( ImC );</td>
<td>Export( ImC );</td>
</tr>
<tr>
<td>Delete( ImA );</td>
<td>Delete( ImA );</td>
</tr>
<tr>
<td>Delete( ImB );</td>
<td>Delete( ImB );</td>
</tr>
<tr>
<td>Delete( ImC );</td>
<td>Delete( ImC );</td>
</tr>
</tbody>
</table>

(a) Sequential.                                                                 (b) Parallel (default).

Listing 6.1: Abstract sequential application (a) and equivalent parallel program after default algorithm expansion (b): additional operations in parallel code are indented.

Table 6.2: Additional abstract function specifications for parallel operation.
on the left, first imports image `ImA`, which is used as input to a unary pixel operation. Subsequently, the resulting output image `ImB` is used as input to a binary pixel operation. Finally, the resulting image `ImC` is exported, and all images are destroyed.

The equivalent parallel program, obtained after default algorithm expansion, is shown on the right of Listing 6.1. Because any data structure passed as input to a unary pixel operation is defined to have a one-to-one data access pattern type, 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 data structures are destroyed. Subsequently, the images that are passed as source and argument to the binary pixel operation are spread throughout the parallel system in a `Scatter` operation. 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 in this manner is guaranteed to produce a legal and correct parallel version of any legal sequential program implemented using our software architecture. This is simply because each abstract function call in the sequential code is replaced by an equivalent sequence of one or more (parallel) operations. The resulting program 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 parallel code often can be expected to be slower than the original sequential program. Although other parallelization tools may be implemented differently, all library-based tools suffer from the very same problem — and for improved performance a solution is essential.

### 6.1.3 Inefficiencies from Default Algorithm Expansion

When considering the parallel code of Listing 6.1(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 `locImA`, which is used as source structure for the unary pixel operation, is removed by `DelLocal` and subsequently recreated in the second occurrence of the `Scatter(ImA, locImA)` call. For improved performance, both operations simply could be removed. The same holds for the sequence of instructions applied to the `locImB` structure preceding the `BinPixOpI` call (i.e., `Gather` followed by `DelLocal` and `Scatter`). Listing 6.2(b) presents the optimized program obtained after removing the redundant communication steps from the parallel code.

A second source of inefficiencies is due to the fact that each individual communication step is performed irrespective of other message transfers in the program. Consequently, removal of redundant communication operations is a form of *local* performance optimization only, as it may be better to combine multiple messages in a single transfer. As an example, a `Scatter` operation followed by a `Broadcast` performed on the same data structure at a later stage in a program, could be replaced by a single `Broadcast` at the first essential point of message transfer, possibly followed by a `MemCopy` operation to extract the partial data structures on each processing unit.

A third category of performance inefficiencies is due to the fact that default algorithm expansion ignores the performance characteristics of the parallel machine at
Listing 6.2: Abstract sequential application (a) and equivalent parallel program after inter-operation optimization (b).

hand. As indicated in Chapters 4 and 5, communication overhead also depends on the specifications of the underlying interconnection network, and the implementation of the applied message passing primitives. As a consequence, it is essential for the APIPM-based performance models of Chapters 4 and 5 to be incorporated in the optimization process as well.

From these types of inefficiencies, the first (i.e., the presence of redundancy) is by far the most important to be resolved. This is because redundant operations are responsible for the bulk of all unnecessary communication overhead. In fact, a program which is stripped of all redundant communication is generally quite efficient, and is often comparable to hand-optimized code. Redundancy avoidance is therefore the focal point of the optimization strategy proposed in the next sections. The latter two types of inefficiencies are still important, however, as these may have a significant impact on execution time as well. This is especially true for large clusters, as the relative impact of communication on performance increases with every node added to the system. Consequently, the remainder also proposes an extended optimization strategy that takes into account the latter two types of inefficiencies.

6.2 Finite State Machine Definition

To guide the process of operation removal, we have defined a finite state machine (fsm) which is used for operation redundancy detection, the monitoring of the life-span of (distributed) data structures, and the resolution of data structure inconsistencies. In this chapter, we restrict ourselves to so-called deterministic finite acceptors, which have no temporary storage and which can not produce strings of output. A deterministic finite acceptor (or dfa) is defined by the quintuple

\[ M = (Q, \Sigma, \delta, q_0, F), \]

where
6.2. Finite State Machine Definition

$Q$ is a finite set of internal states,
$\Sigma$ 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.

Initially, a dfa is assumed to be in the initial state $q_0$, with its input mechanism on the leftmost symbol of the input string. During each move of the automaton, one input symbol is consumed. When the end of the string is reached, the string is accepted if the automaton is in one of the final states. Otherwise, the string is rejected.

Deterministic finite acceptors as described here have been applied successfully in many fields of computer science, e.g. digital design, programming languages, and compilers [70, 96]. The following presents a specification of the finite state machine for global application optimization as applied in our software architecture.

6.2.1 States and Lifespan of (Distributed) Data Structures

As described in Section 3.3.2, 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 collective communication operation performed on a global structure.

There is a strong relationship between a global structure and the set of derived local structures (a set which is referred to as a distributed data structure). Clearly, at any time during the 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 data 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

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

In set $G$, none indicates that no space has been allocated for the global data 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
Chapter 6. A Finite State Machine for Global Optimization of Application Performance

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 for the distributed structure. In addition, partial indicates that the set of constituent local structures is the result of a non-overlapping Scatter operation, while full indicates that the structures are obtained in a Broadcast operation. Finally, not-reduced indicates that all elements of the constituent, fully overlapping, local structures yet have to be subjected to an element-wise ReduceOne or ReduceAll 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, at application run time many of these possible representations can not (or should not) occur. As an example, a representation given by $q = (\text{invalid}, \text{invalid}, \text{full})$ should not be present in a program, as neither the global structure nor the distributed structure contains all correct and up-to-date values. In addition, the representation given by $q = (\text{none}, \text{none}, \text{full})$ is not useful, as it contains as much information as the more accurate representation $q = (\text{none}, \text{none}, \text{none})$.

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

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

with

- $q_0 = (\text{none}, \text{none}, \text{none})$
- $q_1 = (\text{created}, \text{none}, \text{none})$
- $q_2 = (\text{valid}, \text{none}, \text{none})$
- $q_3 = (\text{invalid}, \text{none}, \text{none})$
- $q_4 = (\text{valid}, \text{valid}, \text{partial})$
- $q_5 = (\text{valid}, \text{valid}, \text{full})$
- $q_6 = (\text{invalid}, \text{valid}, \text{partial})$
- $q_7 = (\text{invalid}, \text{valid}, \text{full})$
- $q_8 = (\text{invalid}, \text{invalid}, \text{not-reduced})$

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 required 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 DelLocal 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, certain states can 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.

### 6.2.2 State Transition Functions and State Dependencies

The input alphabet for our finite state machine is formed by the abstract functions of Tables 6.1 and 6.2, 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 given 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 is extended as follows:

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

where $\Sigma_d$ 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 $\Sigma_d$ 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 [101]). For simplicity of notation, however, in the remainder of this chapter we keep to the parallel view of simple state machines.

Table 6.3 presents the state transition functions for the image processing functionality available in our software 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 6.1). 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 fully sequential execution using global data structures only. All other transition functions refer to parallel execution using distributed data structures.

\[
\begin{align*}
\delta(q_0, (\text{Create}, -)) &= q_1, & \delta(q_i, (\text{Delete}, -)) &= q_0, \\
\delta(q_0, (\text{Import}, -)) &= q_2, & \delta(q_j, (\text{Export}, -)) &= q_j,
\end{align*}
\]

with $i \in \{1, 2, 3\}, j \in \{1, 2, 4, 5\}$,

\[
\begin{align*}
\delta(q_0, (\text{op}, q_2)) &= q_2, & \delta(q_0, (\text{op}, q_6)) &= q_6, \\
\delta(q_0, (\text{op}, q_4)) &= q_6, & \delta(q_0, (\text{op}, q_7)) &= q_7, \\
\delta(q_0, (\text{op}, q_5)) &= q_7, & \delta(q_i, (\text{op}, q_0)) &= q_i,
\end{align*}
\]

with $\text{op} \in \{\text{Memcopy, UnPixOp}\}, i \in \{2, 4, 5, 6, 7\}$,

\[
\begin{align*}
\delta(q_0, (\text{BinPixOpV}, q_2, q_2)) &= q_2, & \delta(q_2, (\text{BinPixOpV}, q_0, q_2)) &= q_2, \\
\delta(q_0, (\text{BinPixOpV}, q_i, q_j)) &= q_i, & \delta(q_i, (\text{BinPixOpV}, q_0, q_j)) &= q_i, \\
\delta(q_0, (\text{BinPixOpV}, q_k, q_l)) &= q_k, & \delta(q_k, (\text{BinPixOpV}, q_0, q_l)) &= q_k,
\end{align*}
\]

with $i, j \in \{4, 6\}, k, l \in \{5, 7\}$,

\[
\begin{align*}
\delta(q_0, (\text{ReduceOp}, q_2)) &= q_2, & \delta(q_2, (\text{ReduceOp}, q_0)) &= q_2, \\
\delta(q_0, (\text{ReduceOp}, q_i)) &= q_i, & \delta(q_i, (\text{ReduceOp}, q_0)) &= q_i, \\
\delta(q_0, (\text{ReduceOp}, q_j)) &= q_j, & \delta(q_j, (\text{ReduceOp}, q_0)) &= q_j,
\end{align*}
\]

with $i \in \{4, 6\}, j \in \{5, 7\}$,

\[
\begin{align*}
\delta(q_0, (\text{op}, q_2)) &= q_2, & \delta(q_2, (\text{op}, q_0)) &= q_2, \\
\delta(q_0, (\text{op}, q_i)) &= q_i, & \delta(q_i, (\text{op}, q_0)) &= q_i,
\end{align*}
\]

with $\text{op} \in \{\text{GeoMat, GeoRoi}\}, i \in \{5, 7\}$.

Table 6.3: State transition functions (including annotated state dependencies) for image processing functionality.
6.2. Finite State Machine Definition

\[
\begin{align*}
\delta(q_1, (\text{CreatLocalPart}, -)) &= q_4, & \delta(q_i, (\text{DelLocal}, -)) &= q_2, \\
\delta(q_1, (\text{CreatLocalFull}, -)) &= q_5, & \delta(q_j, (\text{DelLocal}, -)) &= q_3, \\
\end{align*}
\]

with \(i \in \{4, 5\}, j \in \{6, 7, 8\},

\[
\begin{align*}
\delta(q_2, (\text{Broadcast}, -)) &= q_5, & \delta(q_8, (\text{ReduceOne}, -)) &= q_2, \\
\delta(q_2, (\text{Scatter}, -)) &= q_4, & \delta(q_8, (\text{ReduceAll}, -)) &= q_5, \\
\delta(q_6, (\text{Gather}, -)) &= q_4, & \delta(q_i, (\text{BorderExchange}, -)) &= q_i, \\
\delta(q_7, (\text{Gather}, -)) &= q_5, & \delta(q_i, (\text{Redistribute}, -)) &= q_i, \\
\delta(q_6, (\text{GatherAll}, -)) &= q_5. & \\
\end{align*}
\]

with \(i \in \{4, 6\}.

Table 6.4: Additional state transition functions for parallel execution.

State transition functions related to the additional communication functionality, and the memory management of local data structures, are presented in Table 6.4. In all of these transition functions the list of state dependencies is empty, as the functions work on a single data structure only. The importance of the additional transition functions is that these are used to resolve data structure state inconsistencies which may appear in an application. As an example, consider the first three lines of code in Listing 6.1(b). The first operation (Import) moves structure \(\text{ImA}\) from \(q_0\) to \(q_2\) (see Table 6.3). In case the third operation (UnPixOp) is to be executed in parallel, the input data structure is expected to be in one of the states \(q_4, q_5, q_6,\) or \(q_7\). None of these states immediately matches with the output state of structure \(\text{ImA}\) after the Import operation. This state inconsistency is resolved by executing a Scatter operation (as in Listing 6.1(b)) or a Broadcast operation immediately after the Import operation. This is because these operations change an input structure's state from \(q_2\) either to \(q_4\), or to \(q_5\) respectively (see Table 6.4).

Figure 6.1 presents a reduced state transition graph for our finite state machine definition. For better readability, the graph contains only those operations that actually cause a data structure to move from one state to another state. As such, the graph incorporates the complete lifespan of a data structure, and covers any state a data structure can possibly reach at run time. Also, it should be noted that it is exactly these operations that play an essential role in the process of operation redundancy avoidance as will be presented in Section 6.3.

6.2.3 Legal Sequential Code and Legal Parallel Code

A program is a legal program, if and only 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 6.1 and 6.2 only, (2) it contains no data structure state inconsistencies, and (3) all internal 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 as well. 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 definition is used to obtain legal and correct parallel code, which is optimized in that the execution of any redundant communication operations is avoided.

*1, *2, *3, *4 = creation of datastructure by one of several image operations

Figure 6.1: Reduced state transition graph.
6.3 Redundancy Avoidance by Lazy Parallelization

In the approach of lazy parallelization it is simply assumed that each communication or memory management operation inserted in the default algorithm expansion process 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 compile time method. Conceptually, the approach of 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 6.3 gives an example of the application of lazy parallelization. The abstract code for a simple example program is shown in Listing 6.3(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 6.3(e)) is performed using the state transition functions of Section 6.2.2 (i.e., only those incorporated in the reduced state transition graph of Figure 6.1). The Broadcast (ImA, locImA) 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
Listing 6.3: Example of code optimization by lazy parallelization (compile time): (a) original sequential code, (b) code obtained after default algorithm expansion, (c) code obtained after removal of ‘redundant’ communication operations and memory management operations, (d) code obtained after partial loop unrolling, (e) code obtained after resolution of state inconsistencies by default operation re-insertion, (f) optimized parallel code obtained after loop recombination.
be in state $q_5$ or $q_7$ (see Table 6.3). The only available operation that provides a resolution to this state inconsistency is the \textit{Broadcast} operation, as it moves a data structure from state $q_2$ to $q_9$. Similarly, $\text{Gather}(\text{locC}, C)$ 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 \textit{Export} operation. The additional operation re-insertions work in a similar manner, and all further interpretation of Listing 6.3 is left to the reader.

6.3.1 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 6.1, 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 transition functions 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 or memory management 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 unrestricted and highly efficient, as no prior knowledge regarding the behavior of loops and branches in the code is required. This knowledge is simply obtained during execution of the application, and is not required any sooner.

Although lazy parallelization works well in many situations, it does not guarantee to always produce the fastest possible version of a program under consideration. First, this is because the approach 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 to be executed at a later stage. Secondly, the approach does not incorporate any knowledge obtained from our APIPM-based performance models described in Chapters 4 and 5. To overcome these problems, the next section proposes an extension to the approach of lazy parallelization, such that it is indeed capable of producing the (expected) fastest parallel version of any sequential program.

6.4 Application State Transition Graph

The process of lazy parallelization always results in the execution of a single pre-selected solution for resolution of data structure state inconsistencies. For each specific state inconsistency, each default resolution represents the cheapest operation (or sequence of operations) that is available in the software library. Although this strategy generally produces parallel code which is quite efficient, the approach is sub-optimal,
as it does not acknowledge that

1. the execution of more costly communication steps (e.g., \texttt{Broadcast} instead of \texttt{Scatter}) may avoid additional communication at a later stage in the program,

2. a single straightforward domain decomposition may deliver non-optimal performance (see Chapter 5),

3. the optimal routing pattern for the distribution of data partially depends on the characteristics of the interconnection network (again, see Chapter 5), and

4. the use of all available processing power is not always time-optimal.

Optimization in the light of these issues is obtained by constructing an \textit{application state transition graph} (or ASTG), that characterizes an application’s run time behavior, and incorporates all possible (combinations of) parallelization and optimization solutions. By annotating the vertices in the graph (representing all operations which are possibly performed by the application) with cost estimations obtained from our APIPM-based performance models described in Chapters 4 and 5, the expected optimal parallel implementation for an application is represented by the cheapest branch.

Figure 6.2 shows a simplified version of the ASTG constructed for the first three lines of code in Listing 6.1, assuming that a maximum number of only two processing units is available. After execution of the \texttt{Import} operation, several different execution paths can be followed. One choice is to execute the \texttt{UnPixOp} in a sequential manner, as is depicted by the uppermost branch in the graph. Parallel solutions involve either a \texttt{Scatter} operation or a \texttt{Broadcast} operation performed on the imported data structure. As explained in the Chapter 5, multiple versions of these operations exist in our

![Figure 6.2: Simplified partial application state transition graph.](image-url)
library, each having different performance characteristics. For the Scatter operation, it is required to also choose a logical processor grid onto which the data structure is to be mapped (see Section 3.2). All of these choices result in a different expected execution time for the program, as is indicated by the annotated performance estimations at each vertex in the graph. Although one of the branches in Figure 6.2 is cheapest for this initial part of the program, to obtain optimal performance for the complete application a different path may have to be followed.

Discussion

While the expected optimal parallel implementation is always obtained in this manner, the construction of a complete ASTG has several major disadvantages. First, in order to find the cheapest branch, the creation of an ASTG needs to be performed at compile time. As such, the approach is restrictive, as it is now required to have prior knowledge regarding the branching behavior of the application at hand. Another drawback is that it is often costly to actually obtain the cheapest branch in the graph. This is because an ASTG is generally large, even for applications of moderate size.

6.4.1 Heuristics for Search Space Reduction

To overcome the stated problems, we have defined several heuristics to reduce the size of any application state transition graph. The use of heuristics implies that our approach can no longer guarantee to find the expected optimal parallel implementation for any sequential program. However, in almost all situations a close-to-optimal program is still obtained, and application performance is generally still comparable to that of optimal hand-crafted parallel code (as will be demonstrated for all example applications evaluated in Chapter 7).

First, to overcome the problem of having to acquire prior knowledge regarding the branching behavior of an application, our optimization approach simply ignores unknown branches. At run time, any code block that has not been evaluated because of undetermined conditional behavior is simply executed according to the default lazy parallelization approach. In such a situation, all current logical data structure mappings are maintained, however, to avoid having to execute costly remapping operations. Although this approach solves the problem in the simplest possible way, it should be noted that we have learned that not many applications implemented using our software architecture actually contain such unknown branches.

A significant reduction of any ASTG is obtained by assuming that a specific data mapping that was found to be optimal for a certain operation, is also optimal for other operations with similar behavior. In other words, it is simply assumed that each parallelizable pattern entails a single optimal data partitioning strategy, irrespective of the actual operation that is implemented by that pattern. As a consequence, in an ASTG a sequence of operations applied to the same set of data structures is often reduced to a single block of code which is not 'interrupted' by any communication operations. Once a data structure has been partitioned and distributed, its logical mapping is maintained as long as possible.
An ASTG is also significantly reduced by assuming that data structures that are used as arguments representing kernel structures (as in the GenConvOp operation) or vector data (as in the BinPixOpV operation) at any point in a program, are never to be partitioned. This is realistic, as such data structures are usually much smaller than regular image data structures. Calculations on such small structures are simply assumed not to gain from parallel execution at all.

Other heuristics, such as evaluating partial execution paths either for a single node or for all available nodes only, and considering only a small number of possible logical data mappings for the maximum system size, also reduce each ASTG significantly. It is expected that additional heuristics can reduce each ASTG even further, without compromising too much on the run time performance of the resulting parallel code. This, however, is research we have left as future work.

### 6.5 Related Work

Although a multitude of library-based environments has been described in the literature, the process of optimization across library calls is not explicitly incorporated in many of these. Even in several relatively recent software architectures, performance optimization issues often are considered at the intra-operation level only (e.g., see [80, 81, 86, 87, 93, 111, 154, 159]). Other environments (e.g., [118]) leave part of the optimization process to a third-party compiler, as these require applications to be implemented in a high-level parallel language such as Compositional C++ [26].

The environment implemented by Morrow et al. [109] does incorporate a partial mechanism for inter-operation optimization. It is based on the concept of a *self-optimizing class library*, which is extended automatically with optimized parallel operations. In case a program is being executed for the first time, a syntax graph is constructed for each statement in the program, which is evaluated when an assignment operator is met. Any such syntax graph for combinations of primitive instructions (i.e., those incorporated as a single routine within the library) 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. Although optimal performance may be guaranteed for a sequence of library routines in this manner, a drawback of this approach is that time-optimality is often not obtained for complete applications.

Other environments, such as developed by Jamieson et al. [73, 74], Lee et al. [94, 95], and Moore et al. [108], do incorporate a method for full inter-operation optimization. In all of these architectures the methods are purely static, however, and can be applied at compile time only. In this respect, our approach of lazy parallelization is more flexible, as it allows much of the optimization process to be performed at run time — without any significant overhead cost. In case run time performance obtained from the standard lazy parallelization approach is deemed insufficient, one can decide to incorporate additional compile time results obtained from the ASTG.

As far as we know, the use of a finite state machine specification is new in the field of library-based parallel imaging environments. Moreover, to our knowledge the
application of an fsm definition has not been considered at all in the field of parallel image processing. In several related research areas, however, fsm definitions have been applied before. For example, Chatterjee et al. [27] apply a finite state machine for the generation of optimal communication sets in distributed-memory implementations of data-parallel languages such as High Performance Fortran. As in our case, results indicate that the fsm approach requires very little runtime overhead. For ad-hoc optimization of specific algorithms (e.g., see [31]), or complete applications (e.g., see [106]), finite state machine definitions have been applied successfully as well.

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 [99, 152], 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 [51] for adaptively generating time-optimal FFT algorithms, and the ATLAS project [169] for deriving efficient implementations of basic linear algebra routines.

Finally, our work shares common goals with that of Baumgartner et. al. [14], in the search of an optimal data partitioning strategy with minimal communication overhead for applications in the field of quantum chemistry and physics. Similar to our work, an operator tree is generated, in which multiple data partitioning and communication strategies are incorporated. This work goes even one step further, in that memory usage is to be optimized at the same time. This approach is also entirely static, however, and includes no possibility for partial optimization performed at run time.

### 6.6 Conclusions

In this chapter we have presented a finite state machine based approach for global optimization of data parallel image processing applications. The approach, called 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 software architecture.

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 incorporate the approach in an elegant manner — i.e., such that the long-term sustainability of the library implementations is not compromised.

Although lazy parallelization was shown to work well in many situations, it cannot guarantee to always produce the fastest possible version of the program under consideration. To overcome this problem, an extension to the approach of lazy par-
allelization was also presented. The extended technique requires an application state transition graph (ASTG) to be generated. An ASTG incorporates all optimization decisions which can possibly be made at application run time. As each decision is annotated with a run time cost estimation (obtained from our APIPM-based performance models), the fastest version of the program is represented by the 'cheapest' branch in the ASTG.

An important drawback of the application state transition graph, however, is that it is often costly to actually obtain the cheapest branch. This is because the ASTG is generally large, even for applications of moderate size. For this reason we have also defined additional heuristics for search space reduction. Another drawback is that the creation and traversal of an ASTG can not be performed at run time. However, in case the default approach of lazy parallelization proves to deliver sufficiently high performance, the creation of an ASTG can be avoided altogether.

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 fsm definition. Consequently, we feel that a similar approach could be applicable in other library-based architectures as well. This is especially true for the many environments for linear algebra operations, which include similar patterns of communication and calculation.