Chapter 4

Semi-Empirical Modeling of Parallel Low Level Image Processing Operations*

"Pour avoir une vérité il faut deux facteurs — un fait et une abstraction."

Remy de Gourmont (1858 - 1915)

As described in Chapter 3, for each sequential algorithmic pattern available in our library we have implemented only one parallel counterpart. Because no single parallel implementation is guaranteed to provide optimal performance on all target platforms, each operation is implemented such that it is capable of adapting to the specific performance characteristics of a parallel machine at hand. In the previous chapter we have indicated that two of the parameters that determine an application’s parallel performance are fixed in the library implementations: i.e., the parallelization granularity (or, the amount of computation performed between two communication steps) as well as the data dependencies. Optimization decisions relating to several additional parameters, however, are still made at application compile time, and even at run time. Such parameters include (1) the logical processor grid to map data structures onto, (2) the routing pattern for the distribution of data, (3) the number of processing units, and (4) the type of data distribution (e.g., broadcast instead of scatter).

To make optimization decisions automatically, knowledge is required of the performance characteristics of the routines applied in a particular application. In our software architecture we have incorporated this knowledge by annotating each user-callable library operation with a performance model for run time cost estimation. Due

*This chapter combines our papers published in Proceedings of the 7th International Euro-Par Conference (Euro-Par 2001) [147] and Parallel Computing [149].
to the intended portability of the software architecture to clusters. The performance models have been designed to be applicable to all machines in this class of architectures. Also, the complexity of the models is kept to a minimum to allow high-speed evaluation of complete applications possibly even at run time. In addition, the models are capable of generating estimations of sufficiently high accuracy to allow optimization decisions to be made correctly.

In the literature a multitude of performance analysis and modeling techniques has been described. The techniques range from direct measurement to detailed mathematical and simulation models, and adhere to widely varying performance requirements in terms of both estimation accuracy and speed of evaluation. As will be discussed in this chapter, a major problem with existing performance estimation techniques is that these generally incorporate a direct relationship between the estimation accuracy and the technique's complexity (for example, the number of model parameters). In other words, increased estimation accuracy is obtained at the expense of greater complexity and reduced evaluation efficiency.

In this chapter we propose a semi-empirical modeling technique that is specifically designed to overcome this problem. While being simple and portable, the semi-empirical modeling approach also provides a sufficiently high estimation accuracy. The approach is based on a high-level abstract machine definition (the Abstract Parallel Image Processing Machine, or APIPM) which is designed to capture typical behavior of low level image processing operations executing on a cluster. From the APIPM instruction set a high-level abstract performance model is obtained that is applicable to all such platforms. The crux of the semi-empirical modeling approach is that an additional benchmarking phase is required to capture implicit but essential cost factors, and to bind each abstract model parameter to a concrete performance estimation for a parallel machine at hand.

Hence, the primary research issue addressed in this chapter is as follows: How to apply benchmarking in combination with simple analytical models to obtain accurate performance estimates for optimization of complete parallel image processing applications? In this respect, it is interesting to note that this research issue closely relates to the more general problem statement put forward by Professor Tony Hey in his invited talk at the Euro-Par 2001 conference: “The ultimate goal in the field of parallel and distributed computing is to use a combination of benchmarking kernels and simple models for accurate performance estimation of full applications” [68]. Essentially, our APIPM-based semi-empirical modeling approach forms a domain-specific solution to this much broader — and as of yet: unsolved — problem.

This chapter is organized as follows. Section 4.1 investigates the requirements for a performance estimation technique to be applied in our software architecture. Several existing approaches are evaluated according to these requirements as well. On the basis of two estimation techniques described in the literature, a generalized description of our semi-empirical modeling approach is given in Section 4.2. Section 4.3 introduces the APIPM and its instruction set. The APIPM-based performance models, and the applied benchmarking technique, are presented in Section 4.4. In Section 4.5 model predictions are compared with results obtained on a real machine from the class of platforms under consideration. Concluding remarks are given in Section 4.6.
4.1 Computer System Performance Estimation

The success of our software architecture greatly depends on the quality of the code optimization, which is to be performed automatically, hidden from the user. Code optimization is implemented by leaving each operation in the library a choice between several parallelization strategies. Each such strategy has a different effect on the performance on each parallel platform. By providing accurate estimations of the performance of each strategy for the parallel machine at hand, the fastest code alternative is selected with ease.

The effectiveness of the optimization process entirely depends on the technique for estimating the performance of a computer system. In the following we present the general requirements for a performance estimation technique to be applied in our software architecture. In the light of these requirements a short evaluation is given of the most significant estimation approaches described in the literature.

4.1.1 Estimation Technique: Requirements

A performance estimation technique designed for our purposes should incorporate all relevant tasks typically performed by data parallel imaging operations. In our case such tasks relate to either computation, communication, or I/O. Computational tasks include all parallelizable patterns as defined in Chapter 3, as well as the basic memory operations for creation, destruction, and copying of data structures. Communication tasks are formed by the bulk of operations from the set of parallel extensions described in the previous chapter, including overlap communication and all distribution and redistribution routines. I/O tasks include all operations for transporting data between a processor's main memory and external devices such as disk drives and cameras.

Apart from having to reflect the typical behavior of parallel low level image processing routines, the performance estimation technique should also conform to the following (more general) requirements (similar to [62]):

1. **Simplicity.** In a realistic estimate, the number of samples is proportional to the number of parameters. To reduce the costs of performance evaluation, the number of free parameters should be kept to a minimum.

2. **Accuracy.** To make sure the architecture can make correct optimizations, the generated performance estimations must be of sufficiently high accuracy. The degree of accuracy is considered sufficient if correct decisions are made in at least 95 percent of all cases, and poor decisions are generally avoided.

3. **Applicability.** For portability, the performance evaluation technique integrated in our software architecture must be applicable to all clusters.

It is important to note that — in general — the requirement of simplicity enhances applicability, but reduces accuracy. Therefore, care must be taken in the design of the estimation technique to ensure that it can produce good performance estimates with relative ease.
4.1.2 Estimation Techniques in the Literature

Techniques for computer system performance estimation abound in the literature. Roughly speaking, each such technique can be classified into one of three main categories: (1) measurement, (2) modeling and (3) hybrid methods. Estimation techniques that belong to the second category can be further divided into the subcategories of (2a) mathematical analysis and (2b) simulation [72].

Category 1: Measurement

Performance estimation by measurement is generally performed on a real system under conditions that reflect typical workload and behavior. Execution times of real problems are then inferred from measured results. Application of this approach in our software architecture has several drawbacks. First, in many cases the complete system to be evaluated has yet to be developed, and may change over time. Second, even if a complete system is available it is often not clear what workload is realistic or typical. Finally, if the measurement process is biased towards certain aspects of the underlying hardware, the measurement technique may not be applicable to other platforms.

Benchmarking is an alternative technique, which is often used for comparison of multiple computer systems (e.g., see [39, 69, 167]). Rather than reflecting typical behavior, benchmarks often represent non-typical, artificial workloads. In comparison with direct measurement, benchmarking has the advantage that the system to be evaluated does not have to be available. The use of non-typical workloads, however, often has a negative effect on the accuracy of the performance estimations. A solution — albeit complex — is to capture results for small instruction mixes and a variety of workloads, and to interpret the measurement results with utmost care [44, 156].

Category 2: Modeling

Performance modeling can be applied in cases where direct measurement is too costly, or where the computer system to be evaluated is not available. In the category of mathematical analysis, models range from simple (linear) algebraic expressions to complex formalisms such as queueing networks [72, 135]. In general, such models have a high response time due to their ease of evaluation. An additional advantage is that parameter values may be varied to observe their relative impact on performance. However, to obtain high estimation accuracy, the large number of model parameters may violate the simplicity and applicability constraints.

In simulation models behavior and workloads are described (imitated) in a special computer program — usually an annotated or otherwise adapted version of a ’real’ program [72, 123]. Performance predictions are obtained by monitoring the execution of the adapted program. The main advantage of simulation models is that dynamic system behavior is easily captured. Also, simulation makes it easy to ’zoom in’ on interesting or expensive parts of a system. A disadvantage is that the system to be evaluated must be available, at least in some rudimentary form. Another drawback is that it is a costly method for obtaining even moderately accurate performance estimates, thus violating the simplicity constraint.
Category 3: Hybrid Methods

In hybrid estimation techniques a combination of measurement and modeling is applied [108, 172]. Such techniques have the advantage that the complexity of using either measurement or modeling in isolation can be avoided, while a high level of estimation accuracy can still be obtained. The following discusses two hybrid approaches that form the basis for the estimation technique applied in our software architecture.

1. Machine Characterization Based on an Abstract Fortran Machine

In [133], Saavedra-Barrera et al. acknowledge that many state-of-the-art sequential computer systems have become too complex to be accurately captured in a mathematical model. The authors measure system performance in terms of an Abstract Fortran Machine (AFM) — an approach referred to as narrow spectrum benchmarking. The AFM instruction set consists of the primitive operations available in Fortran, such as arithmetic and logical functions, procedure calls, loops, etcetera. All primitive operations are measured separately, and the combined set of measurements characterizes a specific machine. The approach is based on the assumption that the execution time of any program can be partitioned into independent time intervals, each corresponding to one AFM instruction. Although, in general, high level operations are never completely independent (e.g., due to compiler optimizations), the authors have shown that the assumption is reasonably accurate for a wide range of systems [132]. It should be noted that an earlier technique, described by Peuto et al. [122], is similar to the AFM-based approach. It is different, however, in that all machine characterizations are incorporated at the much lower level of machine instructions.

The model of the total execution time of a program $A$ as described in [133] is formalized as follows. Let $P_M = (P_1, P_2, \ldots, P_n)$ be the set of parameters that characterizes the performance of machine $M$. Each of the $n$ performance parameters is related to a different operation in the AFM instruction set. Let $C_A = (C_1, C_2, \ldots, C_n)$ be the normalized dynamic distribution of the AFM instructions present in program $A$, and let $C_{total}$ denote the total number of AFM instructions executed in program $A$. The expected execution time of program $A$ on machine $M$ is then obtained by

$$T_{A,M} = C_{total} \sum_{i=1}^{n} C_i P_i = C_{total} C_A \cdot P_M, \quad \text{with} \quad \sum_{i=1}^{n} C_i = 1.$$

The authors indicate that the only way in which this linear model can give accurate results is when (1) the measurements of the AFM instructions are representative of typical occurrences in real programs, (2) errors caused by the intrusiveness of the measurements are not significant, and (3) variance in the mean execution time caused by the system, and by the instructions themselves, is small. Still, experiments have shown that for many applications the performance predictions were sufficiently close to actual execution times. In general, occurrences of bad estimations were easily explained by code optimizations performed by the compiler, which had not been captured in the benchmarking process.
2. Incorporating System Variance by Adaptive Sampling

The AFM-based approach of narrow spectrum benchmarking provides a solution to the problem of the high complexity of complete analytical study of computer systems. A drawback of the approach, however, is that system variance is almost completely ignored. For applications working on extensive dense data fields (e.g., image data structures) this is too crude a restriction as variations in the hit ratio of caches and system interrupts often have a significant impact on performance [59, 136].

In [90] a prediction method is presented that incorporates both program behavior and machine variance. The predictions are based on the approach of adaptive sampling, which is constrained by a fixed time budget for all measurements. In other estimation methods significant inaccuracies in performance estimates may arise, as a known execution time for one input size is often a poor predictor of the performance for other input sizes. In general, the main source of variation is due to the availability of small amounts of fast cache memory. As there is a decreasing portion of data residing in cache with increasing input size, linearity in response is disturbed.

This problem is attacked by the adaptive sampling approach, which measures the execution time of an algorithm for several input sizes. The advantage of the approach is that, in part, it also incorporates sources of variation inherent to an application. In matrix multiplication, for example, a linear increase in the sizes of the data-structures being applied results in a non-linear growth in execution time. Another nice feature of the approach is that it fixes the time needed for the measurement process. One may be tempted to run a benchmark at the largest size believed to fit within the budget. However, due to the many possible sources of variation the assumed execution time may be far from realistic.

Some image processing functions (e.g., data-driven segmentation) have an inherent randomness, and an execution time that is much less predictable. For such algorithms it is difficult to obtain accurate estimations on the basis of adaptive sampling. Another problem is that the approach may only measure small sized inputs not representative of typical workloads. As will be discussed in Section 4.4, due to these latter two problems we have chosen to apply a measurement technique similar, but not identical, to the adaptive sampling approach.

A Combination of Techniques

From the presented overview we conclude that several effective estimation techniques exist that are based on measurement, modeling, or a combination thereof. Unfortunately, no estimation technique exists that provides a level of abstraction that is truly applicable for optimization of applications implemented using our software architecture. Also, many measurement techniques have proven to be a weak basis for accurate performance estimation, as the impact of system variance is often ignored [90].

In the following section we present a description of the performance estimation approach applied in our software architecture. Essentially, it is a combination of the two hybrid techniques described above, as it integrates the impact of system variance with high level abstractions relevant for image processing applications. We refer to our approach as semi-empirical modeling.
4.2 Semi-Empirical Modeling

As stated, any accurate performance estimation technique should cover all relevant aspects of a computer system under consideration. Consequently, many existing estimation techniques incorporate detailed behavioral abstractions relating to the major components of such a system [72, 100], a.o. including: (1) the central processing unit, (2) the memory hierarchy, including multiple cache levels, (3) I/O devices, (4) the interconnection network, (5) the operating system, and sometimes also (6) a specific piece of application software. A major problem with this approach is that one may need tens, if not hundreds, of platform-specific machine abstractions to obtain truly accurate estimations. Consequently, the essential requirements of simplicity and applicability as put forward in Section 4.1.1 are not satisfied.

To overcome this problem we have designed a new technique for performance estimation of parallel image processing applications running on clusters. The technique, which we refer to as semi-empirical modeling, allows for high-speed evaluation of complete applications or any relevant constituent subtask. Also, the technique is sufficiently accurate to allow correct optimization decisions to be made automatically, on any machine in the class of target platforms. The semi-empirical modeling approach is based on three essential ingredients:

1. A high level abstract machine definition for parallel low level image processing (the APIPM), including a related instruction set.

2. A simple, APIPM-based, linear performance model related to each user-callable library operation.

3. A benchmarking method — aimed at the application domain of low level image processing — to capture essential cost factors not made explicit in the models.

In other words, the technique is based on a combination of relevant abstraction, simple modeling, and domain-specific measurement.

The essence of the semi-empirical modeling approach is that any behavior or cost factor that can not be assumed identical for all target platforms is abstracted from in the definition of the model parameters. To still bind each abstract model parameter to an accurate performance estimation for a parallel machine at hand, benchmarking is performed on a small set of sample data to capture all such essential, but implicit cost factors. In the remainder of this chapter the three essential ingredients of our modeling approach are discussed in more extensive detail.

4.3 Abstract Parallel Image Processing Machine

As in the AFM-based approach described in Section 4.1.2, we have introduced well-defined system abstractions by specifying a high level abstract machine for image processing: the Abstract Parallel Image Processing Machine, or APIPM. In the APIPM common hardware characteristics of the target machines are reflected by the definition of abstract hardware components. In addition, the typical behavior of the routines to be run are reflected in a related instruction set.
4.3.1 APIPM Components

An APIPM consists of one or more identical abstract sequential image processing machines (ASIPMs), each consisting of four closely related components (see Figure 4.1):

1. a sequential image processing unit (SIPU), capable of executing APIPM instructions, one at a time,
2. a memory unit, capable of storing (image) data structures,
3. an I/O unit, for transporting data between the memory unit and external sensing or storage devices,
4. data channels, the means by which data is transported between the ASIPM units and external devices.

Although the memory unit of each ASIPM is connected with those of all other ASIPMs, no ASIPM has direct access to data maintained by any other ASIPM. The ASIPMs are ordered and identified by a unique number. The range of valid identifiers is 0, ..., $N - 1$, where $N$ is the number of ASIPMs in the APIPM. Each ASIPM has knowledge of the range of valid identifiers, and of its own unique number.

The definition of the APIPM reflects a state-of-the-art homogeneous commodity cluster. It only differs from a general purpose machine in that each sequential unit is

![Figure 4.1: Abstract Parallel Image Processing Machine (APIPM) comprising of four ASIPMs.](image_url)
designed for image processing related tasks only. Each ASIPM is capable of running code individually, independent of all other ASIPMs. The programs executed by each ASIPM need not be identical. Exchange of data between processing units is possible by communication over the interconnecting data channels.

Although most realistic clusters do not have a fully connected communication network, we have decided to include one in the APIPM. This is because in modern multicomputer systems data transfer between nodes that are not directly connected does not require the intermediate nodes on the complete send path to be interrupted. Consequently, the time required for transferring a message from one node to another is not significantly influenced by the distance between the nodes.

### 4.3.2 APIPM Instruction Set

The APIPM instruction set (see Table 4.1) consists of four classes of operations:

1. *Generic image processing instructions*, i.e. the specialized parallelizable patterns described in Chapter 3.

2. *Memory instructions*, for allocation and copying of (image) data.

3. *I/O instructions*, for transporting data between the memory unit and external devices.

4. *Communication instructions*, for exchanging data among ASPIM units.

For reasons of simplicity, in the overview of Table 4.1 the operands (i.e., arguments) for each opcode have been left out. A complete overview will be given in Section 4.A.

In the instruction set we have included only two communication instructions (i.e., `SEND` and `RECV`). Collective communication operations are not included, as these can be implemented using the two point-to-point operations. The definition of the `SEND` and `RECV` instructions is identical to the standard blocking communication operations available in MPI [104] (i.e., `MPI_Send()` and `MPI_Recv()`).

In the abstract machine multiple real-world objects must be represented, which should be passed as parameters to the APIPM instructions. The most prominent objects are images, but templates, matrices, and the likes, are essential as well. In the instruction set we do not introduce a special data representation for each of these objects. As will be explained in detail in Section 4.A, we make use of *memory references* instead. Such references contain information about the internal data representation, but lack any semantic information. The semantics are determined by the APIPM instruction the memory reference is passed to as a parameter.

It is important to note that for several generic image processing operations in the instruction set *data element homogeneity* is required. This means that the scalar type and the dimensionality of the elements in multiple data structures passed as parameters to a single instruction must be identical. The restriction of data element homogeneity is enforced to acknowledge the differences between operations on homogeneous and heterogeneous types. If homogeneity would not be required additional casting or copying of data would be hidden inside the APIPM. For many instructions such additional tasks constitute a significant overhead, which must be made explicit.
Chapter 4. Semi-Empirical Modeling of Parallel Low Level Image Processing Operations

<table>
<thead>
<tr>
<th>opcode</th>
<th>generic image processing instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPOP</td>
<td>unary pixel operation</td>
</tr>
<tr>
<td>BPOPV</td>
<td>binary pixel operation (constant value as argument)</td>
</tr>
<tr>
<td>BPOPI</td>
<td>binary pixel operation (complete image as argument)</td>
</tr>
<tr>
<td>REDUCOP</td>
<td>global reduction operation</td>
</tr>
<tr>
<td>NEIGHOP</td>
<td>neighborhood operation</td>
</tr>
<tr>
<td>GCONVOP</td>
<td>generalized convolution</td>
</tr>
<tr>
<td>GEOMAT</td>
<td>geometric transformation (matrix as argument)</td>
</tr>
<tr>
<td>GEOROI</td>
<td>geometric transformation (region of interest)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opcode</th>
<th>memory instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>CREATE</td>
<td>allocate data block in memory unit</td>
</tr>
<tr>
<td>MEMCOPY</td>
<td>copy data in memory unit</td>
</tr>
<tr>
<td>DELETE</td>
<td>free up data block in memory unit</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opcode</th>
<th>I/O instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPORT</td>
<td>import data from external device</td>
</tr>
<tr>
<td>EXPORT</td>
<td>export data to external device</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>opcode</th>
<th>communication instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>SEND</td>
<td>send data to other ASIPM</td>
</tr>
<tr>
<td>RECV</td>
<td>receive data from other ASIPM</td>
</tr>
</tbody>
</table>

Table 4.1: Simplified APIPM instruction set.

4.3.3 Discussion

The definition of the abstract parallel image processing machine and its related instruction set is not complete, as it cannot be used as a basis for an actual implementation. This, however, is also not the reason for introducing the abstract machine. We stress that the APIPM is defined to serve as a basis for platform independent performance models. Many components deliberately have been left out of the specification, to keep the APIPM-based performance models as simple as possible.

The specification includes no information on how to load programs on each ASIPM unit. Also, no memory area has been identified in which APIPM programs are stored, and no program table or program counter has been defined. In other words, all hardware components that are essential to actually let programs run on the APIPM are left out of the specification. All such components are deemed too low level to be of any use in the performance models, and hence are not incorporated in the APIPM.

The APIPM instruction set is not complete either. For example, the APIPM lacks value comparison and conditional jump instructions. Such instructions have a relatively insignificant impact on execution time, and should not be incorporated in a performance model. Also, no instructions are included to set up special data structures, such as templates, and matrices. Again, such instructions are essential for the APIPM to run correctly, but the effect on the execution time in general is insignificant. In this respect, one may argue that the "DELETE" operation is expected to have no effect on performance either, and should have been left out of the instruction
set as well. However, this instruction is essential to see which memory references are still in use, and which are not. As will be discussed in Chapter 6, this knowledge is of great importance in the optimization and scheduling of complete applications.

4.3.4 Related Work

In the field of parallel low level image processing the definition and use of abstract machines is relatively new. In fact, the only references we know of are all from one research group at The Queen's University of Belfast in Northern Ireland. In several papers Crookes et al. [35, 36] discuss the design of a Portable Parallel Abstract Machine (PPAM), whose instruction set is based on the Image Algebra Language (IAL), which in turn is based on Image Algebra [131]. As discussed in Section 2.2.2, IAL is a machine independent programming language capable of parallel implementation on a range of distributed memory parallel machines. In later work both the PPAM and IAL have been extended considerably. The languages I-BOL [20] and TULIP [155] are two of the more sophisticated extensions of IAL. In later papers, a more recent version of the PPAM is referred to as the EPIC abstract machine [34]. The basic ideas behind the abstract machines have not changed throughout the years, so in the remainder of this discussion we will only consider the original PPAM.

The Portable Parallel Abstract Machine is designed as the hypothetical target machine for the IAL compiler. The PPAM consists of two main components: a sequential controller (implemented on a front end machine, such as a SUN workstation), which communicates with an abstract parallel co-processor (see Figure 4.2). This co-processor can be any kind of parallel system. The use of the parallel co-processor by the sequential controller can be thought of in rather the same way as a floating point co-processor is used by a microprocessor. Although the PPAM design is dissimilar to that of the APIPM, its related instruction set is almost identical to ours.

The main differences from our work stem from the fact that the PPAM is used as an aid in the design of a parallel compiler rather than as a basis for a performance model definition. On the one hand, the PPAM incorporates a higher level of abstraction than the APIPM, as the communication aspects of parallel execution are not incorporated in its definition. On the other hand, the inclusion of low level abstract hardware components (such as an instruction control unit) often makes the abstraction level much lower. Essentially, the differences in the design of the two abstract machines are explained by the fact that the two research directions are non-overlapping.

Figure 4.2: Portable Parallel Abstract Machine [36].
4.4 APIPM-Based Performance Models

In our software architecture, all library operations are assumed to be implemented by concatenation of APIPM instructions only. Also, it is assumed that the execution time of each library operation can be partitioned into independent time intervals, each corresponding to the cost of a single APIPM instruction. The performance of a library operation is then simply obtained by adding the execution times of all APIPM instructions used.

Similar to the AFM-based models described in Section 4.1.2, this idea is formalized as follows. Let \( I = \{I_1, I_2, \ldots, I_n\} \) be the APIPM instruction set. Also, let \( P = \{P_1, P_2, \ldots, P_n\} \) be the set of performance values for all \( n \) instructions in \( I \). We assume that, for any given system capable of running APIPM instructions, and for each instruction in \( I \), \( P_i \) can be obtained by benchmarking. Also, let \( L = \{L_1, L_2, \ldots, L_m\} \) be the set of all \( m \) operations implemented using instructions in \( I \) only. For all library operations \( L_x \ (x \in \{1, \cdots, m\}) \) we define \( L_x = \{I_1, I_2, \ldots, I_n\} \), in combination with the total number of occurrences (or count) of each APIPM instruction in \( L_x \): \( C_x = \{C_{I_1,x}, C_{I_2,x}, \cdots, C_{I_n,x}\} \). The count of each instruction can have any value in \( \mathbb{N} \) (including 0). The expected total execution time of each library operation \( L_x \) is then obtained by

\[
T_{L_x} = \sum_{i=1}^{n} C_{I_i,x} P_{I_i}.
\]

Similarly, the expected total execution time of any application implemented using our library is obtained by adding the execution times of each library operation used.

A problem with the simplistic model formalized here is that most APIPM instructions are not single static entities. This is because the execution of an instruction is often dependent on the values of its operands. Therefore, a static entity for each possible operand combination must be incorporated in our model. To avoid an explosion of the number of static entities we allow each instruction \( I_i \) and each value \( P_{I_i} \) to be parameterized. Because the operands of the APIPM instructions are discussed in the appendix to this chapter (Section 4.A), a detailed overview of the model parameterization is deferred to the appendix as well. To give a straightforward example, however, in almost all APIPM instructions a 'datatype' parameter is incorporated (e.g., giving \( I_i('int') \) and \( I_i('float') \)). Also, a 'data-input-size' parameter is required for most performance values in \( P \) (e.g., giving \( P_{I_i.datatype}(size) \)). The choice of model parameters is dependent on the actual implementation of each APIPM instruction. For more detailed information we refer to Section 4.B.

Benchmarking

To capture system variation without having to rely on platform specific model parameters, the semi-empirical modeling approach requires an additional benchmarking phase to be performed. For our software architecture to be used on a specific platform, benchmarking results need to be obtained only once. As long as the underlying hardware layers and supporting software layers (e.g., operating system, compiler, etcetera)
4.4. APIM-Base Performance Models

Figure 4.3: Performance estimation in case of neighborhood operations and generalized convolutions, whose performance values depend on two input size parameters — i.e., image size and kernel size. Any required performance value (above represented by the small black square) is obtained by simple bilinear interpolation. In this example, measurements were performed for image sizes of 40, 90, and 250 Kb, each combined with kernel sizes of 9, 25, and 49 bytes.

are not upgraded, the same set of measurement results can be applied for estimation of any application implemented using our parallel image library.

As in the adaptive sampling approach of Section 4.1.2, in our software architecture each APIM instruction is measured for multiple input sizes. In contrast to adaptive sampling, however, we do not define a fixed time budget for all measurements. By default we use a small, predefined set of input sizes for all benchmarking operations. To avoid the benchmarking phase to be unacceptably lengthy, the set of input-sizes may be user-defined as well.

To capture non-linear performance growth without having to perform measurements for any possible workload, between each pair of measured performance values performance growth is taken to be piecewise linear. For estimation of instructions whose performance value is dependent on one data input size parameter this interpolation is straightforward. The performance values of neighborhood operations and generalized convolution operations, however, are dependent on two data input size parameters - i.e., the size of the input image and the size of the kernel or template structure. As is indicated in Figure 4.3, in such situation we apply bilinear interpolation to obtain the required performance estimation.
4.4.1 Extended Model for Point-to-Point Communication

Whereas the performance model described above is sufficient for all sequential APIPM instructions, an extension is required for the two communication operations (i.e., SEND and RECV). First, this is because an accurate prediction of the end-to-end communication time usually cannot be obtained by considering the time a processor is busy executing a SEND or a RECV instruction alone. Second, in its current form the model does not closely match the capabilities of the communication instructions as defined in MPI. Most notably, the impact of a message's memory layout on communication costs is not incorporated in the model. This is an important point, as one of the tasks of the scheduler of Section 2.3.2 is to make decisions regarding the domain decomposition of an application under consideration. Depending on the type of such domain decomposition, it may be necessary to communicate data stored noncontiguously in memory. As was shown by Prieto et al. [125], knowledge of a message's memory layout is important, as non-unit-stride memory access may have a severe impact on performance due to caching. Also, the MPI operations may handle the transmission of noncontiguous data differently from contiguous blocks, possibly causing additional overheads due to the packing of data into a contiguous buffer.

To incorporate such essential cost factors we have designed an extended model for point-to-point communication. The model, called P-3PC (or the Parameterized model based on the Three Paths of Communication), closely resembles other models described in the literature (e.g., the Postal Model [11, 21], LogP [38], and LogGP [1]). The model is capable of modeling the essential communication patterns as used in data parallel image processing applications. In addition, and in contrast to the models mentioned above, it is also capable of accurately predicting the communication costs related to any type of domain decomposition. As this topic is outside the scope of the current chapter, an extensive overview of the P-3PC model and its capabilities is given in Chapter 5. In the evaluation of the APIPM-based models presented in the remainder of this chapter, all P-3PC specific modeling properties have been left out.

4.4.2 Discussion

The most important advantage of the APIPM-based performance models is that predictions are based on very high level instructions — even in comparison with the AFM-based models of Section 4.1.2. It would have been possible to define a model on the basis of much lower level instructions as well, but execution times of such instructions tend to be less independent than those of higher level instructions. This is mainly due to optimizations performed by the applied compiler. Also, it is much more difficult to obtain accurate values for lower level instructions, due to the inherent intrusiveness of the benchmarking process.

A possible drawback of the models is that the instructions and related performance values are parameterized with quite a large number of instruction behavior and workload indicators. Obtaining accurate performance values for all possible combinations of parameters is both costly and difficult. However, it is possible to combine several parameters to obtain a more general indicator. As an example, promising candi-
dates for parameter merging are those that relate to data structure sizes (e.g., width, height, depth, etc.). Furthermore, the benchmarking tool that plays an essential role in our software architecture is implemented such that it allows a user to set regions of interest, to restrict the set of all possible measurements. In addition, it is possible to let the benchmarking process be run in parallel on multiple nodes within a target architecture, to reduce the benchmarking costs even further.

4.5 Measurements and Validation

In this section we show how a realistic image processing application, implemented using our software architecture, is executed in parallel. The application is highly relevant as it incorporates all of the important APIPM instructions defined in Section 4.3.2. First, a short description is given of the underlying algorithm. Next, both a straightforward sequential implementation as well as its related parallel execution are discussed. Finally, measured results are compared with APIPM model predictions.

4.5.1 Detection of Curvilinear Structures

As discussed in [55, 56], the problem of detecting curvilinear structures in images is solved by considering the second order directional derivative in the gradient direction, for each possible line direction. This is achieved by applying anisotropic Gaussian filters, parameterized by orientation \( \theta \), smoothing scale \( \sigma_v \) in the line direction, and differentiation scale \( \sigma_w \) perpendicular to the line (Figure 4.4), given by

\[
r''(x, y, \sigma_v, \sigma_w, \theta) = \sigma_v \sigma_w |f^{\sigma_v, \sigma_w, \theta}_{x^w y^w}| \frac{1}{b^{\sigma_v, \sigma_w, \theta}}. \tag{4.1}
\]

When the filter is correctly aligned with a line in the image, and \( \sigma_v, \sigma_w \) are optimally tuned to capture the line, filter response is maximal. Hence, the per pixel maximum line contrast over the filter parameters yields line detection:

\[
R(x, y) = \arg \max_{\sigma_v, \sigma_w, \theta} r''(x, y, \sigma_v, \sigma_w, \theta). \tag{4.2}
\]

This directional filtering problem can be implemented sequentially in many different ways. For each orientation \( \theta \) it is possible to create a new filter based on \( \sigma_w \) and \( \sigma_v \). In effect, this yields a rotation of the filters, while the orientation of the input image remains fixed. Another possibility is to keep the orientation of the filters fixed, and to rotate the input image instead. Yet another solution is to integrate the notion of orientation in the filter operation itself. In this case image pixels are accessed according to the size of the neighborhood as well as the given orientation.

In this example, we have implemented the operation by applying fixed filters to rotated image data. We have selected this implementation as we have found it to be the solution of choice for several researchers in image processing. As such, the implementation reflects parallelization problems encountered in a realistic situation. It should be noted, however, that the alternative sequential implementations presented in Chapter 7 yield better sequential as well as parallel performance.
The main body of the sequential implementation is presented in pseudo code in Listing 4.1. The program starts by rotating the original input image for a given orientation \( \theta \). In addition, for all \((\sigma_v, \sigma_w)\) combinations the filtering is performed by six library operations executed in sequence. First, \( f_{w,v}^{\sigma_v,\sigma_w,\theta} \) and \( b_{w,v}^{\sigma_v,\sigma_w,\theta} \) (or Filtered1.IM and Filtered2.IM, respectively) are produced by executing two generalized convolutions, each with the appropriate parameters. For cost effectiveness the Gaussian convolutions are performed by applying two 1-dimensional filters in both cases. Next, the result of Equation (4.1) is obtained by executing two binary pixel operations, one having an image, the other having a constant value as argument. Finally, the result image is rotated back to match the orientation of the input image, and the maximum response image is obtained.

Figure 4.5(a) gives a typical example of an image that is used as input to the program. The result obtained after applying the program for a reasonably large parameter subspace of \((\sigma_v, \sigma_w, \theta)\) is shown in Figure 4.5(b). On a state-of-the-art sequential machine the program may take from a few minutes up to several hours to complete, depending on the size of the input image and the extent of the chosen parameter subspace. Consequently, for the directional filtering program parallel execution is highly desired.

FOR all orientations \( \theta \) DO
\[
\text{Rotated.IM} = \text{GeometricOp(Original.IM, "rotate", } \theta)\; ;
\]
FOR all smoothing scales \( \sigma_v \) DO
\[
\text{FOR all differentiation scales } \sigma_w \text{ DO}
\]
\[
\text{Filtered1.IM} = \text{GenConvOp(Rotated.IM, "gauss", } \sigma_w, \sigma_v, 2, 0)\; ;
\]
\[
\text{Filtered2.IM} = \text{GenConvOp(Rotated.IM, "gauss", } \sigma_w, \sigma_v, 0, 0)\; ;
\]
\[
\text{Detected.IM} = \text{BinPixImArgOp(Filtered1.IM, "absdiv", Filtered2.IM)}\; ;
\]
\[
\text{BackRotated.IM} = \text{GeometricOp(Detected.IM, "rotate", } -\theta)\; ;
\]
\[
\text{Contrast.IM} = \text{BinPixImArgOp(Contrast.IM, "max", BackRotated.IM)}\; ;
\]

Listing 4.1: Pseudo code for the directional filtering program.
4.5. Measurements and Validation

4.5.2 Parallel Execution

As all parallelization issues are shielded from the user, the pseudo code of Listing 4.1 directly constitutes a program that can be executed in parallel as well. Optimization of the efficiency of the program is to be taken care of by the software architecture’s scheduling component. For this evaluation, however, we have created two different schedules for the program by hand. In the first schedule all library operations are forced to run in a data parallel manner, using all available processors. The second schedule differs from the first in that the last two operations in the innermost loop of the program are run on one node only.

In both schedules the Original.IM structure is broadcast to all nodes. This is because the structure is applied in the initial rotation operation, which expects it to have a data access pattern of type 'other' (see Section 3.4). This broadcast needs to be performed only once, as Original.IM is not updated in subsequent operations. In addition, in both schedules the first four operations in the innermost loop are executed locally on partial image data structures. The only need for communication is in the exchange of shadow regions in the two Gaussian convolution operations.

In the first schedule the last two operations in the innermost loop are run in parallel as well. This requires the distributed image Detected.IM to be available in full at each node, because it has an access pattern of type 'other' in the back-rotation operation. This is achieved by executing a gather-to-all operation, which is logically equivalent to a gather operation followed by a broadcast. Finally, a partial maximum response image Contrast.IM is calculated on each node, which requires a final gather operation to be executed just before termination of the program. In the second schedule the last two operations are not executed in parallel. As a result, the intermediate result image Detected.IM is gathered to the single node that produces both the back-rotated image, as well as the complete maximum response image.

Figure 4.5: (a) Typical 1000×554 input image obtained from the Apollo training manual "Apollo Spacecraft & Systems Familiarization" (March 13, 1968). National Aeronautics and Space Administration (NASA), Office of Policy and Plans, NASA History Office. Used by kind permission. (b) Maximum response image obtained after application of the directional filtering program.
It is the purpose of the architecture's scheduling component to pick the optimal solution out of multiple competing schedules of this kind. In the following we will show that the APIPM-based performance models are powerful enough to allow the scheduler to make such decisions correctly.

### 4.5.3 Performance Evaluation

To initialize the APIPM-based performance models we have performed a small set of benchmarking operations. For each instruction used in the directional filtering program not more than two measurements were performed, i.e. for input sizes of $200^2$ and $1000^2$ elements. Model predictions for each instruction and for each required input size were obtained as indicated in Section 4.4.

The benchmarking operations, as well as the directional filtering program were executed on the 24-node homogeneous DAS-cluster (Distributed ASCI Supercomputer [7]) located at the University of Amsterdam. All nodes in the cluster contain a 200 Mhz Pentium Pro with 64 MByte of EDO-RAM, and are connected by a 1.2 Gbit/sec full-duplex Myrinet SAN network. The nodes run the RedHat Linux 6.2 operating system. At the time of measurements, 4 nodes in the system were unusable. As a consequence, performance results are presented only for up to 20 processors.

Based on intuition alone a programmer would have great difficulty deciding which of the two schedules described in the previous section should be executed. Clearly, a schedule is preferred if the set of operations unique to that schedule is faster than the set of operations unique to another schedule. Hence, for the directional filtering program the first schedule is preferred if:

$$
\theta \sigma (P_{\text{rotate}}(\text{size}/N) + P_{\max}(\text{size}/N) + P_{\text{broadcast}}(\text{size})) + P_{\text{gather}}(\text{size}) < \theta \sigma (P_{\text{rotate}}(\text{size}) + P_{\max}(\text{size}))
$$

where $N$ denotes the number of nodes, and $\theta \sigma$ the size of the parameter subspace. For the first schedule the large number of broadcasts is expected to have a significant impact on performance. For the second schedule the many rotations of non-partitioned image data is expected to be costly.

Based on the benchmarking results we are able to decide which schedule is optimal. As shown in Figure 4.6 (depicting the complete execution time of both schedules), our models indicate that the first schedule is always preferred - for any number of processors. Clearly, broadcasting a full-sized image structure is not as expensive as performing the image rotation sequentially on one node. The 'hops' in the graph of schedule 1 are explained by the fact that the broadcast operation is implemented using a spanning binomial tree (SBT), which has a cost related to $\log N$. Figure 4.7 shows similar predictions for a smaller input image, but for a larger parameter subspace.

To test the accuracy of our performance models we have executed the directional filtering program for both schedules. The resulting mean execution times for each run are included in the graph of Figure 4.6 as well. Error bars are not shown, as the performance of the DAS is quite stable. In most situations measured lower and upper bounds are within 0.5 seconds of the mean execution times. The presented results indicate that the model predictions for both schedules are highly accurate - for any
Figure 4.6: Comparison of model predictions and measurements for the two program schedules. Results for directional filtering of extended Apollo image of size 1098x1098, and for a parameter subspace including 12 orientations and 4 ($\sigma_v, \sigma_w$) combinations.

Figure 4.7: Comparison of predictions and measurements for input image of size 707x707, and for a parameter subspace including 36 orientations and 4 ($\sigma_v, \sigma_w$) combinations.
number of processors. Even worst case predictions are within 5.5% of the measured values. It is noteworthy, however, that our models are slightly optimistic in all situations. This is explained by the fact that the mean performance values measured in the benchmarking process tend to be somewhat lower than what is actually obtained at application run time. This is because 'outliers' obtained during benchmarking are not included in our database of performance values, while extremely high values may still occur during normal runs of a particular application.

The graph of Figure 4.8 shows that our performance models are capable of providing accurate estimations at the lowest level of APIPM instructions as well. The accumulated estimations on a per-instruction basis are optimistic as well as pessimistic, depending on the applied instruction. The importance of this graph, however, lies in the fact that errors in the estimations for the most significant instructions applied in the application of Listing 4.1 are, in general, not more than 10%. As a consequence, we feel that a sufficiently high level of estimation accuracy is obtained for the models to be applied in our software architecture's optimization process.

Given schedule 1, it can be derived from the models that the impact of communication (especially the repeated broadcast) on overall application performance is huge. Figure 4.9 shows that for 16 nodes the program spends almost half of its time communicating. For 64 nodes 84.1% of the time is lost in all communication steps combined, and 71.1% in broadcasting alone. Although parallel performance is often significantly better for alternative sequential implementations of this particular line detection problem (see Chapter 7), communication costs do play an important role in almost any parallel application. Therefore, it is essential for our performance models to also provide accurate estimations for the SEND and RECV instructions. The next chapter of this thesis is devoted entirely to the modeling of these basic communication operations, and includes a detailed evaluation of our performance estimations as well.
4.6 Conclusions

In this chapter we have described the performance estimation approach as applied in our software architecture. We have introduced the notion of *semi-empirical modeling*, which is a performance estimation technique based on a combination of relevant abstraction, simple *modeling*, and domain-specific *measurement*. We have compared the technique with existing estimation approaches, and have shown semi-empirical modeling to be similar to a combination of two techniques described in the literature: (1) the AFM-based approach of narrow spectrum benchmarking that incorporates very high level system abstractions, and (2) the approach of adaptive sampling that captures system variance by measuring execution times for multiple workloads.

We have indicated that in our semi-empirical modeling approach all abstractions are introduced on the basis of a high level abstract machine specification for parallel image processing (the APIPM), and a related instruction set. Also, we have shown the definition of the abstract machine to reflect the relevant hardware components and behavior common to all projected target platforms for our software architecture (i.e., state-of-the-art homogeneous commodity clusters). Subsequently, we have indicated how to define a simple, linear performance model on the basis of the APIPM-abstractions. Finally, we have shown how domain-specific benchmarking is incorporated for estimation of system variance.

A comparison of model estimations and experimental measurements has indicated that, for a realistic image processing application, the APIPM-based performance models are highly accurate. The models are capable of providing good estimations for full applications, as well as for any constituent subtask. Given these results we are confident in that the core of our software architecture forms a powerful basis for automatic optimization of a wide range of parallel low level image processing applications.

Figure 4.9: *Predicted impact on communication (for schedule 1).*
The evaluation of our performance models as presented in this chapter is not complete, as the costs related to interprocess communication hardly have been touched upon. Because communication is such a prominent cost factor in many parallel applications, all modeling aspects related to this issue are deferred to the next chapter. Also, the evaluation has not shown that optimization on the basis of our performance models indeed results in highly efficient parallel applications. All issues related to application optimization and scheduling are discussed extensively in Chapter 6. An assessment of the effectiveness of our software architecture in providing significant performance gains is presented in Chapter 7. Finally, in the appendix to this chapter a more detailed overview is given of the APIPM instruction set, and the related model parameterization.

4.A APIPM Instruction Set Definition

This section presents a detailed discussion of the APIPM instruction set definition. A complete overview of the instructions and their related operands is given in Tables 4A.1 - 4A.3.

Memory References

In the Abstract Parallel Image Processing Machine multiple real-world objects need to be represented. The most prominent objects are images, but templates, matrices, and the likes, are essential as well. In the instruction set we do not introduce a special data representation for each of these objects. Instead, we use memory references that contain information about the internal data representation, but lack any information on the semantics of the data referenced to. The semantics are determined by the APIPM instruction the memory reference is passed to as a parameter.

Given the notion of memory references, the operands (arguments) of the instructions fall into one of four categories:

1. Memory references to single data elements (smref). Operands of this type refer to single data elements stored in main memory. Apart from a pointer to a memory location, it holds information regarding the scalar type and dimensionality of the data element stored. In a realistic program a single memory reference usually represents a pixel value of a certain scalar type and dimension.

2. Memory references to aggregated data elements (amref). Operands of this type refer to aggregations (such as arrays) of data elements stored in main memory. Apart from a pointer to the memory location containing the first data element, it also contains information regarding the size and origin of the domain of the aggregated structure, in combination with the type and dimensionality of the structure’s elements. The size and origin of an n-dimensional aggregated structure are both represented by an n-dimensional vector. A memory reference of this type pointing to a data aggregation of size 1 (thus: containing only one element) is considered equal to the single data element
### CREATE

<table>
<thead>
<tr>
<th>opcode</th>
<th>operands</th>
<th>memory instructions</th>
</tr>
</thead>
</table>
| CREATE | amref, vector, vector, string, value | #1: reference to destination data structure  
#2: domain size of destination structure  
#3: domain origin of destination structure  
#4: scalar type of data elements in destination structure  
#5: dimensionality of data elements in destination structure |

### DELETE

<table>
<thead>
<tr>
<th>opcode</th>
<th>operands</th>
<th>memory instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>DELETE</td>
<td>amref</td>
<td>#1: reference to source data structure</td>
</tr>
</tbody>
</table>

### MEMCOPY

<table>
<thead>
<tr>
<th>opcode</th>
<th>operands</th>
<th>memory instructions</th>
</tr>
</thead>
</table>
| MEMCOPY | amref, amref, vector, vector, value | #1: reference to source data structure  
#2: reference to destination data structure  
#3: offset from start of source data structure  
#4: offset from start of destination data structure  
#5: number of data elements |

### UPOP

<table>
<thead>
<tr>
<th>opcode</th>
<th>operands</th>
<th>memory instructions</th>
</tr>
</thead>
</table>
| UPOP | amref, amref, string | #1: reference to source data structure  
#2: reference to destination data structure  
#3: name of internal unary pixel operation |

### BPOPV

<table>
<thead>
<tr>
<th>opcode</th>
<th>operands</th>
<th>memory instructions</th>
</tr>
</thead>
</table>
| BPOPV | amref, amref, string, smref | #1: reference to source data structure  
#2: reference to destination data structure  
#3: name of internal binary pixel operation  
#4: reference to single argument value |

### BPOPI

<table>
<thead>
<tr>
<th>opcode</th>
<th>operands</th>
<th>memory instructions</th>
</tr>
</thead>
</table>
| BPOPI | amref, amref, string, amref | #1: reference to source data structure  
#2: reference to destination data structure  
#3: name of internal binary pixel operation  
#4: reference to argument data structure |

### REDUCOP

<table>
<thead>
<tr>
<th>opcode</th>
<th>operands</th>
<th>memory instructions</th>
</tr>
</thead>
</table>
| REDUCOP | amref, smref, string | #1: reference to source data structure  
#2: reference to single destination value  
#3: name of internal reduction operation |

Table 4A.1: **APIPM instruction set.**
<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>Generic Image Processing Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEIGHOP</td>
<td>amref, amref, string, amref</td>
<td>#1: reference to source data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#2: reference to destination data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#3: name of internal neighborhood operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#4: reference to kernel structure</td>
</tr>
<tr>
<td>GCONVOP</td>
<td>amref, amref, string, string, amref</td>
<td>#1: reference to source data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#2: reference to destination data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#3: name of internal binary pixel operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#4: name of internal reduction operation</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#5: reference to kernel structure</td>
</tr>
<tr>
<td>GEOMAT</td>
<td>amref, amref, amref, smref, string, vector</td>
<td>#1: reference to source data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#2: reference to destination data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#3: reference to transformation matrix</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#4: reference to single background value</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#5: interpolation type</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#6: translation vector</td>
</tr>
<tr>
<td>GEOROI</td>
<td>amref, amref, vector, smref</td>
<td>#1: reference to source data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#2: reference to destination data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#3: offset from start of source data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#4: reference to single background value</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Opcode</th>
<th>Operands</th>
<th>I/O Instructions</th>
</tr>
</thead>
<tbody>
<tr>
<td>IMPORT</td>
<td>amref, value, value</td>
<td>#1: reference to destination data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#2: unique external data structure identifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#3: unique external device number</td>
</tr>
<tr>
<td>EXPORT</td>
<td>amref, value, value</td>
<td>#1: reference to source data structure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#2: unique external data structure identifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>#3: unique external device number</td>
</tr>
</tbody>
</table>

Table 4A.2: APIM instruction set (continued).
4. **APIPM Instruction Set Definition**

<table>
<thead>
<tr>
<th>opcode</th>
<th>operands</th>
<th>communication instructions</th>
</tr>
</thead>
</table>
| SEND   | amref, vector, value, value | #1: reference to source data structure  
         |          | #2: offset from start of source data structure  
         |          | #3: number of data elements  
         |          | #4: unique destination ASIPM identifier |
| RECV   | amref, vector, value, value | #1: reference to destination data structure  
         |          | #2: offset from start of destination data structure  
         |          | #3: number of data elements  
         |          | #4: unique source ASIPM identifier |

**legend:**

- **amref** memory reference to aggregated data elements
- **smref** memory reference to single data element
- **string** string value (constant)
- **value** numerical value (scalar)
- **vector** numerical value (vector)

Table 4A.3: *APIPM instruction set (continued).*

Reference described above. References to aggregations of size 1 and references to single data elements can be interchanged at will. In a realistic program a memory reference of this type usually refers to image data.

3. **Numerical (constant) values** (*value* and *vector*). Operands of this type refer to single numbers or vectors of single numbers, and are used to represent sizes, positions, etcetera.

4. **String (constant) values** (*string*). Operands of this type refer to character strings recognized by each sequential image processing unit (SIPU). A string value determines the behavior of an instruction, and is either used as an *operation indicator*, or as a *type indicator*.

Operation indicators refer to internal operations recognized by the SIPU. As an example, indicators such as "NEGATE" and "SQRT" can be used to represent valid unary pixel operations.

Type indicators represent additional information required for an operation to be executed. For example, the memory allocation instruction "CREATE" needs an indicator for the specification of the datatype of each element in the structure to be allocated. Also, the geometric transformation instruction "GEOMAT" needs an indicator for the specification of the type of interpolation to be used.
Sequential Instructions

The instruction set contains memory operations to allocate and free memory space ("CREATE" and "DELETE"), and to copy data from a source area to a destination area ("MEMCOPY"). As described above, references to single data elements are equal to aggregated data structures of size 1. This means that a newly created aggregated data structure of size 1 can be used as an argument to an instruction that requires a single data element reference as one of its operands (such as the 'smref' operand in the "REDUCOP" instruction).

Two I/O operations are available in the instruction set to transport data to and from external devices ("IMPORT" and "EXPORT"). Apart from a reference to aggregated data, both instructions need a unique device number and identifier to specify the apparatus itself and the data structure residing on that device.

The generic image processing operations in the instruction set are those we have discussed in Chapter 3. Although we have indicated that many image processing operations can be performed in-place, in all but the reduction operation ("REDUCOP") both a source image reference and a destination image reference are required. This scheme does not introduce additional copying of data because the same reference could be given as an argument twice.

As stated in Section 4.3.2, all image processing operations that have a reference to argument data structures or kernel structures as an operand require data element homogeneity, to acknowledge the differences between operations on homogeneous and heterogeneous types. Homogeneity means that the scalar type and the dimensionality of the data elements of both the source structure and the additional structure must be identical. Data element homogeneity is not required for destination image data structures, as the resulting scalar type and dimensionality of the data elements is determined by the type of internal SIPU instruction performed.

Communication Instructions

The instruction set includes two communication operations for the exchange of data between two ASIPMs. Data can be sent to another ASIPM by using the "SEND" operation. Data can be received from another ASIPM by using the "RECV" operation. These point-to-point operations provide reliable message transfer. This means that a message sent is always received correctly, and that no additional checks for errors are needed. Collective communication operations (i.e.: communication routines that involve multiple ASIPMs) are not included in the instruction set. This is because a complete set of collective communication routines can be created using the two point-to-point communication operations.

The first operand in the "SEND" operation specifies a send buffer in the main memory of the sending ASIPM from which the message data is taken. The starting point in the send buffer and the number of data elements to be sent are specified by the second and third operand. The last operand specifies the unique identifier of the receiving (destination) ASIPM. The scalar type and dimensionality of the data elements sent are specified in the reference to the source data structure.
The "RECV" operation requires a receive buffer to store incoming message data. Similar to the "SEND" operation, the offset in the receive buffer and the number of data elements to be received are specified by the second and third operand. Also, the last operand specifies the unique identifier of the sending (source) ASIPM.

All send and receive operations must be matched. This means that for each message sent the destination node needs to execute a receive operation with the sending node as source identifier. Furthermore, in both the send call and the receive call the number of data elements, as well as with the scalar type and dimensionality of the elements all must be identical. Also, all messages that are sent over the communication channels are non-overtaking. This means that if one ASIPM sends two messages in succession to the same destination, the first message will always be received first.

We assume that the two communication operations are blocking. For the "SEND" operation this means that it does not return until the message has been copied into a matching receive buffer (on another ASIPM), or stored away safely in a local temporary buffer. For the "RECV" operation this means that it does not return until the message has been fully copied into its receive buffer. Although not expected under normal circumstances, it is possible for a receive call to complete before its matching send call has completed.

4.B APIPM Model Parameterization

This section presents a detailed discussion of the APIPM model parameterization. A complete overview of the parameterized model instructions and related performance values is given in Tables 4B.1 and 4B.2.

Parameterized Model Instructions

As stated in Section 4.4, a problem with our simple, linear performance model is that most APIPM instructions are not single static entities. This is because the execution of an instruction often depends on the values of its operands. Therefore, a static entity for each possible operand combination must be incorporated in the model. To avoid an explosion of the number of static entities we allow each instruction \( l_i \) to be parameterized instead. The number of parameters for a model instruction is not necessarily identical to the number of operands of the related APIPM instruction. For example, the background value required in the geometric transformation instructions (Table 4A.2, operand #4 in either "GEOMAT" or "GEOROI") does not call for additional static model instructions. The execution of these instructions is expected to be independent of the applied background pixel value.

Essentially, any possible source of relevant change in instruction behavior must be captured in a model parameter. Here, the task of choosing relevant model parameters is steered by the actual implementations of each APIPM instruction in our software library. An overview of the parameterized model instructions related to all sequential APIPM instructions is presented in Table 4B.1. The communication instructions have been left out, as parameterization of these instructions (a.o., including the memory layout of a message, see Chapter 5) is outside the scope of this chapter.
<table>
<thead>
<tr>
<th>instruction</th>
<th>parameterized model instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>UPOP</td>
<td>$I_{UPOP}(\text{opname}, \text{idim}, \text{stype}, \text{ival})$</td>
</tr>
<tr>
<td>BPOPV</td>
<td>$I_{BPOPV}(\text{opname}, \text{idim}, \text{stype}, \text{ival}, \text{aval})$</td>
</tr>
<tr>
<td>BPOPI</td>
<td>$I_{BPOPI}(\text{opname}, \text{idim}, \text{stype}, \text{ival}, \text{aval})$</td>
</tr>
<tr>
<td>REDUCOP</td>
<td>$I_{REDUCOP}(\text{opname}, \text{idim}, \text{stype}, \text{ival})$</td>
</tr>
<tr>
<td>NEIGHOP</td>
<td>$I_{NEIGHOP}(\text{opname}, \text{idim}, \text{kdim}, \text{stype}, \text{ival}, \text{kval})$</td>
</tr>
<tr>
<td>GCONVOP</td>
<td>$I_{GCONVOP}(\text{popname}, \text{ropname}, \text{idim}, \text{kdim}, \text{stype}, \text{ival}, \text{kval})$</td>
</tr>
<tr>
<td>GEOMAT</td>
<td>$I_{GEOMAT}(\text{idim}, \text{stype}, \text{ival}, \text{mtype}, \text{itype})$</td>
</tr>
<tr>
<td>GEOROI</td>
<td>$I_{GEOROI}(\text{idim}, \text{stype}, \text{ival})$</td>
</tr>
<tr>
<td>CREATE</td>
<td>$I_{CREATE}(\text{idim}, \text{stype})$</td>
</tr>
<tr>
<td>MEMCOPY</td>
<td>$I_{MEMCOPY}(\text{idim}, \text{stype})$</td>
</tr>
<tr>
<td>DELETE</td>
<td>$I_{DELETE}(\text{idim}, \text{stype})$</td>
</tr>
<tr>
<td>IMPORT</td>
<td>$I_{IMPORT}(\text{idim}, \text{stype})$</td>
</tr>
<tr>
<td>EXPORT</td>
<td>$I_{EXPORT}(\text{idim}, \text{stype})$</td>
</tr>
</tbody>
</table>

**legend:**
- **aval**: value indicator of argument data structure
- **idim**: dimensionality of source data structure
- **itype**: type of interpolation used in geometric operation
- **ival**: value indicator of source data structure
- **kdim**: dimensionality of kernel data structure
- **kval**: value indicator of kernel data structure
- **mtype**: type of matrix used in geometric operation
- **opname**: name of internal operation
- **popname**: name of internal binary pixel operation
- **ropname**: name of internal reduction operation
- **stype**: scalar type of data elements

Table 4B.1: Parameterized model instructions.
The parameters for the model instructions presented in Table 4B.1 fall into one of four categories:

- **Type indicators.** Three different type indicators are introduced: 'stype', 'mtype', and 'itype'. The scalar type parameter can have any of the values "byte", "short", "int", "float", and "double". The type of the data structure elements often will have an important impact on the behavior of an instruction. As an example, arithmetic floating point operations (such as sqrt) are often much more expensive than the non-floating point versions.

The interpolation type indicator is applied in geometric transformation operations, and can have any of the values "nearest" and "linear". The matrix type indicator decides which geometric transformation is performed. Currently, the available valid values are "rotate", "reflect", and "scale".

- **Value indicators.** Three different value indicators are introduced: 'ival', 'aval', and 'kval'. This is because the actual values present in a data structure may have an important impact on the performance of an instruction. For example, when the value '1' is presented to a base-10 logarithm operation '0' is returned. However, if '0' is presented to the operation an error value is returned, and an exception may be raised, possibly causing additional overhead. Currently valid values for the parameters 'ival', 'aval', and 'kval' are:
  - "ALL0" (most elements have the value 0),
  - "ALL1" (most elements have the value 1),
  - "0TO1" (most elements have a value between 0 and 1), and
  - "ANY" (no value indication, used by default).

- **Dimensionality indicators.** Two dimensionality indicators are introduced: 'idim' and 'kdim'. In our software library a choice has been made to provide different implementations for operations on 2-dimensional and 3-dimensional images. Also, multiple fundamental kernel operations have been implemented to allow for optimization of operations that make use of separable kernel data. For this reason the 'idim' parameter can have the value "2D" or "3D". The 'kdim' parameter either can have the value "1D" or "nD". Here we use "nD" to represent non-separable kernel dimensionality.

- **Operation indicators.** Operation indicators refer to the internal operations recognized by the Sequential Image Processing Unit (SIPU). See also Section 4.A.

### Parameterized Performance Values

The performance values in set \(P\) are not single static entities either. This is because the execution time of many instructions is dependent on the size of the workload. For this reason we also have parameterized the performance values related to the model instructions. In Table 4B.2 an overview is given of the parameterized performance values related to the sequential operations in the APIPM instruction set.
# Parameterized Performance Values

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P_{IUPROP}(opname, idim, style, ival) (ddim, w, h, d)$</td>
<td>Performance value related to the dimensionality of the data elements (usually: pixels) of the structure passed to a given instruction.</td>
</tr>
<tr>
<td>$P_{IBPROP}(opname, idim, style, ival, aval) (ddim, w, h, d)$</td>
<td></td>
</tr>
<tr>
<td>$P_{IREDCOP}(opname, idim, style, ival) (ddim, w, h, d)$</td>
<td></td>
</tr>
<tr>
<td>$P_{INEIGHOP}(opname, idim, style, ival, kval) (ddim, w, h, d, kw, kh, kd)$</td>
<td></td>
</tr>
<tr>
<td>$P_{IGCONVOP}(opname, ropname, idim, kdim, style, ival, kval) (ddim, w, h, d, kw, kh, kd)$</td>
<td></td>
</tr>
<tr>
<td>$P_{IGEOMAT}(idim, style, ival, mtype, style) (ddim, w, h, d)$</td>
<td></td>
</tr>
<tr>
<td>$P_{IGEOROT}(idim, style, ival) (ddim, w, h, d)$</td>
<td></td>
</tr>
<tr>
<td>$P_{ICREATE}(idim, style) (ddim, w, h, d)$</td>
<td></td>
</tr>
<tr>
<td>$P_{IMEMCPY}(idim, style) (ddim, w, h, d)$</td>
<td></td>
</tr>
<tr>
<td>$P_{IDDELETE}(idim, style) (ddim, w, h, d)$</td>
<td></td>
</tr>
<tr>
<td>$P_{IMPORT}(idim, style) (ddim, w, h, d)$</td>
<td></td>
</tr>
<tr>
<td>$P_{EXPORT}(idim, style) (ddim, w, h, d)$</td>
<td></td>
</tr>
</tbody>
</table>

**Legend:**
- $ddim$: Dimensionality of data elements (usually: pixels)
- $w, h, d$: Extent of structure's domain in each of 3 dimensions
- $kw, kh, kd$: Extent of kernel's domain in each of 3 dimensions

Table 4B.1: Parameterized performance values.

The `ddim` performance value parameter relates to the dimensionality of the data elements (i.e., pixels) of the structure passed to a given instruction. All other performance value parameters relate to the sizes of the data structures passed to an instruction. Although the software architecture's benchmarking component incorporates all, by default the size and dimensionality parameters are taken together to form a single 'total size' parameter. It should be noted that the performance estimations presented in Section 4.5 are all based on benchmarking results obtained after this type of parameter merging.