Implementation and Evaluation of the IUA C++ Class Library on the Connection Machine[1]

Katja Daumüller and Charles C. Weems

Image Understanding Architecture Laboratory
Department of Computer Science
University of Massachusetts, Amherst MA 01003

Technical Report 94-40
May, 1994

Abstract

It is always of interest how portable a language is, especially when it has been designed for a specific machine and application. It is also of interest how well languages specific to a machine can be used for porting another language onto this machine. In this project, the Image Understanding Architecture (IUA) C++ Class Library (ICCL) was ported to the Connection Machine (CM) with PARIS. Design tradeoffs are described. The portability of the ICCL, and the suitability of PARIS for this task are investigated. We explored performance bottlenecks in the transport of the ICCL on three different machines by running several algorithms, written in the ICCL, and comparing execution times. Also, a new vision algorithm was developed, which was optimized for speed in the ICCL. The suitability of the ICCL for this task and design criteria are described below.

[1]This work was supported in part by Army Research Laboratory contract DAAL02-91-K-0047

Contents

1 Introduction

In the introduction, the vision architecture on which the ICCL runs is described together with the ICCL itself, the CM and PARIS.

1.1 The Image Understanding Architecture

The field of computer vision calls for particular kinds of computational power. For example vision may be divided into high-level vision, intermediate-level vision and low-level vision (pixel-level) and processing may occur on the three levels in parallel. Another special processing requirement in computer vision is to select particular subsets of pixels for various types of processing.
The heterogeneous Image Understanding Architecture (IUA) [6] represents a hardware implementation of these three levels of abstraction inherent in our view of computer vision. It consists of three different, tightly coupled parallel processors. These are the Content Addressable Array Parallel Processor (CAAPP) at the low level, the Intermediate Communication Associative Processor (ICAP) at the intermediate level, and the Symbolic Processing Array (SPA) at the high level. The CAAPP and ICAP are controlled by a dedicated Array Control Unit (ACU), that takes its directions from the SPA. In each of these layers, the processing elements are tuned to the computational granularity and algorithms required by that particular level of abstraction. As a result, the CAAPP is a SIMD array and the ICAP and SPA are respectively MIMD arrays with different computational granularity. With this architecture, the three levels of vision can be executed in parallel at their most natural computational granularity.

In the following, our main area of interest will be the CAAPP. The CAAPP is a square grid SIMD array of custom 1-bit serial processors, intended to perform low-level image processing tasks. Its size can vary from 64 x 64 to 512 x 512 processors. The first-generation IUA has size 64 x 64, and the processors are organized in an 8 x 8 mesh on each chip, with one board containing 4 chips. The system contains 16 boards.

There are four different ways of communicating among the CAAPP cells. One way is through global feedback and rebroadcast. A tree connection from the CAAPP cells to the controller enables fast feedback. Broadcast is done on the instruction bus by the controller. Another way is through the ICAP. This could possibly be used in case of large blocks of data, but is rarely chosen. The third way uses the nearest-neighborhood (S,E,W,N) mesh. The fourth way of communication distinguishes the CAAPP from previous architectures: Independent groups of processors are created that share a local associative feedback circuit. Multiple neighboring processors can write to the circuit at the same time. Switches on a mesh are used to reconfigure this **Coterie Network** into groups of neighboring processors. The mesh and Coterie Network continue across chip boundaries.

Also, noted above, region-dependent processing can be done on the Coterie Network in the CAAPP. For example, the maximum of a parameter in all regions of the image can be determined or a value can be broadcast within each region independently. On the other hand, non-region dependent operations like smoothing or enhancing execute faster on the nearest-neighbor mesh.
Among the components of the IUA, the focus of this paper is, as previously mentioned, the CAAPP. The next subsection explains how to program in the ICCL.

1.2 The IUA C++ Class Library

The ICCL is a C++ class library, used as a parallel language for vision applications, more explicitly, for low-level vision.
C++ offers several useful features for basic constructs, like the construction of a net of classes with their relation being e.g. friend or ancestor. Also, for each class, there exist protected or public members which can be functions or variables. The first feature is important for the predefined classes of the ICCL. One of the classes with no ancestor is e.g. 'plane', which means image. Planes are defined at the beginning of the program with their size, like a conventional variable. The plane type, however, is specified by a prefix to 'plane', e.g. IntPlane or CharPlane. IntPlane and CharPlane are classes themselves and descendents of the class Plane.
There are 7 predefined plane types in the ICCL, which are: BitPlane, CharPlane, UCharPlane, ShortPlane, UShortPlane, IntPlane and FloatPlane. The 'U' means unsigned, whereas ommitting the 'U' means signed. BitPlane is as big as CharPlane, but is assumed to contain only 0s or 1s. The second feature is convenient for retrieving parameters belonging to a particular class and for applying functions to classes. In the ICCL, there is usually a function for accessing a protected variable of a class.

In addition to the 'Plane' constructs, the ICCL also provides constructs dealing with the special feature of the CAAPP, the Coterie Network, and the control structures in the parallel language. The ICCL has predefined classes for both, 'Select' and 'Coterie' respectively. Both states get enabled by a call to a constructor function, and their state exists as long as the block where they were defined.
Except for the 'Plane' classes, the 'PlaneSize' class (which determines the size of a plane), the 'Coterie' and the 'Select' class, there are no other predefined classes. The user, however, is able to create new symbols by defining a new class with its operators and methods.

In C++, operators between classes are used like operators between variables in C, if they have been defined previously for each class. This feature is called overloading of operators and functions. The ICCL has the advantage of being easily transportable to other machines. The member and operator functions in C++ are instantiated with the various (Plane) types using a macro, which can conveniently be rewritten in another programming language.

In the next subsection, the CM is introduced and compared to the IUA.

1.3 The Connection Machine

It is best in general to test the portability and compare the execution of a specific language on a machine used relatively often and equipped with a basic architecture. We use the Connection Machine CM-2 for this task. The CM-2 is a SIMD-machine with a front end and back end. The front end is either a Sun-4 Workstation, a DEC VAX 8000 or a DEC 6300 minicomputer. The back end is a hypercube of processors with 4K, 8K, 16K, 32K or 64 K processors, where K is 1024. The processors are organized in processor chips with 16 processors each, which are linked in a boolean m-cube (hypercube). $m = log_2(N) - 4$, where N is the total number of processors. Communication within the processors is done in a permutation circuit. Communication from one processor to another uses the permutation circuit inside the chip and the hypercube links exclusively. Global broadcast is also done through the 'Instruction Broadcast Bus'. Special hardware is used for global feedback, to which all processors are connected.
In our case, a DEC 6400 is used as front end, the array contains 64 x 64 processors and the processor chips, each containing 16 processors are linked in an 8-cube. The CM-2 used for the presented experiments did not have a floating point accelerator.

Comparing the IUA and the CM-2 with respect to their connection system, the IUA has the additional Coterie Network, and the (rarely used) opportunity to connect to the higher level and use its connection system. The architecture of the hypercube connection system in the CM-2, however, is more powerful than the mesh system as it contains additional processors.

1.4 C-PARIS

In order to port a language to a machine, it is most speed-efficient to use the lowest-level language implemented on the machine. For the CM, this language is PARIS. PARIS is intended primarily as a base upon which to build the higher-level CM-languages, like *C**, CM Fortran and *Lisp. It is a low-level protocol, in which the user can write data parallel programs for the CM. There are three interfaces - C/PARIS, Lisp/PARIS and Fortran/PARIS. In our case, the ICCL has been written in C++, so it was convenient to use C/PARIS.
PARIS has arithmetic functions, functions relating to communication between processors and functions relating to communication between the front end and the back end. Control structures are implemented in so-called 'context'-setting functions. Before operations are executed, the 'vp-set' has to be specified, which is the ratio of virtual processors to physical processors in a user-defined geometry. Functions are differentiated by various options which are put into the function name. The bit-length of a variable is an operand

to the function. The variables can be allocated over an arbitrary number of bits, which is why they are called 'fields'. 'fields' are simply one or more contiguous bits that start at the same bit location in every processor. Fields can be allocated as a stack or as a heap.
A typical PARIS function might look like:

CM_[type]_function_[operators]_fieldlengthcount([result,] arg1, ... , argN, len1, ... , lenN);

The characters CM in the front of the functions means Connection Machine. 'Type' stands for unsigned, signed or float. 'Function' is usually a short form of the function name. 'Operators' is the number of field variables which contribute to the computation, e.g. there could be 3 variables, two inputs and an output variable, or there could be 2 variables, two input, and the result field is identical to one of the input fields. 'Fieldlengthcount' is the number of different lengths of the fields, appended with a L. So '2L' means that, e.g. the input and output fields have different lengths. Subsequently, 'res', 'arg1', ... , 'argN' are just fields, whose type is CM_field_id_t, and 'reslen', 'len1', ... , 'lenN' are int, short or char numbers. We refer to 'reslen', 'len1', ... , 'lenN' as the lengths of the corresponding result and argument fields 'res', 'arg1', ... , 'argN'.

The comparison of PARIS to the ICCL is straightforward: PARIS contains all the important features of the ICCL, except there do not exist communication functions which leave as much freedom of definition to the programmer as those of the Coterie Network. Considering the explicit specification of the variable type and bit-length in the function name in PARIS, the ICCL is slightly more high-level than PARIS.

2 Porting the ICCL to the CM through PARIS

In the following sections, the transport of the various operators and functions in the ICCL to PARIS is described in detail. Operators such as dyadic, monadic, set, and shift operators have their counterpart in the member functions of the classes, which are nearest-neighbor, index, i/o, route, select, coterie, broadcast and regionselectmin/max functions. The functions which operate on the Coterie network require some additional explanation because no equivalent PARIS operation is available for them. For the other functions and operators, the problem consists of matching the required types from ICCL and the types available in PARIS.

A common concern to both the ICCL and PARIS is the *activity* of the processors. Activity is itself represented in a BitPlane, where 1 means active and 0 inactive. Certain functions are sensitive to activity, i.e. only values in active processors can contribute to the function. In the ICCL, fewer functions are sensitive to activity than in PARIS. This stems from the fact that in PARIS, almost every operation can assign the result value to a parallel result field. Usually, activity plays a role when the right-hand side is assigned to the left-hand side. In the ICCL, apart from assignment, global operations and Coterie functions are sensitive to activity. Also in PARIS, the global operations are sensitive to activity (This did not imply an assignment, because the result variable is not parallel.) as is the corresponding operation to the Coterie operation. Therefore for most functions (the ICCL corresponds to a high percentage of functions in PARIS), the activity scheme in the ICCL and PARIS is equal.

2.1 The Dyadic Operators

There are three kinds of dyadic operators: dyadic operators between two field variables and dyadic operators between a field variable and a scalar with the scalar on the left side or on the right side respectively. We first investigate the dyadic operators on two fields.

2.1.1 Dyadic Operators between two Fields

The dyadic operators have a 'first argument', a 'second argument' and a result. In subsequent discussion, we will refer to the first argument as arg1 and the second argument as arg2. In this section, arg1 and arg2 are fields and have lengths len1 and len2, respectively.

This section can be divided again into the description of several operator types. There are the add, subtract, divide, multiply, mod, greater than, greater than or equal to, less than, less than or equal to, equal to, not equal to, and, or, xor, right-shift and left-shift operators.

For each ICCL function (or operator), there are equivalent PARIS functions. The only difficulties which complicate the application of PARIS functions are caused by the the 7 plane types in the ICCL and their combination in functions. The ICCL defines a table with possible combinations between the two operand types and the outcoming type, for example

IntPlane + ShortPlane = IntPlane or
ShortPlane + UShortPlane = ShortPlane.

Basically, the ICCL instantiates for every combination of types its own function. Each function has a small procedure with a macro call, which includes parameters for types. Every instantiation calls the function with different type parameters, and the macro is therefore called with different type parameters. Here, the preprocessor operator ## is used. In order to avoid costly if-statements, macro expansion is done according to different types by concatenating actual arguments (the types in the first level) to the macro call. At the very lowest level of the macro hierarchy there are then calls to PARIS, and at this point the types are used again: The types are either the input to a PARIS function or they represent part of the PARIS function name.

PARIS has three 'basic' types, which are signed, unsigned and float. These types appear in the function name. The other distinction in types can be made according to the length of the fields in PARIS, which are parameters in the function calls.
PARIS already takes care of most of the types used in the different functions. Unfortunately, PARIS is not consistent over several functions. The following features are coherent over some, but not all functions:

1. Different lengths are allowed for each of the 2 inputs and the output field.
2. The functions are defined on the types unsigned, signed and float operators, where the types of the arguments have to be the same.
3. The arguments for the operator and the result field can have all distinct field IDs.

Unfortunately, PARIS does not provide automatic type conversions, so the functions only take argument and result fields with the same type. Type conversions have to be done separately. This supports the assumption that PARIS is more low-level than the ICCL.

In the following, types are only 'unsigned', 'signed' and 'float'.

There are several levels from fulfilling none of the above requirements up to fulfilling all. For example, a 'low' level with few requirements fulfilled would be: all the same length fields on signed and float (not unsigned) with one of the argument fields being identical to the result field. For some functions, however, it is not necessary to offer the full range of possibilities, for example it is not very useful to apply boolean operators to float variables.

The philosophy of this project was to provide operations which can account for as many special cases as possible.
Several remedies can be applied to work around the unsupported argument combinations (each remedy according to the possible variation cases defined above):

1. *Operations between arguments of different lengths:*
   This case becomes applicable if operands of different lengths are not allowed. Then, (a) new field(s) can be allocated, and in case the allocated field(s) is (are) not the result field, the values of the smaller fields can be copied into larger fields with the CM_s/u/f_move_1L operation. Finally the operation can be executed. In case the smaller field is the result field, the resulting value has to be copied out of the allocated field into the smaller field again. Either only one field in the function or several fields can be allocated and assigned copied values.

2. *Types in functions are missing (most times signed or unsigned) or a function has to be applied to two arguments with different types (which PARIS does not account for)*:
   The remedy here is, to copy the 'less sophisticated type' into a field with the 'more sophisticated' type. Hereby, the order goes from unsigned via signed to float as the 'most sophisticated'. Fortunately, PARIS always provides a means of converting the 'less sophisticated' type to the 'more sophisticated' type.
   The transformation operations are the following:

   - From unsigned to signed, a field with length at least one more bit has to be allocated, and the copying is done with CM_u_move_2L.
     (see ONE_ALLOC, BOTH_RES_ALLOC and ONE_RES_ALLOC below)
   - The PARIS function CM_f_s_float_2_2L takes care of the conversion from signed to float.
     (see FLOAT_CONV below)
   - The PARIS function CM_f_u_float_2_2L takes care of the conversion from unsigned to float.
     (see FLOAT_CONV below)

3. *One argument field and the result field are identical*
   This is the easiest case. The goal with all the functions in ICCL is **not to destroy the argument fields** because the arguments could be the factors of a multiplication where the result is put into a temporary field and the argument reused. In any case, the arguments are members of classes, which should not be destroyed. The solution is to allocate a new field and copy the argument field, which would otherwise be overwritten by the result field, into this new field.

In order to take care of some mixture of the three cases, three macros have proven useful in the application with signed and unsigned (ALLOC macros) and float (FLOAT_CONV macro) cases and were applied in many dyadic operators:

- ONE_ALLOC

  This macro allocates a bigger field for either argument1 (call it arg1) or argument2 (call it arg2), and copies the field with the appropriate move-operation (CM_u/s/f_move_2L) into the bigger field. The operation is then executed with the bigger field. (case 1, 2 and 3 above)

- BOTH_RES_ALLOC

  Here, all three fields are allocated with specific lengths. The appropriate copy operations into the bigger fields take place with appropriate move operations. The operation is then executed with the bigger fields, and the result is copied back into the smaller field. (case 1, 2 and 3 above)

- ONE_RES_ALLOC

  Allocation of one argument field and the result field was designed, but proved to be redundant. (case 1, 2 and 3 above)

- FLOAT_CONV

  This function allocates a float field and copies and converts the first or second argument of the function from a signed or unsigned field into a float field. (case 2)

The above macros are used for many of the 14 dyadic functions. Subsequently, the implementations of all the dyadic functions are briefly explained:

- add-functions

  Fortunately, PARIS provides all the ideal cases mentioned previously for the add functions. The 2 arguments and the result field can all have different lengths, they are all disjoint and the add functions exist in signed, unsigned and float mode. Therefore only two macros were used for type conversion. One is ONE_ALLOC in the case of 'unsigned' add 'signed' or vice versa, and the other is FLOAT_CONV with the combination of a float with an unsigned or signed variable. BitPlanes can be both signed and unsigned due to their small occupancy of bits. The only type conversion with BitPlanes is from unsigned/signed to a float field variable.

- subtract

  The subtract functions in PARIS can be handled like the add functions except for the BitPlanes. Subtracting a BitPlane from something else or vice versa can have a negative outcome, so the ICCL prescribes that the result is signed, and ONE_ALLOC has to be applied to all combinations of a BitPlane and an unsigned plane with subtract.

- multiply

  Multiply can be implemented in the same way as the add function. Because of the costliness of the multiply function, however, a special program was implemented to handle the case where one of the factors is a BitPlane. In this case, an inquiry is made if the BitPlane contains a zero or one, and the appropriate steps are executed. This seemed to be faster than to involve all eight bits in the multiplication process.

- divide and mod

  The difficulty with the divide and modulo functions is that they are provided only for one length. For mod, there exists only the function CM_u/s/f_mod_2/3_1L, and for divide the functions CM_f_divide_2/3_1L and CM_u/s_truncate_2/3_1/3L.

  Another problem was that the CM_s_mod_2/3_1L function was not equivalent to that of the SUN. In case of the left and right argument having different signs, the second argument has to be subtracted from the result of the CM, in order to obtain the result in the SUN.

  The CM_*_mod_*_1L, CM_*_truncate_*_*L and CM_f_divide_*_1L functions are quite expensive. For this reason, the goal with these functions was to avoid using them as often as possible.
  First, the CM_*_mod_*_1L functions will be investigated in more detail.

Some new macros were developed which attempt to make the expensive CM_*_mod_*_1L functions cheaper. This, however, is only possible in special cases: Whenever the first argument is smaller than the second, the mod is equal to the size of the smaller argument. Presumably, this happens more often if the length of the first argument is shorter than the length of the second argument.
A macro of this kind is BIT_MOD_SOMETHING. This macro is invoked whenever a BitPlane mod 'someother'Plane is executed. The result is equal to the first argument whenever the second argument is not 1 (or -1 in case of signed Planes). If the second argument is actually 1 or -1, then the result is zero.
Another macro of this kind is called SMALLER_MOD_BIGGER. It is used whenever the first argument is shorter than the second argument (and therefore more likely to be smaller) and can be used for all combinations of signed and unsigned except the first argument being unsigned and the second being signed. The first step in this macro is to test in the larger (second) argument if the positions in excess of the first argument are all zero or all one. The test for all one is done only if the second argument is signed. If the first argument is signed, the kth position in the second argument (counting from the lowest-order bit position) is also included in this test, where k is the length of the first argument.
If the test is positive, we can be sure that nothing is changed if the excess positions of the second argument are cut off and the CM_*_mod_*_1L function is executed with the field length of the smaller argument.
If the test is negative, we know that the second argument is either bigger or equal to the first argument (in absolute values). If it is bigger, the result would be the first argument field. In order to be sure, let us consider the three cases in SMALLER_MOD_BIGGER:

- arg1 unsigned, arg2 unsigned:

  One of the positions in arg2 in excess to arg1 is not zero, therefore arg2 is actually bigger than arg1, and the result is equal to arg1. If the positions in arg2 in excess to arg1 are all zero, the function CM_u_mod_3_1L is used on len1.

- arg1 signed, arg2 signed:

  In this case, the positions of arg2 in excess of arg1 plus one are tested for all-zeros or all-ones. In case these positions are not uniform, there are two cases: Say, arg1 has 8 bits, and therefore the numbers go from -128 to 127. In the above mentioned leading positions of arg2, there is a discontinuity, so the numbers occur within the intervals -'big number' to -129 and 128 to 'big number'. Therefore, we have to test for the special case arg1 being -128 and arg2 being 128, because the mod would be zero in this case. In all the other cases where this specific test is negative, and the above mentioned leading

positions of arg2 are not uniform, the result is equal to arg1.
In case the leading positions of arg2 are uniform, the function CM_s_mod_3_1L is invoked on len1.

- arg1 signed, arg2 unsigned:

  Here, the positions of arg2 in excess of arg1 plus one are checked for uniformity of zeros. The leading position is checked, too, because if the test is positive, the *signed* CM_*_mod_*_1L function would be executed.
  In this case, the basic assumption len1 < len2 for using SMALLER_MOD_BIGGER can be relaxed to len1 $\leq$ len2.
  Taking the example from the previous item, in case of a negative test, the first argument could have the range -128 to 127, while the second argument's range is from 128 to a big number. Therefore, we have to test the special case with arg1 being -128, and arg2 being 128, too. If this special test is positive, the result is zero. Otherwise, if the leading positions plus 1 are not uniformly zero, the result is equal to arg1.
  In case the leading positions of arg2 are uniformly zero, the function CM_s_mod_3_1L is invoked on len1.

The last case (arg1 unsigned, arg2 signed) is not directly considered by SMALLER_MOD_BIGGER. As arg2 has to be at least 2 bits longer than arg1, this case is reduced to SMALLER_MOD_BIGGER with the three cases. A new field is allocates with length len1+1, and arg1 is copied in there (the new field should be signed). Now, if len1new < len2, SMALLER_MOD_BIGGER can be invoked again. This new macro for (arg1 unsigned, arg2 signed) is called UNSIGNED_MOD_BIGSIGN.

Here are the mod-macros in algorithmic description:

BIT_MOD_SOMETHING

1. test if arg2 is -1 or 1
2. if no, result is arg1
3. if yes, result is zero

SMALLER_MOD_BIGGER

1. test if some leading positions in arg2 are uniformly all-ones (signed) or all-zeros
2. if yes, apply CM_s/u_mod_3_1L to arg1 and arg2 with length len1

3. if no, do the following: test in some special cases if
   $arg1 = -2^{len1-1}$ and
   $arg2 = 2^{len1-1}$.
4. if yes, result is zero
5. if no, result is equal to arg1.

UNSIGNED_MOD_BIGSIGN

1. allocate new arg1 and res field with length len1+1
2. copy arg1 in new field
3. invoke SMALLER_MOD_BIGGER (if applicable) with new field
4. copy new result in original result field

These macros are not necessary for the transport to work, they are introduced for the sake of efficiency. In experiments, it was shown, that if arg1 is signed and arg2 is unsigned and of equal length, then SMALLER_MOD_BIGGER was roughly equal to BOTH_RES_ALLOC, where fields of length + 1 were allocated. In all other cases, where arg2 is much bigger than arg1, SMALLER_MOD_BIGGER was far superior to the allocation functions.

In the case of 'sometype'Plane mod BitPlane, with 'sometype' not being Bit, the result is assumed to be zero, because the condition in which some elements of arg2 are zero is not defined (the exception was not checked, as this would have involved a global inquiry).

Only signed and unsigned mod operations have been shown so far. The ICCL does not support float mod operations.

The div functions work basically the same way as the mod functions. The principle is the same for the signed and unsigned arguments.
In contrast to the mod functions, float arguments can be used with the div functions, too. For the divide with float, the PARIS function
CM_f_divide_2/3_1L is used, and Plane types which are not FloatPlane, but combined with a 'divide' with FloatPlane, are converted with FLOAT_CONV.
'sometype'Plane div BitPlane is assumed to be arg1, because a zero in the BitPlane is not defined (again, the exception was not checked, because this would have involved a global inquiry).
With the div function, in BIT_DIV_SOMETHING there is some more computation. Here are the macros:

BIT_DIV_SOMETHING

1. test if arg2 is -1 or 1
2. if no, result is zero
3. if yes, result is arg1 if arg2 is 1, and -arg1 if arg2 is -1

SMALLER_DIV_BIGGER

1. test if some leading positions in arg2 are uniformly all-ones (signed) or all-zeros
2. if yes, apply CM_s/u_truncate_3_1L to arg1 and arg2 with length len1
3. if no, do the following: test in some special cases if
   $arg1 = -2^{len1-1}$ and
   $arg2 = 2^{len1-1}$.
4. if yes, result is minus one
5. if no, result is zero.

UNSIGNED_DIV_BIGSIGN

1. allocate new arg1 field with length len1+1
2. copy arg1 in new field
3. invoke SMALLER_DIV_BIGGER (if applicable) with new field

At the end of UNSIGNED_DIV_BIGSIGN, in contrary to the modulo case, no new result field has to be allocated. The reason is that in SMALLER_DIV_BIGGER, the dividing function CM_u/s_truncate_2/3_1/3L is used, in which three different lengths are allowed to be used as the input fields.

- equality and inequality

Equality can be tested with the CM_u/s/f_eq_1/2L function. The result of the comparison is stored in a test flag. In CM_u/s/f_eq_1/2L, all different combinations of types and argument lengths demanded by the ICCL are provided, and therefore only type conversions have to be executed with ONE_ALLOC and FLOAT_CONV.

- less than (lt), less than or equal, greater than (gt) and greater than or equal

  These functions work basically the same way as the equality functions. A boolean flag is set in case the comparison is true. All the combinations demanded by the ICCL are provided, except the combinations of different types. Therefore, only ONE_ALLOC and FLOAT_CONV have to be used.

- and, or and xor

  Usually, the 'and', 'or' and 'xor' functions are only executed with arguments of the same length. If the arguments do not have the same length, however, it can be assumed that one argument is filled up with leading zeros. The ICCL defines the boolean 'and', 'or' and 'xor' functions on arguments with different lengths, too. In the ICCL, even signed variables can be combined with boolean operators. If two signed variables have different length, the ICCL defines that the smaller variable is copied into a bigger variable with sign extension. PARIS, however, provides the 'and', 'or' and 'xor' functions only with one-length (unsigned) arguments.
  For arguments with different lengths (call them arg1 and arg2 with len1 < len2), arg1 can be copied into the bigger result field. (In the ICCL, with 'and', 'or' and 'xor', the result field has length MAX(len1, len2).) With CM_u/s_move*, the sign extension is taken care of. If arg1 is unsigned, the boolean operation only has to be performed on len1 bits, because the leading bits are combined with zero. If arg1 is signed, either the boolean operation is performend on len2 with res and arg2, or the leading positions are tested for their sign bit. If the sign bit is zero, the operation can be performend on the smaller length len1, otherwise the operation is performed on len2 with res and arg2.
  In the implementation, for the signed case, no testing was done, and the boolean operation was performed on the whole field.

- Shift

  In the ICCL, right shift and left shift functions can be used. These functions are almost identically represented in PARIS through CM_u/s_shift_2_2L. The minor problem is that PARIS does not provide the *_3_2L. Therefore, a result field has to be allocated. Also, for the left-shift, arg2 has to be negated.

2.1.2 Dyadic Operators between a Scalar and a Field

In this section, dyadic operations between planes and scalar values are discussed.
PARIS covers this domain to some extent. It uses the expression 'constant' or 'const'

in its function names to indicate that one argument is a scalar value, and not a parallel variable.
For this implementation, however an easy solution was chosen. The implementation of the dyadic operators between fields had already been implemented, so it was a small step to allocate a parallel field for the scalar argument, copy the scalar value into the field and execute the dyadic field-to-field operators.
After some experimenting, this turned out not to be very time consuming: Allocating a field plus doing the copy into one field plus executing the 2-field add plus deallocating the field took 45 microseconds, while doing the add directly in the 'const' function of PARIS took 50 microseconds on an 8-bit field. We have not run empirical experiments in this area, however also the reported times show that the constant function takes longer than the 2-field function plus 'support' operations. For the logical 'and', for example, CM_logand_2_1L is reported to take 47 microseconds in 'real time' (which means back end time plus back end waiting time), while CM_logand_constant_2_1L takes 92 microseconds 'real time' on a 32 bit-field. The copy is reported to take only 33 microseconds. Altogether, 80 microseconds for the 2-field operation compare to 92 for the direct PARIS function. Other times are: CM_s_add_constant_3_1L takes 128 microseconds on a 32-bit field and CM_s_add_3_1L takes 150 microseconds on a 32-bit field with CM_s_move_constant_1L taking 38 microseconds. Here, 188 microseconds of the 2-field operation compare to 128 microseconds for the direct PARIS operation. Other timings differed accordingly. With this unpredictability, it was feasible to take the easier solution and implement the scalar-with-field operators using the field-with-field operators.

2.1.3 Other ICCL Functions

In this section, we describe ICCL functions, which are easy to implement on PARIS.

- The Monadic Functions

  In the ICCL, there are two monadic functions: the tilde operator and the negation operator. The tilde operator flips the bits in the argument, and the negation operator negates the value of the field.
  For both operations, there is one PARIS function which fulfills the purpose. For the tilde operator, there is CM_lognot_2_1L, and for the negation operator, there is CM_u/s/f_subfrom_constant_3_1L. The latter, however, was not intended for negation, and is time-consuming for doing just a 2-complement, (on a 32 bit field, it takes 57 microseconds) but an adequate PARIS function could not be found.

- The Set Function

The set functions implement the assignment operator.
In the ICCL, the control threads are implemented through the assignment operator, which means that the assignment only takes place in 'active' processors. So first of all, the processors which should be active have to be made so. Then, the argument is just copied with a 'move' function. For this purpose a 'move' function is defined, for each combination of types. For most combinations, this is CM_u/s/f_move_2L. Other combinations are CM_u/s_f_truncate_2_2L, which copies a float into an int, an (u)short or an (u)char by truncating the real part. CM_f_u/s_float_2_2L copies an int, (u)short or (u)char into a float.
Apart from the standard assignment operator, there is the operator which combines the assignment with a dyadic operation. First, the dyadic function with the arguments res and arg1 is combined, and the outcome is assigned to res. In this case, the field res is both the first argument and the result. This doubling of fields or *overlapping* of fields is not a problem with the dyadic functions used in this implementation.
For the assignment of a scalar to a field, basically the same steps have to be taken. For the simple assignment of a constant to a field, the functions
CM_u/s_move_constant_1L are used. The type conversions are made with a cast to the scalar. In case that the assignment operator is combined with a dyadic operator a field is allocated and the scalar is copied into it with one of the
CM_*_move_constant_1L functions. Then, the dyadic macro is invoked like with the field-with-field operator and the outcomes copied into the result field with a CM_*_move_*L function.

- The Nearest-Neighbor Functions

The nearest-neighbor functions on the CAAPP mesh are retrieved with PARIS by a call to CM_get_from_news_1L. The news net in PARIS is a user-defined multidimensional array. In the vision domain, it is two-dimensional. The south, north, west or east neighbors are retrieved by going 'down' or 'up' the news-axises.

- The Edge Function

The edge operator tests whether a pixel in the image lies on the image boundary or not. The south, north, east and west edges are tested, and the result is a boolean variable which indicates if the pixel lies on the particular edge. There is also a general edge function which indicates if a pixel lies on *any* of these edges.

The algorithm is implemented easily:

1. Get the x and y coordinates of the pixel with CM_my_news_coordinate_1L
2. Test for the particular condition of the desired edge, e.g. in the north edge, if the y coordinate is zero
3. Store the flag which was set in step 2 in the result field

- The Index Function

The index function returns the row and column index of a pixel in one variable. The column index is put into the low order 16 bits of the variable, and the row index is put into the high order 16 bits of the variable. This is accomplished by two calls to CM_my_news_coordinate_1L respectively on the low-order and high-order parts of the result field.

- Miscellaneous Functions

The miscellaneous functions include the resize, convert, any, count, increment and decrement, bit manipulation, minimum and maximum selection and the access functions.
In the following, we will discuss each of them in some detail.

1. Resize

   This function takes a plane with its x and y sizes in the x, and copies it into another plane with different x and y sizes. The upper left corner of the original plane is hereby copied into the upper left corner of the new plane. Empty space in the new plane is undefined, and in case the new plane is smaller, values are discarded.
   PARIS has the function CM_send_1L for this purpose. With this function, every pixel sends its value to a pixel with an 'address' in another coordinate system. In this case, the new coordinte system has different x and y sizes.

2. Convert

   'Convert' converts the value in a field to another type, and puts it into the new field. This is the same as the set functions from field to field, except the activity is not controlled. Therefore, the previously defined MOVE##type1##type2

macro which was used in the set function is used here.

3. Any

'Any' is a boolean function which tests if there are any ones in a BitPlane or not. It is implemented with the equivalent PARIS function CM_global_logior_bit.

4. Count

'Count' counts the number of pixels in a BitPlane which are ones. It is implemented with the equivalent PARIS function CM_global_count_bit.

5. Increment

'Increment' adds the value one to the value of a field. First, the activity of the pixels has to be set. Then, the PARIS function
CM_u/s/f_add_constant_2_1L is used.

6. Decrement

'Decrement' is the opposite to 'Increment'. The PARIS function
CM_u/s/f_subtract_constant_2_1L is used.

7. Bit Manipulation

There are three different bit manipulation operations. The first copies a 'string' of bits beginning at a given position of a source field to the result field at another given position. The second operation is the same, except the source is a scalar. The third operation retrieves the bit value of a given position in a field and returns a BitPlane. The first two operations are sensitive to activity.
All operations can be executed by applying some CM_u/s_move* operations to fields with predefined offsets. The offset for the first operation, for example is the given position of the bit-string in the input argument. The only problem in the implementation was that CM_u/u_move* does not operate on fields with length one. CM_move_reversed_1L, however, does.

8. Min/Max Selection

The min or max selection retrieves the minimum or maximum value in a plane. It is sensitive to activity, that is, it only retrieves the minimum or maximum

of active values in a plane. PARIS provides equivalent functions for this task, which are CM_global_s/u/f_min_1L and CM_global_s/u/f_max_1L.

9. Access

   Finally, the access function returns the logical or of the bits of the active values in the input plane. This is achieved by the PARIS function CM_global_logior_1L. In case that the CM front end is a VAX, the floating point format has to be converted from IEEE standard to VAX format. This can be accomplished with the PARIS function
   CM_f_ieee_to_vax_1L. (The conversion is not necessary the other way round, when a floating point number is given as an argument to a PARIS function.)

- The Route Function

With the route function, every pixel sends the value of the associated field at the source processor to the same field in a destination processor. All pixels can send values to themselves, or all pixels can send to the same receiver pixel. In the latter case, the value of the receiving pixel is not defined.
Besides the 'simple' routing function, there also exists routing functions in combination with dyadic operators. Combining an 'Add' with a 'Route', e.g. adds up the received values at the receiving processor. Other combining operators are 'and', 'or' and 'xor'.
In PARIS, there exist equivalent functions for 'Route', and 'Route' in combination with dyadic operators which were described already in the Resize part of the miscellaneous functions. The PARIS functions are CM_send_1L,
CM_send_with_u/s/f_add_1L and CM_send_with_logand/logior/logxor_1L.
For the combination with 'add', type conversions between the source and destination fields have to be taken care of with the CM_u/s_move* function. The type conversions are simplified by the convention of the ICCL, that the result type is IntType or FloatType. The algorithm is straightforward now:

1. Copy default value into result field.
2. Create send addresses with the given destination rows and columns.
3. If the route function is combined with an 'add', do type conversions
4. Execute the appropriate PARIS send operation.

- The I/O Functions

The purpose of the IO functions is to *read* an array from the front end into the processors in the back end or to *write* all the values from the back end into one piece of memory in the front end.
The functions CM_u/s/f_write_to_news_array_1L for the read and
CM_u/s/f_read_from_news_array_1L take care of this task.
In the read functions, the types of the source and destination plus their row and column lengths are given. Thus the types and lengths of the dimensions have to be adjusted. For the purpose of the read, another front end array is allocated with the desired axis lengths, to which the values are copied under type conversion and considering the potentially different axis lengths. The use of the several parameters of CM_u/s/f_write_to_news_array_1L was not investigated, but might have provided another solution.
In the write case, no constraints for the target type or target axis length are given, so a front end array is allocated with the same type and axis lengths as the back end field variable, and CM_u/s/f_read_from_news_array_1L is invoked.
The BitPlanes define a special case, because their representation in the back-end, which uses 8 bits of storage for one bit of information was compressed in the front-end representation.
Another IO routine for the ICCL is 'Access_Pe', which takes as input a specific coordinate of the plane, and returns the value of the plane at this location in the Plane. In contrast, the other 'Access' function returned the logical 'or' of the *active* values in the BitPlane. The PARIS function for the 'Access_Pe' function described here is CM_u/s_read_from_processor_1L, where different types have to be distinguished. For the floating point retrieval, the function CM_f_ieee_to_vax_1L has to be applied first.

- How the Paris variables are allocated and deallocated

For the allocation, first parameters of the variable like type and size are checked against the parameters of the variables already in the free list. Like in the sequential version, if the overall byte count of a variable is identical to that of a free piece, the geometry or the virtual processor set can be changed in order to use the variable from the free list. If no variable from the free list fits the specification, the new varibale, a parallel memory piece is allocated.
As with the sequential ICCL variables, PARIS variables are inserted into a linked 'free' list once they have been deallocated.

In experiments, the reuse of variables turned out to be slightly superior to the repeated allocation and deallocation of PARIS fields every time a variable is needed or disposed.

2.1.4 The Implementation of the Coterie Network

This section is dedicated to the implementation of the Coterie functions on the Connection Machine with PARIS.

The Coterie network is, as mentioned above, a reconfigurable mesh with broadcast. It partitions the image array into user-defined, continuous regions, inside which the Coterie operations are executed. The goal of the Coterie operations is to communicate data inside the regions, which is accomplished by one broadcast operation and the extraction of a maximum or a minimum out of the region. 'RegionBroadcast(sender)' broadcasts the bitwise logical *or* of the associated Plane at processors where sender has a one. 'RegionSelectMin/Max()' finds the minimum or maximum of values in the associated Plane and returns a BitPlane with ones exactly at the processors whose associated Plane value is equal to the minimum or maximum.

It is remarkable that all three functions work more or less with the same principle: They try to make all pixels in a Coterie region contain or test the same value. For RegionSelectMax/Min(), this is the maximum or minimum value in the Coterie region, and for RegionBroadcast(sender), this is the bitwise logical *or* of the plane values whose pixels had a one in the BitPlane sender. Therefore, we can use the same concept for all three functions, except that RegionSelectMax/Min() have to compare in addition the outcoming plane with the plane associated to the function and in case of equality put a one in the resulting BitPlane and a zero otherwise.

In PARIS, all the following functions communicate to some extent between the processors, so they could be of use for our task:

1. CM_get
2. CM_global
3. CM_multispread
4. CM_reduce
5. CM_scan
6. CM_send
7. CM_spread

In the following, a series of possibilities is investigated regarding the applicability to our task:

1. CM_get

   The function CM_get_from_news_1L is applicable to a wide range of problems. In our case, however, the sole application of this function can not solve the problem, because it does not partition the plane in any manner.
   CM_get_from_power_two_1L is a special case for exchanges between processors which are a power of two away from each other. This is not applicable here because the partitions can be random.

2. CM_global

   With CM_global_*, an operation is done globally on all processors, and the combined result is returned to the front end.
   As before, this does not allow partitions of the plane, so it does not solve our problem.

3. CM_multispread

   CM_multispread_* makes use of the different dimensions in the definiton of the field and uses CM_spread_* (see below). An axis-mask is given as input which defines the dimensions or axis' selected for the operation. On each axis, a CM_spread function is then executed.
   In our case, this is not useful. The spread-function provides a uniform output along one axis, which is not desired with the arbitrary Coterie partitions. The axis-mask is not of help either, because we only have 2 axes (mesh), and so the axis-mask 01 or 10 would result in a common spread along one axis, and the axis-mask 11 results in a spread on all elements in the plane. The axis-mask 11 would then lead to the same result as the CM_global* function.

4. CM_reduce

   CM_reduce_* is a variation of CM_spread_* (see below). Only one processor along each axis receives the result.

5. CM_scan

   The CM_scan_* function divides each axis into arbitrary parts. While scanning, it combines the accumulated value with the scanned value on the arbitrarily partitioned axis'. As combination operators, there exist several dyadic and boolean

operators in the CM_scan_with functions, among which are 'or', 'min' and 'max'. A BitPlane determines the partition of the axis, where there can be three modes. In the first, the scan *direction* does not effect the partition, in the second it does effect the partition, and in the third, there is no need for partition. The first mode works as following: The upward scan starts at the lowest-addressed processor. It scans (and performs the specific operation) until there is a one. Then, it disregards the result, and starts scanning again at the processor with the one. In the downward scan, the scan starts from anew *after* the one, not *with* the one. This way, the partitioning is fixed.
For our purposes, this one-dimensional partitioning scheme is useful, because it supports arbitrary partitioning, at least along one axis. With this function, the Coterie network could be simulated, however, it would involve some tedious programming to obtain a two-dimensional partitioning.

6. CM_send

In the CM_send* operation, every processor in the plane has a destination processor, where it sends its value to. If one processor receives several values, the CM_send_with_'operation'_* operation combines those values with a specific operation. The CM_send_to_news operation sends the value in a processor plane to a neighboring processor.
In the Coterie application, sending values from one processor to a particular second processor is not needed. In the ICCL, the Route function already takes care of that. The 'sending' part of the Coterie operations could also be taken care of by the spread function.

7. CM_spread

The CM_spread_with_'operation'* function combines pixels in a plane in a chosen dimension, that is, along a chosen axis. Basically, CM_spread* does a CM_scan_with_'operation'* in one direction across the whole axis, and copies the value at the highest position back to all positions of the axis.
This scheme does not support any partitioning, and so it is not used in the implementation of the Coterie operations.

The CM_scan* function in PARIS seems to be the only one which can partition the image array into arbitrarily shaped regions. It also exists for CM_scan_with_min, CM_scan_with_max and CM_scan_with_logior. The problem with this function is, that it partitions only along one axis.
Defining the Coterie Network in one dimension is not a problem; on every boundary,

a one has to be defined in the upward direction. The CM_scan computes the parallel prefix, so in order to have uniform values on one axis, the function CM_scan* has to be applied twice (forward and backward, which will be called 'double scan'). After one axis is uniform, the one-dimensional Coterie Network has to be extended to two dimensions. Therefore, the same scan for the one dimension, with the division of the axes into segments, depending on the bit-value 0 or 1, is done for the other dimension, too.
The functions used for the Coterie operations are CM_scan_with_logior_1L for RegionBroadcast(sender), and CM_scan_with_u/s_min/max_1L for RegionSelectMin() and RegionSelectMax(). In order to have unique segments, it would be convenient to use the function CM_scan_with_copy_1L for the backward scan. This function, however, is listed with 4538 microseconds on a 64 bit field versus 1343, 1923 and 1872 for the other three functions in the PARIS manual for version 5.2. Thus, an upward and a downward scan are performed with the same function. After one axis direction is uniform within its partitions, the Coterie operation could potentially be done already. Therefore, a test is performed if the other axis direction is completed as well. The test is only a nearest neighbor function which tests if all neighbors are equal except the PARIS partitioning where the coterie simulation changes from 0 to 1 in upward direction. If the test is unsuccessful, the erroneous axis direction (the axis direction *not* just worked on) is scanned with CM_scan_with_'operation'_*. As the test is only a nearest neighbor operation and therefore cheap, it is reasonable to perform a test after each double scan versus after a couple of double scans. This completes the scheme of the basic algorithm. The algorithm so far lacks one important detail: The CM_scan_* functions were not implemented with wraparound functions. Therefore, after every double scan on one axis direction, the pixels on the image edges have to be exchanged.

In summary, the basic algorithm for all Coterie operations has the following form:

```
for (1 == 1)
    do double scan in x direction
    swap values depending on operation in the ends of x-axes
    if (segments in y direction are uniform) break
    do double scan in y direction
    swap values depending on operation in the ends of y-axes
    if (segments in x direction are uniform) break
endfor
xy = axis not scanned by last scan
if (segments in xy directions are not uniform)
    do double scan in xy direction
return result of the scan
```

Now the special Coterie Networks, which operate only on one axis, are easy to implement: instead of two double scans in the for loop, there is only one double scan,

test and swap.

Although an adequate simulation of the Coterie Network has been found in the CM_scan_* functions, the worst case execution time is very high. In the case of a spiral, the infomation out of the innermost point has to be transmitted along the spiral to the outermost point. In real images, however, this scenario is rare. Nevertheless, the execution time for a Coterie operation on the CM is just slightly faster than on a SUN. See also section 6 for timings.

Now that all the functions of the ICCL are implemented, some deficiencies of to PARIS are discussed and improvements suggested, which would have helped the implementation somewhat.

3 Suggested Improvements to PARIS

In general, the implementation of the ICCL on PARIS, version 5.2 was relatively straightforward because the dyadic and the 'other ICCL functions' (see 3.1.2 and 3.1.3) have almost identical operations in Paris. The only major problem is that the Coterie Broadcast and RegionSelectMin/Max functions do not have their equivalent in PARIS. The Coterie functions operate on arbitrarily shaped, 2-dimensional regions. In PARIS, only one-dimensional axes can be divided arbitrarily, but nothing in the two-dimensional space. First, some general problems are mentioned which arose with the use of PARIS, and then specific problems, linked with the use of the Class Library are shown.

3.1 General Problems

- One of the minor general problems with PARIS is that there is no obvious rule when certain options are present; the decision is made based on the particular function. Many options are nonuniform among the functions.

  - For example in the dyadic operations, the always option (which does the operation in all processors regardless of the context-flag) appears only with the floating point option, but not with mod and rem or the comparison operations.

  - Also almost all dyadic functions have the option of the operands and the result having different lengths. However this is not the case with mod, rem and the boolean functions. Furthermore, the option of having three different fields for each of the two operands and the result does not exist with the shift function. There is only the option for having the first operand in the same field as the result.

  - Not only the dyadic functions are missing some options. In order to do branching in parallel, you have to use the context and with it eliminate certain processors. It would be much less complicated in some cases to use a CM_logior_context_with test() instead of combining CM_logand_context_with_test() with other functions.

  - On the one hand, options are missing, on the other hand, consistency is maintained where there is no need for. The function CM_u_lt_zero_1L(..) is a constant function with the value zero.

- Another drawback lies in the difficulty in learning the functions. The length of a function name is difficult to memorize, and sometimes, the same options appear in different short forms.

  - If a field is combined with a constant, then in the function name, the constant is referred to as 'const' or 'constant', like:
    CM_f_add_const_always_2_1L and CM_f_add_constant_3_1L. Probably in connection with 'always', 'const' is used, and otherwise 'constant'.

  - Also, it is not immediately clear whether to use a number indicating the number of operands before the number of different lengths in the fields or not. Comparison and boolean functions omit this number, which is confusing, e.g. CM_s_ge_1L(source1, source2, len), but the dyadic operations with only source, destination and length parameters contain it.
    (e.g. CM_s_add_2_1L(dest/source1, source2, len)).
    However besides that, there is no rule. For example consider the nearest neighbour functions CM_get_from_news_1L(dest, source, axis, directions, len) and CM_f_news_sub_2_1L(dest, source, axis, direction, s, e). The former does not have the number, the latter has.

  - Concerning the identification of the type, in most cases the type (u for unsigned, s for signed or f for float) is put after the initials CM_ .
    (like CM_s_ge_1L(..)) This rule is broken by the function CM_global_s_add_1L(..).

- The operation CM_set_field_alias_vp_set(..) is used to possibly assign a new vp-set and type to the plane. It would be convenient to have the same for the original field, not only an alias field.

Subsequently some problems in the respective kind of functions are described.

3.2 Specific Problems in the Implementation with the ICCL

1. Dyadic functions:

   (add, subtract, multiply, truncate, divide, mod, logand, logior, logxor, less [or equal] than, greater [or equal] than, equal to, not equal to)

   Dyadic operations, especially with operands of different lengths and different type are a problem.

- **Different Types**
  Combining two variables which have different types is not at all supported. For example, in order to multiply an unsigned short with a signed short, first the unsigned short has to be moved to a longer signed field.
- **Different Lengths**
  For example, combining a short with a char by mod (short mod char) requires us to allocate short fields and move the char field to a short field and also the result field to a short field (The result field is originally char in the ICCL). This problem is worse if one operand is a constant. In PARIS, the bits of a constant are only considered up to the length of the field with which they are combined. Assume an int constant has to be combined with a short field. Then an int field has to be allocated first and the short field has to be copied into the int field.
- **'Constant Operator Field' with non-commutative Operators**
  There exists no function in PARIS where the constant is on the left side of the operators. This is especially inefficient for non-commutative functions. A new field has to be allocated each time.

2. Bit Insertion Functions:

   The functions CM_s_move_* and CM_u_move_* are impractical insofar that their destination and source fields cannot have length one. Fields of the function CM_move_reverse_*_1L, however, can.

3. Access of a single Pixel in the News-Array and i/o of the News-Array:

   If the front end is a DEC-station, the floating point numbers have to be converted from IEEE floating-point format to vax floating-point format or vice versa. Maybe this could be done automatically with a compiler option.

4. Shift Function:

   In the shift functions (CM_u_s_shift_2_2L(..) and CM_s_s_shift_2_2L(..)) the only option is that the left operand field and the result field are identical. If you do not want to change the incoming parameters, the left operand has to be copied into the result field first. Other dyadic operations have this option.

5. Region Functions:

As previously mentioned, in the Class Library there exist the region operations Broadcast, RegionSelectMin/Max which operate on arbitrarily shaped two-dimensional regions. Broadcast broadcasts the logical 'or' of values whose points in the argument BitPlane are one. RegionSelectMin/Max puts out a BitPlane with ones in processors where the values of the points are the min/maximum of all selected points (points with a context-flag one) in the region. Arbitrarily shaped regions can be best approximated in PARIS by the function CM_scan_with_*, which operates on arbitrarily segmented axes (the function in its one-dimensional form was introduced in section 3).

Apart from the fact that CM_scan_with_* only operates in one dimension, solutions to the following problems would make programming much easier and not require major changes:

- In order to broadcast in just one dimension, two scans have to be made.
- The scan function does not allow wraparound. On the other hand, CM_get_from_news_1L(..) allows wraparound.

Overcoming these problems with the scan function made the algorithms for the Coterie functions quite complex.

6. Allocation of Fields:

For reuse of fields, the operation CM_set_field_alias_vp_set(..) can be used to assign a new vp-set and type to the field. There is no operation which changes the vp-set of the original field.
Aliases are created with CM_make_field_alias(field_id). An alias, however, can by itself only be created from the original field, not from another alias. This costs some minor effort.

7. Timing Issues:

The times of a specific part of the program can only be directly printed out. For further computations on these times (e.g. percentage) it would be convenient if the front end maintained a list with the respective times each part of the program has taken.

3.3 Summary and Suggestions

In summary the language PARIS is well suited for the implementation of the Class Library on the Connection Machine.

Problems exist with the region functions because there is no real equivalent for the Class Library functions in PARIS. Also there exists no wraparound in the planes when the PARIS function 'scan' was applied. Eliminating these two problems would increase the performance.

The lack of some specific options has to be worked around. In this specific implementation the addition of some options would have made the programming much easier and maybe improved the performance.

Also, the abundance of options which are expressed in the function names and their organization make it almost impossible to program without looking up almost every command. A clearer scheme would make it a little easier to program.

4 A Fast Line Extraction Algorithm

In this section, a parallel line extraction algorithm, implemented in the ICCL, is presented. The algorithm is used in the timing analysis of Section 6.
The goal with this line extraction algorithm is to run in real time with robust output. Applications for a real-time line extraction algorithm at UMass are the Unmanned Ground Vehicle (UGV) and the algorithms implemented on it.
The line extraction algorithm was developed, with the goal to be as fast as e.g. the FLF [4] and as accurate as e.g. the Burns algorithm [1]. It is based on the line extraction algorithm of Burns, but was modified significantly.
Developing the algorithm on the IUA, some design criteria in the algorithm were made considering the architecture of the IUA.

4.1 The Burns Algorithm

First, some basic principles of the Burns algorithm are introduced, which are used in the presented algorithm.

In line extraction algorithms prior to the Burns algorithm, long, partly low contrast lines were segmented because of their fluctuation or change in gradient magnitude. Therefore, Burns was looking for a criterion that does not take into account the gradient magnitude of the pixels contributing to the line. He found that the edge orientation carries important information about the set of pixels that participate in the intensity variation that underlies the straight line, particularly its spatial extent.
The *gradient orientation* is defined as the direction of maximum gray-level change. The important observation is that the gradient orientation does not vary much on the intensity surface associated with straight lines, while the gradient magnitude varies significantly.
As a result, Burns introduces *line-support-regions* which are ideally equal to the area of intensity variation that underlies the straight line.
The creation of the line-support-regions is the part of the Burns algorithm that is adopted in the presented algorithm. The Burns algorithm will be investigated in some detail:
Steps for the construction of line-support-regions are:

1. Compute the gradient orientation using an appropriate mask.
2. Apply a connected components algorithm using some partition of gradients.

Computation of the gradient orientation can be accomplished using several types of masks. The criteria Burns was looking for were symmetrical response to rotation of the line and sensitivity to detail. After experimenting with 1 x 2, 1 x 3, 2 x 2 and 3 x 3 masks, he found that the non-quadratic masks do not obtain symmetrical response,

and the bigger 3 x 3 mask did not respond to high-frequency 1-pixel wide regions or fine detail. Thus for Burns, the optimal choice was the 2 x 2 mask.
A partition of the gradient can be accomplished by either taking into account with some method the retrieved gradients or by applying a fixed-partition scheme.
Due to the problems in grouping with the first method, Burns has used the fixed-partition scheme, in which the 360 degree range of gradient directions is arbitrarily quantized into a small set of regular intervals, say eight 45 degree intervals or sixteen 22.5 degree intervals. A simple connected components algorithm is then used to form distinct region labels for groups of adjacent pixels with the same gradient orientation and create the support regions.

Now, for each support region, the Burns algorithm computes a line with a Plane Fitting Method. This method is not relevant to the new algorithm, so it will not be investigated in further detail. The resulting set of lines, however, does not represent the real lines for special situations. The reason lies back in the computation of the support region and the use of one fixed partition scheme for their creation. The first problem is the merging of spatially contiguous lines with similar orientation into one bucket. The second problem arises when the line orientation falls onto a partition boundary and as a result the support regions are fragmented. The merging of spatially contiguous lines tends to be reduced as the partition size gets smaller, but the fragmentation problem demands larger sizes.
The computation of a second support region representation, based on another fixed partition scheme, solves the second problem.The first problem can then be reduced using adequate partition parameters. This is accomplished by applying the *overlapping bucket scheme* and rotating the partition used by the second support region representation by half a partition size, see also Figure 1. With the second support region scheme, image lines which lie on boundaries of the first support region representation are now in the middle of the bucket and therefore adequately represented.
The line extraction is therefore done with the first support region representation as well as with the second support region representation, and two sets of lines are computed. In the end, the resulting redundant set of lines is removed by applying a voting scheme which uses the length of a line as a criterion for a vote.

4.2 A Fast Line Extraction Algorithm

In this section, the new line extraction algorithm is presented, which uses the support regions from the previous section.
The major steps of the algorithm are the following:

1. Compute the gradient in x and y direction with a 3x3 Prewitt mask.

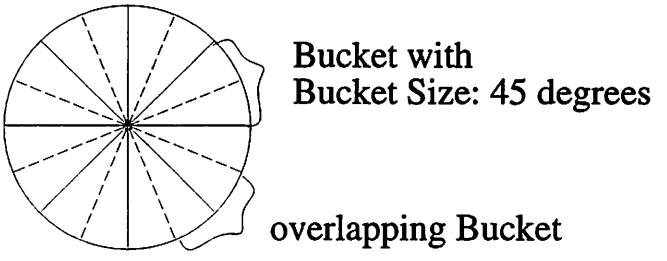

Figure 1: The Overlapping Bucket Scheme

2. Build gradient orientations and gradient magnitudes for the overlapping bucket scheme.
3. Create two support region representations.
4. Count the pixels in each region of the two support region representations.
5. Build up final support region representation.
6. Compute the master pixel of every region.
7. Split the support regions into n parts.
8. Select the m points of highest gradient magnitude in each of the sub-regions of the support regions.
9. Retrieve the position and orientation of the line with a least-squares fit method using the m * n points.
10. Compute the endpoints of the line.
11. Apply filtering values to select subsets of the lines.

The first step is common in most line extraction algorithms. The Burns algorithm and the FLF also perform the second and third step, which creates support regions. They too maintain the support regions as the basic data type for any further computation. One of the main differences of the presented algorithm to Burns algorithm is steps 4 and 5. In the presented algorithm, the two support region representations are combined *before* the lines are computed, whereas in Burns algorithm, they are combined *after* the lines are computed. This is a time-saving step, which the FLF also performs (yet differently).

In steps 7-9, each of the three algorithms computes the lines uniquely, which is another main difference. The last two steps are standard steps in line extraction algorithms.

Four essential input parameters control this algoritm:

- Number of Buckets
- Magnitude
- Length
- Contrast

The 'Number of Buckets' is used for the fixed partitioning scheme, the 'Magnitude' for eliminating pixels in the connected components algorithm, and 'Length' and 'Contrast' are filtering values.
Other parameters are set to a particular value before the execution of the algorithm, but theoretically can be changed, too. These are:

- Number of Subparts of the Support Region n
- Number of Points to select in each Subpart m

N and m are essential parameters in the computation of the line within a support region.

In the following, the steps with the input of the respective parameters will be described and explained in more detail:

1. *Compute the gradient in x and y direction with a 3x3 Prewitt mask.*
   The 3x3 mask is preferred over the 2x2 mask. The 2x2 mask is more sensitive to high-frequency ([4]), but the 3x3 mask recovers the orientation of a line better ([5]) and is more stable against blurred lines.

2. *Build gradient orientations and gradient magnitudes for the overlapping bucket scheme.*
   The gradient magnitude is the sum of squares of the gradient in x-direction and the gradient in y-direction. Pixels whose value of 'Magnitude' is below a threshold, are unselected and not considered for the next steps.
   One bucket representation is built by first dividing the 360 degree circle into 'Number of Buckets' pieces of equal size (see figure 1). In the end, the index for the

bucket is computed with
$index = round(\frac{gradient\_orientation \cdot 'Number of Buckets'}{360^0})$
The second bucket representation is built by rotating the first by half of the Bucket Size (with wraparound). The pixels are assigned the bucket indexes corresponding to their orientation values.

3. *Create two support region representations.*
   A connected components algorithm is run on the pixels, which connects neighboring pixels whose gradient orientation falls into the same index of step 2. Neighboring pixels are those adjacent on the north-south axis or on the east-west axis.
   With the overlapping bucket scheme, two bucket representations create two support region representations.

4. *Count the pixels in each region of the two support region representations.*
   In this step, first, master pixels for the support regions are computed. Then, each pixel sends a one with add-combine to its region's master pixel. The number received at the master pixel represents the count for the support region. In the end, each pixel has two counts for the two region representations available, which represent the number of pixels in the respective support regions that the pixel is part of. A boolean flag is set which indicates whether the first or second region representation has bigger count.

5. *Build up final region representation.*
   A connected components algorithm is run which connects pixels in the final region representation if

   (a) two adjacent pixels have the same flag.

   (b) two adjacent pixels have the same index in the support region representation determined by the boolean flag.

   The connected components from this algorithm construct the final support region. Note that with this scheme, each new support region is a sub-region of an original support region out of the selected representation.
   In the 'merging' of region representations just described, the **number of pixels** in a support region was used as a criterion that this support region will produce a 'better' line. 'Good' lines are usually long with oval support regions. The closest solution therefore is to use the criterion 'length' instead of 'number of pixels', however this criterion is not available at this stage of the algorithm.
   Most of the time, regions with many outliers and irregular shape (many outlier paths) are low contrast regions. This is taken care of in step 1, where the pixels are filtered out for 'Magnitude'. Other cases in which the length is a significantly better criterion than the number of pixels are not very common. Usually, the peak in the number of pixels of a support region is reached with its 'ideal' gradient orientation, which is perpendicular to the orientation causing the line. Only with

line-crossings does the greatest number of pixels and the longest length result from different gradient orientations.

6. *Compute the master pixel of every region.*
   Now, the master pixels of the new support region representation scheme are found.

7. *Split the support regions into n parts.*
   In this section, the support regions which were retrieved in step 5, are split into parts. We will describe the split for three parts, because that is the minimum number of parts required to avoid common mismatches.
   Splitting a support region into n parts is done perpendicular to its orientation, that is, if, for example, the support region has the shape of an oval, the major axis is split into three parts with the splitting lines being perpendicular to the major axis. For this task, the orientation of one arbitrary pixel is taken, and through every point (x_i, y_i) in the support region, a line is drawn with slope m. The intersection of this line with the y-axis at y-value t_i can be computed in the following way:

   $y_i = x_i \cdot m + t_i$ is transformed into
   $t_i = y_i - x_i \cdot m$

   Naturally, some points (x_i, y_i) have the same t_i.
   After that, the two t_i most distant from each other are obtained and are called t_1 and t_2. See Figure 2 for the construction of the split.
   The lines with slope m through t_1 and t_2 are the most extreme lines bounding the region that are orthogonal to the selected orientation.
   Usually, these extrema points are unique. But as the figure shows, it is possible to have multiple extrema for a region. In that case, we choose arbitrary extreme points on t_1 and t_2 and call them p_1 and p_4. Next, the line segment from p_1 to p_4 is divided into three parts of equal length, with points p_2 and p_3 as splitting points. The final splitting lines l_1 and l_2 pass through p_2 and p_3 with slope m, so that they divide the region into three parts along its length.

   The orientation of the split was taken from an arbitrary pixel in the support region. It varies at most one bucket size from the true perpendicular orientation of the line, because the chosen pixel belongs to the same region, and therefore its gradient orientation to the same bucket.
   The next step will explain why the minimum number of divisions is three.

8. *Select the m points of highest gradient magnitude in each of the sub-regions of the support regions.*
   In this step, m points of highest gradient magnitude are selected in each sub-region. In experiments, assigning 3 to m gave the best precision-time tradeoff.

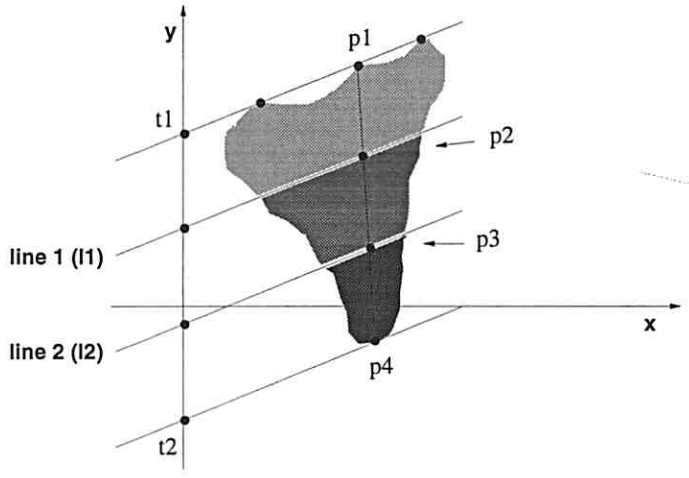

Figure 2: Split of support region (shaded region) into three parts. The three new subregions are indicated by the light, medium and dark shaded regions

Then the x and y coordinates of the selected points are stored in the following way: The pixel from the previous step, which determines the orientation of the split, is taken, and if its gradient in the x direction is greater than its gradient in the y direction, the x_i and y_i coordinates of the selected points in the region are exchanged. After the computation of the line parameters in the next step, the regions with reversed x and y coordinates adjust them accordingly.

Now the reasons for the split of the support region into three parts can be explained: The spatial distribution of the selected points is crucial for the direction and location of the extracted line. Thus, the points which contribute to the computation of the parameters have to be scattered *along* the line as much as possible. With two sub-regions, however, points could still be clustered around the splitting line and not contribute at all to the real orientation with their clustered location. Therefore, the minimum number of splits for the region is three.

9. *Retrieve the position and orientation of the line with a least-squares fit method using the m * n points.*

In this step, the line parameters are computed from the points of highest magnitude in the whole region. The line parameters are represented in slope m and intersection with y-axis, i.e. intercept c. In the ideal line match, for every point i, the following equation is true:

$$y_i = m \cdot x_i + c$$

The error function consists of the differences of the y-coordinates of the points to the virtual y-coordinate of the line at the x-coordinates of the points. (If the coordinates have been exchanged in the previous step, exchange the x's and y's in the preceding sentence.) This error is squared for every point, and the sum of the squares is the error function.

$$E^2 = \sum_i (y_i - m \cdot x_i - c)^2$$

The error function is then derived with respect to m and c and both derivations are set to zero. After the derivation we have:

$$c = \frac{\sum(y_i - m \cdot x_i)}{n}$$
$$m = \frac{\sum(x_i \cdot y_i) - c \cdot \sum(x_i)}{\sum x_i^2}$$

Now it is an easy step to resolve for c and m and take into account, if necessary, the earlier exchange of the x and y coordinates. The case where m overflows happens only in regions with exchanged coordinates and is prevented before computing the quotient which will be m: The boundary value of $m = 150$ was chosen, beyond which the line is considered vertical, which is a separate case.

This step also comprises the boolean labeling of points, if they are 'on the line' or not. For this purpose, the 'distance' from each point in the region to the line was computed. Hereby if the slope of the line is $|m| \leq 1$, 'distance' means the distance of each point to the intercept with the line along the y-axis. If $|m| > 1$, 'distance' means the distance of each point to the intercept with the line along the x-axis. In the case where m exceeds 150 and therefore was considered vertical, 'distance' is the difference of the x-coordinate of each point with the x-coordinate of the vertical line.
The points are labelled true if 'distance' is ≤ 0.5.

The least squares method was chosen because it has a linear solution, and is therefore easy to implement and fast on the IUA. On the IUA, trigonometric functions and iterative arithmetic floating point functions are time consuming. With the least squares fit, these functions can be avoided. Furthermore, the least squares fit produces optimal results, as long as no outliers are present. In case of outliers, to some extent, the other points will prevent the line from distortion.

10. *Compute the endpoints of the line.*
In this step, the coordinates of the endpoints of the lines are computed.

The endpoints are the points on the line ('linepoints') furthest away from each other. In case of a positive slope, this is done by taking the maximum of the row index and the maximum of the column index which construct the first endpoint, and the minima of the row and column index, which construct the second endpoint. In case of a negative slope, the maximum of the row index and minimum of the column index construct the first endpoint, and the opposite ones the second. As the search for the maximum and minimum is time consuming, the row and column indexes are summarized into one unique index, in which the maximum and minimum are searched for. In case of the negative slope, first the column index is subtracted from the number of columns in the image. This subtraction is reversed after the search.

11. *Apply filtering values to select subsets of the lines.*
Lines which do not exceed a minimal length or contrast, are filtered out. The contrast is computed by averaging the gradient magnitude of the linepoints. Previously, in step 2, pixels were already filtered out which by themselves did not exceed a certain 'Magnitude'.

5 Comparison of Execution Times on the SUN, the CM and the IUA for Different Algorithms written in the ICCL

In this section, execution times for algorithms written in the ICCL are directly compared between the SUN, the CM and the IUA. The machines that were used were the SUN Sparcstation 2, the CM-2a and the IUA simulator, version 1.4. On these machines three vision algorithms were run, which are the low-level portion of the DARPA benchmark, the line finder presented in section 5, and a dense depth map retrieval algorithm.

First, the execution times of some member functions in the ICCL are reported on the three machines. This gives insight into potential bottlenecks in the execution of the above algorithms on the three machines. Then, each algorithm is split into its basic parts, and the execution times of the basic parts are compared, given the execution times of the basic functions of the ICCL.

The IUA simulator is simulating a real machine, and therefore timings vary slightly for the same instruction regarding which instructions were executed previously. In the experiments, timings of the basic functions are an average of several executions in sequence. This tests only one setting (the previous instruction is exactly the same), but reduces the variation in timings, depending on where the sequence starts, to one percent. For the same program, however, timing results are reproducable.
The IUA simulator only provides back end times, which are compared to the PARIS back end times.

For the CM and the SUN, the PARIS timing functions and the *getrusage* facility were used. In contrast to the IUA simulator times, the timings vary with the execution of the same program.
The accuracy percentages in the paragraph below were obtained by examining a few runs on the respective machines.
The PARIS timing functions have microsecond precision. Their execution times vary by at most 15 percent (measuring the difference of two times divided by the smaller time, times one hundred) for the back end, but usually (around 90 percent of the time) the variation is below 5 percent. For the idle back end time, however, times usually vary significantly; in the experiments they varied by up to 1200 percent!
The accuracy of *getrusage* is one hundredth of a second, which is quite coarse. The stability of the results, however, is as high as in PARIS, because the results vary by at most 15 percent with low execution times. With high execution times, the stability is usually below 5 percent and was not observed to be above 10 percent.

For these reasons, multiple executions on the CM and SUN are averaged.

For the times reported, no virtualization of the parallel machines was necessary. The test images were all 64 x 64 pixels in size, which is equivalent to the size of the CM and the first generation prototype of the IUA.

5.1 Execution Times of Basic Functions of the ICCL

Timings were taken of the ICCL functions shown below, which are a selection of functions in the ICCL.

float, int Index
float, int West
float, int plus
float, int multiply
float, int multiply with Select
int function abs
float function abs
Any
float, int SelectMax
int Route
float, int Route within Coterie
int RoutePlus
time to build Coterie
float, int Broadcast
float, int RegionBroadcast with Select
int RegionSelectMin
time to build Select1
time to build Coterie1
Any1
int Broadcast1
int RegionSelectMin1
int Route1_region
int Route1_transl_20
int Route1_transl_20_wrap
int RoutePlus1_region
int RoutePlus1_transl_20
int RoutePlus1_transl_20_wrap

The above functions represent some selection of the arithmetic functions, the non-Coterie communication functions and the Coterie functions, which were used frequently in the presented algorithms. Other functions, which were not used very often in the

presented algorithms, like Input
Output functions, Display functions, the Resize function, etc., were omitted.

All functions were tested with the equality operator. As the equality operator involves activity, there is a slight timing difference if the activity is declared everywhere or if it has been set explicitely.
The functions *multiply* and *RegionBroadcast*, however, are not directly affected by activity, and they are included in the experiment 'with[in] Select' only to verify that setting the activity causes no side effects. The same is true for the Route functions tested within the Coterie Network. (*float, int* Route within Coterie)

The functions *RegionSelectMin* and *RoutePlus* were only tested on integer planes because at test time they were not implemented in the micro code on the IUA simulator.

The functions with suffix one were run only on the SUN and the IUA simulator. Prior to testing them on the CM, access was lost due to decommissioning of the machine. Because of this, *Route[Plus]*, *RegionBroadcast*, *RegionSelectMin* and *Any* were only run on a simple underlying pattern which should provide a lower time bound, whereas *Route[Plus]1*, *RegionBroadcast1*, *RegionSelectMin1* and Any1 were run on underlying patterns which are closer to situations in real images.

For the *Route[Plus]* function, each processor routes just to itself.
Route[Plus]1_transl_20 is a one-to-one permutation, where the destination pixel is 20 pixels translated in x-direction as well as in y-direction with respect to the sending pixel. For the destination pixels which lie beyond the borders of the image array, the functions without the suffix *wrap* discard the sending value, and the functions with the suffix *wrap* take the destination coordinates modulo the imagesize. For the *Route[Plus]1_region* functions, the image is split into regions, which correspond to equal integer values in the gradient orientation of the image. For every region, a master pixel is chosen, which serves in the *Route[Plus]1_region* functions as destination pixel for the pixels in the region. Because of the applied scaling factor the regions can contain as many as 200 pixels, where in most of these regions the subregions are linked together only by a one-pixel wide bridge. This property of the regions makes a *Route* time-taking.

The Coterie functions without suffix one (*RegionBroadcast* and *RegionSelectMin*) operate on regions of size one. The Coterie functions with suffix one, however, operate on the same regions which were described above for the *Route[Plus]1_region* function.

In the *Any* function, the first non-zero element is found at the top left corner in the image and the function is returned, whereas the *Any1* function finds the first non-zero element in the center pixel of the image.
The *RegionBroadcast within Select* function measures the execution time of the function

plus the assigment to the result plane, with the activity set. The *mulitply with Select* function, however, measures the time for the multiply and assignment plus the setting of the activity.
The term *time to build Select1* indicates the time to set the activity, where one processor is active at each row. This is also the activity which is set for the functions with suffix *with[in] Select.* The term *time to build Coterie[1]* indicates the time to set up the Coterie network, where the Coterie networks correspond to the suffix of the above introduced Coterie functions.

The basic functions give some first insight into the time ranges on the three machines. For functions that are missing above, similar functions indicate the likely time range. For example minus is expected to be roughly as fast as plus, and *RegionSelectMax* as fast as *RegionSelectMin.*

5.1.1 Dependence of the ICCL Operations on Input Data.

The three machines show a different behaviour for variation in the input data or 'underlying structure'. 'Underlying structure' means the definition or activity or the Coterie Network. Here are some known facts about data dependency, which also stem from knowledge of the implementation code for the SUN and the CM:

- The IUA Simulator

  On this machine, only the timings of the Route operations depend heavily on the input data. For Coterie operations, partitioning of the image in spiral-like form versus one-pixel regions should also result in different numbers of cycles. A fixed time, however, is reported for the Coterie operations that is an average case estimate. Therefore, for a spiral-like pattern on a large image the timing estimate is low, but for a 64 x 64 array the assumed time is sufficient even for a spiral.

- The SUN

  Here, only the construction of the Coterie network is significantly sensitive to the underlying structure. The reason for this is that by constructing different region representations, pixels could get questioned if they are already in some region a different number of times. Also, with regions extending over several rows and row-major storage, pieces of the image might not be readily accessible in the cache.
  Apart from this sensitivity to the underlying structure, timings do not vary much because all functions scan the image in a loop one to several times, which infers a

common basic cost already.
Small savings can occur if the body of this loop is omitted under some conditions. One example is the setup of the activity, which queries for active pixels in e.g. *RegionSelectMin/Max*. Another example is *RegionBroadcast*, where only values for broadcast are considered when the corresponding value in the input plane is one.

- The CM

  The execution times vary slightly depending on the definition of activity (implicitly by *Everywhere active* or explicitely by *Select* statements), because with the explicit definition, a logical 'and' has to be executed in the assignment operation or the function sensitive to activity.
  The dependence on the underlying structure and input data is greatest in the Coterie functions. The Coterie functions are implemented as a loop with a global combining function to test for completeness. As the values in the regions have to be equal in the end (for both *RegionBroadcast* and *RegionSelectMin/Max*), both input data and underlying structure influence the timing.

Below, the results of the experiments with the basic functions on the three machines are presented. The functions are listed in the order of their execution time on the respective machine.

5.1.2 The SUN

In Table 1, the times are shown for the basic ICCL functions executed on the SUN. Because of the coarse granularity of the timing function (one hundreth of a second), the experiments were repeated several times, and their average is reported.

The times in Table 1 correspond to the execution of the basic functions on the sequential version of the ICCL.
In the table, all the functions with times below 10 milliseconds are below the precision of the timing function. This means that their result is the average between runs with the same input where the timing function outputs zero time and runs where it outputs 10 milliseconds time. As the experiments were done with 400 runs of the program, even the functions with times below 10 milliseconds differed only by at most 15 percent, and the majority of the functions differed by at most 5 percent. An exception hereby are the *Any* and *Any1* functions, which are barely measurable.

The timings for the ICCL functions on the SUN are divided into distinct groups: The most expensive functions are the Coterie functions, including their setup, followed by the *Route* functions. All other functions are less expensive than the Coterie and *Route*

functions.

ICCL function	time in millisec.	percent of total time
Any	0.070	0.03 %
Any1	0.660	-
time to build Select1	1.803	-
int Index	2.890	1.10 %
float Index	3.670	1.39 %
float multiply	3.825	1.45 %
int plus	3.839	1.46 %
float plus	4.080	1.55 %
int SelectMax	5.270	2.00 %
float SelectMax	6.590	2.50 %
int West	6.770	2.57 %
int Route1_transl_20	6.845	-
float multiply with Select	6.870	2.61 %
float West	7.050	2.68 %
int multiply	7.105	2.70 %
int RoutePlus1_transl_20	8.740	-
int Route within Cot.	9.250	3.51 %
int multiply with Select	9.305	3.53 %
int Route	9.370	3.56 %
int Route1_transl_20_wrap	9.700	-
float Route within Cot.	9.766	3.71 %
int Route1_region	10.455	-
int function abs	10.610	4.03 %
float function abs	11.210	4.26 %
int RoutePlus	12.795	4.86 %
int RoutePlus1_transl_20_wrap	13.145	-
int RoutePlus within Cot.	13.549	5.14 %
int RoutePlus1_region	13.730	-
int RegionBroadcast1	15.296	-
int RegionBroadcast	16.735	6.35 %
int RegionSelectMin1	16.875	-
int RegionSelectMin	16.965	6.44 %
float RegionBroadcast	17.229	6.54 %
float Reg.Broad. within Sel.	17.680	6.71 %
int Reg.Broad. within Sel.	17.795	6.75 %
time to build Coterie	33.155	12.59 %
time to build Coterie1	38.784	-
Total	263.443	100.000 %

Table 1: ICCL functions executed on the SUN.

The most striking observation in Table 1 is that the time required to build the Coterie Network is a major bottleneck in an ICCL program. It is the slowest function,

and two times slower than the second slowest function. The reason is that in the sequential version, an array with the master pixels for every pixel in the image has to be built first.
On the SUN, the time to build this array depends on the characteristics of the resulting Coterie Network. The setup for the one-pixel sized region takes 17 percent less time than for larger regions. The reason is that for larger regions, the pixels have to test all their neighbors for possible expansion of the region, and in the scan of the whole array, some connections are tested several times. Also, the non-cache accesses are more frequent if the regions extend accross large pieces of memory.
In the Coterie functions themselves, there is a small difference in timing between functions with and without suffix one. One reason for this is the frequent determination of the sender coordinates in the one-pixel sized regions (all pixels are senders and broadcast values to themselves).

Among the Coterie functions *RegionBroadcast* and *RegionSelectMin*, there is only a small difference in timing, which points to their similar structure.
The Coterie functions are four times slower than e.g. *int plus* (and 12 times including the setup time), and are the most costly functions on the SUN.

Most of the *Route* functions take 9-10 milliseconds, and the *RoutePlus* function takes 12-13 milliseconds. Here, it makes little difference if the routing scheme is route-to-self, a translation, or routing to a master pixel within a region. There is one exception, however, both with *Route* and with *RoutePlus*: *Route[Plus]1_transl_20* takes about one third less than the other functions. The explanation is that in the loop over all pixels of the image, there is a query if the destination pixel is within image sizes. If it is not, the loop body is skipped. In this case, for every row, 20 out of 64 values are discarded, which explains the time reduction by roughly one third.
According to the low time difference between
Route[Plus] and *Route[Plus]1_transl_20*, if no values are discarded the timings of the *Route[Plus]* functions are insensitive to the distance between sending and receiving pixel. The low time difference between *Route[Plus]* and *Route[Plus]1_regions* shows that the *Route[Plus]* functions are insensitive to the number of collisions as well. This makes the *Route[Plus]* functions relatively insensitive to input data on the SUN, if no values are discarded.
The time difference between the *Route* and the *RoutePlus* function stems from the *plus* operation which is needed for the *RoutePlus* function, and which would fill approximately just the time difference (4 milliseconds).

Out of the remaining functions, only the *multiply* function has a characteristicly high difference between float and int.
The *Index* functions are on the low end, next are the addition functions, then the *SelectMax* functions, and finally the nearest neighbor functions. The *int multiply* function is

slower than the nearest neighbor function, whereas the float function is faster than the addition function.
The least expensive function is the *Any* function, whose timing was almost not measurable. In the *Any* function, an BitPlane is scanned, until a one is found. The difference between *Any* and *Any*1 stems from the different positions of the first one in the BitPlane. In *Any*, the bit is found earlier.

The difference between the *multiply* within the Select scope (including the setup) and the *multiply* outside the Select scope is distinct. The timing difference is probably due to the setup time for the activity, measured in 'time to build Select1', which is 1.8 milliseconds. The small remaining time difference can stem from the assignment operation, which performs a logical 'and' with the current activity. The same can be the reason for the small time difference between the *RegionBroadcast* functions inside and outside the Select scope.
There is, however, no timing difference for the *int Route* function inside and outside the Coterie network.

In summary, algorithms on the SUN are merely dependent on the input data or underlying structure, and therefore are predictable. The only significant difference between integer and floating point occurs in the multiply function.

5.1.3 The CM

In Table 2, the results for the CM are shown. The times are ordered by back end times because of the instability of the 'excess front end' times. 'Excess front end' times are idle back end times.
As with the SUN, the Coterie functions take longest, but are then followed by some *multiply* and the *abs* functions, succeeded by the *Route[Plus]* functions, and then followed by the other functions.

The table shows that like on the SUN, the Coterie functions operating on one-pixel sized regions are slowest on the CM. As the Coterie functions operate over a while loop which tests for uniformity within the regions, the actual Coterie timings on regions created by real images could increase by factors. This would make the Coterie functions a bottleneck of the program. The setup time for the Coterie Network, however, is low in comparison to the Coterie functions. It consists merely of a bit insertion into the setup variable, plus time on the front end.
Another interesting fact with the Coterie functions is that, unlike other functions, the time spent in the back end outweighs the time spent only in the front end. This could be due to the length of the PARIS program in the respective function macro. In other functions, the time for the PARIS program is short because a comparable PARIS - ICCL

function is available. Therefore, a smaller percentage of time is spent in the initialization of the routine.

ICCL function	back e. time	front e. time	total time	perc.ti. ba.end	perc.ti. fr.end	perc.tot. time
Any	20.9	206.8	227.7	0.048 %	0.395 %	0.238 %
time to build Cot.	344.1	1413.2	1757.3	0.796 %	2.700 %	1.838 %
int plus	409.2	1807.1	2216.4	0.946 %	3.453 %	2.319 %
int Index	450.0	1362.2	1812.2	1.040 %	2.603 %	1.896 %
int West	527.8	1471.3	1999.1	1.220 %	2.811 %	2.091 %
float West	545.7	1449.9	1995.5	1.261 %	2.770 %	2.088 %
int SelectMax	560.9	1701.1	2262.1	1.297 %	3.250 %	2.366 %
float SelectMax	605.6	1738.5	2344.1	1.400 %	3.322 %	2.452 %
float Index	673.3	1478.8	2152.1	1.556 %	2.826 %	2.251 %
int Route	847.6	1714.4	2562.1	1.959 %	3.276 %	2.680 %
int Route within Cot.	857.6	1827.1	2684.7	1.983 %	3.491 %	2.808 %
float plus	889.6	1482.2	2371.8	2.056 %	2.832 %	2.481 %
float Route within Cot.	905.2	1811.9	2717.1	2.093 %	3.462 %	2.842 %
int RoutePlus	934.0	1662.6	2596.6	2.159 %	3.177 %	2.716 %
int RoutePlus within Cot	954.0	1857.3	2811.3	2.205 %	3.549 %	2.941 %
float multiply	1051.5	1529.7	2581.2	2.431 %	2.923 %	2.700 %
int function abs	1254.2	4640.3	5894.5	2.899 %	8.867 %	6.166 %
float multiply with Select	1386.8	2634.5	4021.2	3.206 %	5.034 %	4.207 %
int multiply	1427.7	1730.1	3157.8	3.300 %	3.306 %	3.303 %
int multiply with Select	1755.8	2604.0	4359.7	4.059 %	4.976 %	4.561 %
float function abs	1814.4	4481.2	6295.6	4.194 %	8.563 %	6.586 %
int RegionBroadcast	4818.9	2195.9	7014.8	11.140 %	4.196 %	7.338 %
int Reg.Bro.within Sel.	4851.6	2147.0	6998.6	11.215 %	4.102 %	7.321 %
float Reg.Bro.within Sel.	4862.5	2057.6	6920.1	11.241 %	3.932 %	7.239 %
float RegionBroadcast	4914.2	2908.4	7822.5	11.360 %	5.557 %	8.183 %
int RegionSelectMin	5595.5	2421.5	8017.0	12.935 %	4.627 %	8.387 %
total	43258.7	52334.5	95593.3	100.00 %	100.00 %	100.00 %

Table 2: ICCL functions executed on the CM, times in microseconds.

The *int Route* function is about five times less time consuming than the *int RegionBroadcast* function on the back end. This makes the time difference between these two communication functions big, although only their basic patterns or underlying structure were tested. For more complicated patterns the hypercube architecture of the CM can benefit the timings of the *Route[Plus]* functions versus for example the mesh architecture. Note that the difference between *Route* and *RoutePlus* is not as big as on the SUN and does not reflect the timing of the *int plus* operation.

On the CM, both integer and floating point multiplication are much slower than the *int plus* operation, in contrast to the time relations of these functions on the SUN. The multiplication functions both are even slower than the *RoutePlus* functions.

In general, for some functions, there is a considerable difference between the floating point and the integer operations. On the back end, the floating point addition takes double the time than the integer addition, and the *float multiply* is slower in comparison to the *int multiply* than on the SUN (*float multiply* takes two thirds of *int multiply* versus less than one half on the SUN). Also, the *abs* function shows some difference for floating point and integer plane arguments. The higher differences between operations on integer and floating point planes, in comparison to the SUN, could stem from the lack of a floating point accelerator in the machine used for these experiments.
A remarkable observation is the timings of the *Index* function. The *Index* function returns the position of every pixel in the image array, and the only access to the actual input plane (class object) is to get its image size. Nevertheless, the difference between floating point and integer planes as class objects is high, which is not explicable with this reason. This difference was also recorded on the SUN.

Concerning the activity setup (called 'Select' here), the excess front end time of the *float multiply* function almost doubles when the activity setup (Select) is added and its time is taken inside the Select scope. Also, the excess front end time of the *abs* functions, where activity has to be set first, are almost 4 times as high as the back end times. The timings for the *int RegionBroadcast* function indicate that the assignment statement inside the Select setup involve almost no overhead. This suggests that the mere setup of the activity involves some fair amount of excess front end time plus some back end time.

In summary, the Coterie functions are the most time-taking functions on the CM. This is true for one-pixel sized regions, and the times can increase by multiple factors as other regions are used. *Route* times are also expected to increase with different input patterns, but the hypercube architecture may improve the timings.

The excess front end times in the CM are usually 2-3 times larger than the back end times, except with the Coterie functions. For the faster functions (upper part in Table 2), the excess front end times smooth the effect of the back end times, however they also represent to some extend the magnitude of the back end times.

5.1.4 The IUA

On the IUA simulator, only back end times were available, which can be compared to the back end times of the CM.

As mentioned in Section 6.1.1, on the IUA simulator Coterie functions are not dependent on input data or the underlying structure. This is confirmed by the small difference between *RegionSelectMin* and *RegionSelectMin1* as well as by the small difference between *RegionBroadcast* and *RegionBroadcast1*. In Table 3, the Coterie functions are on the slow end of the basic functions, with the exception of the *Route[Plus]1* functions on a non-trivial pattern. The Coterie functions are, however, not much slower than the integer multiply function, which needs one fourth less time. The setup for the Coterie network takes very little time, which brings the Coterie operations in the range of a multiply function.

The *Route[Plus]* functions exhibit a significant dependency between the routing scheme and the timings. Similar to the timings on the SUN, the functions without wraparound are faster than the functions with wraparound. The *Route[Plus]* functions are also faster than the *Route* functions, in most cases. In contrary to the SUN, however, the distance between sending and receiving pixels and the number of collisions have a distinct influence on the timings of *Route[Plus]*.
On the IUA simulator, a permutation with a translation of 20 in x and y direction and executed with wraparound takes 100 times longer than the route-to-self scheme, and 30 times longer if executed without wraparound. Here, the overhead for computation of the new coordinates for the wraparound as well as the computation of the right tile for each pixel are responsible for the difference. In our experiments, only one tile was used, however, the special case for one tile has not been implemented yet on the IUA simulator. It is also notable that for the permutation with wraparound, the time difference between *Route* and *RoutePlus* is proportionally much bigger than the time difference of *Route* and *RoutePlus* without wraparound. The addition in *RoutePlus* delays the processing of packets, so that some packets could 'fall behind' due to the additional time spent on the wraparound.
Collisions are caused intentionally in *Route[Plus]_region* by assigning the same destination pixel to several source pixels. In *Route[Plus]_region*, regions of up to 200 pixels with extensions in x and y direction of 20 and 30 pixels and a shape containing many 'outlier' paths, route to their master pixel. As the timings for *Route[Plus]_region* show, this slows down the functions significantly. The reason is that in the *RoutePlus* function, collisions between packets with the same destination result are resolved by combining the packets (i.e. executing the *plus*) right at the place of the collision. Therefore, congestion is reduced. This scheme is not yet implemented in the *Route* functions. In summary, the *Route[Plus]* functions are highly dependent on the underlying pattern, and timings can vary by orders of magnitude.

ICCL function	time in microsec.	percent of total time
Any1	2.8	-
Any	3.0	0.08 %
time to build Coterie	3.5	0.09 %
time to build Coterie1	3.5	-
time to build Select1	3.5	-
int Index	9.3	0.24 %
float Index	9.3	0.24 %
int plus	12.6	0.32 %
float West	12.8	0.33 %
int West	13.9	0.36 %
int function abs	17.6	0.45 %
float function abs	27.8	0.71 %
float SelectMax	65.3	1.67 %
float Route within Cot.	67.8	1.73 %
int Route within Cot.	67.8	1.73 %
int Route	67.8	1.73 %
int SelectMax	72.7	1.86 %
int RoutePlus	86.9	2.22 %
int RoutePlus within Cot.	87.0	2.22 %
float plus	148.6	3.80 %
float multiply	207.7	5.31 %
float multiply with select	211.9	5.41 %
int multiply	304.1	7.77 %
int multiply with select	314.9	8.05 %
int RegionSelectMin	354.5	9.06 %
int RegionSelectMin1	356.2	-
int Reg.Broad. within Sel.	434.6	11.10 %
float Reg.Broad. within Sel.	434.7	11.11 %
int RegionBroadcast	437.9	11.19 %
int RegionBroadcast1	438.0	-
float RegionBroadcast	439.6	11.23 %
int Route1_transl_20	2020.3	-
int RoutePlus1_transl_20	2038.6	-
int Route1_transl_20_wrap	5892.8	-
int RoutePlus1_transl_20_wrap	7536.1	-
int RoutePlus1_region	15086.7	-
int Route1_region	726300.0	-
Total	3913.6	100.00 %

Table 3: ICCL functions executed on the IUA simulator.

Concerning the other functions, the ranking is similar to the one for the back end times on the CM. One exception is the time for the setup of the activity (*Select*). The time difference of *int multiply* to *int multiply with Select* is negligible on the IUA simulator, and also the *time to build Select1*. The difference between *int RegionBroadcast*

and *int RegionBroadcast within Select* is negligible as well, where the function within the *Select* scope is even slightly faster. Another difference to the CM is the different ranking of the *abs* functions, which use the *Select* setup. On the CM, they were slower than the *Route[Plus]* functions, while they are 2.5 times faster than the *Route[Plus]* functions on the IUA simulator. These findings indicate that the setup of the *Select* environment does not take much time on the IUA simulator, and also that the assignment operation is influenced minimally by the *Select* environment.

The addition and multiplication functions have the same ranking as the CM back end times. On the IUA simulator, however, the difference between the *float plus* operation and the *int plus* operation is bigger.

As on the CM, the nearest neighbor functions are on the fast end of Table 3 and roughly equal to the *int plus* function. On the SUN, however, the *int plus* function is double as fast as the nearest neighbor functions. Therefore, given these timings the parallel architectures seem to favour nearest neighbor functions.

The *Any* function does not exhibit different timings from the *Any1* function. The *Any[1]* functions do not depend on the input data because the global query happens in parallel.

In summary, on the IUA simulator the Coterie functions do not depend on the input function or underlying structure, but the Route functions do to a large extend. There is a big difference between *float plus* and *int plus*.

5.1.5 Comparison of the Basic Functions on the SUN, CM and IUA

For the purpose of a comparison of the the basic functions timings on the three machines, Table 4 shows the proportion of SUN-CM total timings, and the proportion of CM-IUA back end timings. In order to improve coherency, the functions with suffix one are not included in Table 4 or in comparisons of all three machines in this subsection, because they have no counterpart on the CM.

Looking at the times in Tables 1-3 (excluding the functions with suffix one), the second slowest and second fastest functions differ by the factor 18 on the SUN, by the factor 4 and 15 on the CM (total time and back end time) and by the factor 124 on the IUA simulator. This supports the statement that the IUA depends more on the choice of functions than the SUN or CM. The fact that the timings on the CM front end are less diverse (but usually much greater) than the timings on the back end supports the

statement that the excess front end time smooths the effect of the back end time.

Program Part	SUN / CM	CM / IUA (back end times)
int Index	1.59	48.39
float Index	1.71	72.40
int West	3.39	37.97
float West	3.53	42.63
int plus	1.73	32.48
float plus	1.72	5.99
int multiply	2.25	4.69
float multiply	1.48	5.06
int multiply with select	2.13	5.57
float multiply with select	1.70	6.54
int function abs	1.80	71.26
float function abs	1.78	65.27
Any	0.31	6.97
int SelectMax	2.33	7.72
float SelectMax	2.81	9.27
int Route	3.66	12.50
int Route within Cot.	3.44	12.65
float Route within Cot.	3.59	13.35
int RoutePlus	4.93	10.75
int RoutePlus within Cot.	4.82	10.97
time to build Coterie	18.86	98.33
int RegionBroadcast	2.39	11.00
int Reg.Broad. within Sel.	2.54	11.16
float RegionBroadcast	2.20	11.18
float Reg.Broad. within Sel.	2.55	11.19
int RegionSelectMin	2.12	15.78
Total	2.76	11.05

Table 4: Proportions of Execution Times.

Comparing the execution times of the three machines, all basic functions except the *Any* function are faster on the CM than on the SUN (total time), and all basic functions are faster on the IUA than on the CM (comparing CM-back end times). More precisely, the CM is 0.3 times (for *Any*) up to 19 times (for the *time to build Coterie*) faster than the SUN, and the IUA is 5 times (for *int multiply*) up to 98 times (for the time for build Coterie) faster than the CM. Thus, the time difference from the IUA to the CM seems to be greater than the difference from the SUN to the CM. This is supported by higher numbers in the second column of Table 4 than in the first column. Exceptions to this rule are for the first column the *time to build Coterie*, which decreases dramatically on the CM. Also, the nearest neighbor functions and basic *Route[Plus]* functions have high speedups for the CM. Functions with low speedup are the *plus* and *float multiply*

functions, as well as the *abs* functions. *Any* is slower on the CM than on the SUN. In case different routing patterns are used for *Route[Plus]* or different Coterie Networks for *RegionBroadcast/SelectMin/Max*, ratios between the SUN and CM change in favor of the SUN. On the CM, Coterie timings could change by multitudes, whereas the *time to build Coterie* only changed by 17 percent on the SUN, when a different Coterie Network was used. The *Route[Plus]* operations on the SUN were insensitive to changes in the routing pattern, too.

In the second column of Table 4, the *Index*, *West*, *int plus*, *abs* functions and the *time to build Coterie* have speedups above 30 from the CM to the IUA simulator. The functions which have a speedup below 6.54 on the IUA are the *float plus* and *multiply* functions. The speedup of the global functions *Any* and *SelectMax* is above 6.97, but still below 10. In case of different Coterie Networks for the Coterie functions, the ratios could highly rise in favour of the IUA simulator, as the CM is highly sensitive to different Coterie Networks and the IUA simulator is not. For the *Route[Plus]* functions, different route patterns seem to trigger high times on the IUA simulator, which are comparable to times on the SUN (see Table 3). In real applications, however, timings will not deteriorate as much, as will be shown with the line extraction algorithm.

In the following, some basic functions are compared individually on the three machines.

- Beginning with the **Coterie** operations, they are on the slow end for all the machines (except for *Route1* on the IUA). The CM is 2.5 times (and 7 times including the setup time on the SUN) faster than the SUN, and the IUA is 11 times faster than the CM. For different Coterie Networks, however, the times on the CM could change dramatically. Comparing the two Coterie functions *RegionBroadcast* and *RegionSelectMin*, *RegionBroadcast* is slower on the IUA, roughly equal on the SUN, and faster on the CM than *RegionSelectMin*. The time differences between these two functions is highest on the IUA, where it amounts to 20 percent.

  Another function which is ranked differently on the machines is the **time to build the Coterie Network**. While among the fastest functions on the CM and IUA, it is slowest on the SUN. The reason for the fast setup on the CM and IUA is that only a Plane has to be defined, and on the IUA electrical switches have to be set. In the SUN, however, an array with the master pixel is defined for each element.

- The **West** function seems to fit better on parallel machines, as both speedups from the SUN to the CM and from the CM to the IUA are higher than the speedups of the total times of the basic functions.

- The **Any** function is the fastest instruction on all three machines. Although the time on the SUN is dependent on input it is not expected to exceed double the time of *Any1* (in *Any1*, the first detected 'one' is in the center of the image).

- Arithmetic functions, like **plus or multiply** are part of almost every program, which makes them an important factor in timings.
  As noted earlier, the *plus* functions have a pretty low speedup from the SUN to the CM, as well as the *float multiply* operation, which could be due to the floating point accelerator in the SUN. It is remarkable, however, that the parallel CM is not that much faster than the SUN. For the *multiply* and *float plus* functions the speedup from the CM to the IUA is also below the speedup of the total times for the basic functions. The speedup from the CM to the IUA simulator for the *int plus* function, however, is very big.

- For the **Route** functions with the route-to-self scheme, the speedups are in the range of speedups of the total times for the basic functions. In case of different route patterns, however, the speedups can change dramatically to the favor of the SUN (see Tables 1 and 3). Timings for the *Route1_region* function, however, are close to the worst case scenario. For timings of *Route* functions in real vision applications, see the timings for the line extraction algorithm in section 6.3.

- For the **Select** setup, comparisons are made indirectly. In the SUN/CM comparison, the *multiply* shows no significantly different speedups from the *multiply with Select*, where the speedup for the *multiply* function is low. The speedup of the *abs* functions, which use the *Select* setup, is below the speedup of the total times for the basic functions. This supports the hypothesis that the speedup from the SUN to the CM for the setup of *Select* is below the speedup of the total times for the basic functions.
  In the CM/IUA comparison, the *abs* functions exhibit the second and third biggest speedup. Also, the speedup of the *multiply with Select* functions is slightly higher than the one of the *multiply* functions, which, however, does not make any suggestions regarding the magnitude of the speeup. The high speedup for the *abs* functions points to a speedup above average for the *Select* setup from the CM to the IUA simulator.

In summary, these timings provide a first insight in the speedups of timings between the machines. There are, however, too many unknown influences which could change the timings of a program completely. For example, it is not known how much slower the Coterie functions are on the CM with different Coterie Networks, or it is not known how the *Route* functions behave on the CM, or to what extend the *Route[Plus]_region* timings can be generalized to real applications on the IUA simulator.

What is known so far is that the IUA simulator is the fastest machine, and seems to be especially suited for the Coterie functions, which are in the range of the multiply function. The CM displays the highest speedup to the SUN in the nearest neighbor operations, and the lack of a floating point accelerator in the testing machine is evident. The SUN has the least amount of dependency on input data, and has a floating point accelerator. Nevertheless, given the proportions in Table 4, it is the slowest machine.

The following algorithms are a combination of the above presented functions. We can see which functions are used for which particular vision techniques, and how the techniques influence the total timing.

5.2 The Low-Level Portion of the DARPA Benchmark

The benchmark ([7])intends to be a framework for testing machine performance on a variety of vision operations and algorithms, which require communication and control across algorithms and representations. The goal was not to perform a given task in the best way possible, but to perform typical vision tasks in a typical sequence, in order to test an architecture.
In its full form, the benchmark requires low and intermediate level processing, which can also be separated into bottom-up (data-directed) and top-down (knowledge or model directed) processing. In the ICCL, which operates on the SIMD CAAPP chip, only the low-level portion of the benchmark is executed. The low-level portion developed for the ICCL includes no floating point operations.
The whole benchmark task involves recognizing a 2 1/2 D 'mobile' sculpture composed of rectangles, given images from intensity and range sensors. The low-level part is more specific: It involves recognizing rectangles, given the intensity image.

Subsequently, the sequence of steps of the algorithm is described without detail. Recognition of the rectangles of the mobile sculpture is performed by first splitting the intensity image into regions of equal intensity and marking the boundary pixels of the region. This presumes that only pixels of equal intensity can be grouped into rectangles. Given the connected boundary pixels, the k-curvature is computed and corner pixels are extracted and connected. Among the corner pixels, only corners above a certain threshold are selected. Then, the convex hull is computed over the selected corner chain, and the new corners are marked and connected. Angles between lines connecting the new corners are recomputed, and those regions are selected which have three connected right angle corners. The middle corner in the chain of the three connected right angle corners is the master corner. Finally, the selected regions are assumed to contain rectangles, and attributes are computed and reported, using the master corner.
It can be seen that with the above construction, occluded rectangles can also be recognized, as long as three right angles are still present. The convex hull mechanism overlooks the occluded sides.

In the following, a more detailed step-by-step description is given, along with some off-hand information about the frequency of operations in the respective program parts:

1. *Read in image*
   The time for reading a parallel variable is reported.

2. *Connected Components*
   The parallel connected components algorithm is performed which links pixels of equal intensity.
   The following operations occur four times: test for equality, inversion and the nearest neighbor (retrieving the north, south, east or west value), test if on the border of an image and insert bit operation.

3. *Select Masters*
   A unique pixel is selected as master of each region in this operation.
   This involves one *RegionSelectMax* operation, and one *Index* operation.

4. *Label Region*
   Here, the row and column index of the master pixel is broadcast in each region.
   This involves one *Index* operation and one *RegionBroadcast* operation.

5. *Trace Edges*
   The boundaries of the regions are traced. A pointer exists for every pixel on the boundary, which points to the successor pixel on the boundary (that is, the south, north, east or west).
   This involves mainly nearest neighbor operations, 'and', 'or', 'not' operations, bit-access operations and activity operations (mainly Select).
   Coterie operations are not used (a coterie network is built for display purposes however).
   One while loop makes sure the whole boundary is traced in every region. The termination is tested with *Count*. This makes the trace time input-dependent.

6. *K-Curvature*
   Corners in the region boundary are found here.
   This is accomplished by moving K pixels along the boundary links from the previous step in one direction, and K pixels in the other direction. With the help of a lookup table, approximate angles between the boundary segments on the two sides of the pixel in question are computed. Then, a one-dimensional Gaussian smoothing function is computed on the K * 2 - 1 wide mask, constructed out of the two segments from above, which operates on the approximate angles of every pixel. The Gaussian smoothing value is the sum of the product of the angles with a lookup table value.

Finally, the first derivative of the Gaussian smoothing value is computed over the connected pixels on the region boundary and zero crossings are obtained by comparing the sign of the difference.
In the end, the corners are selected at places with zero-crossings where the smoothing value and the approximate angle exceed a particular threshold.

The operations involved in this part of the benchmark do not differ much from the previous part: Again, mostly dyadic operators like subtraction, multiplication, addition, and, or and test for greater, less and equality are used, and negation and inversion. Apart from that, again the nearest neighbor operations and activitiy operators are used frequently. Except for nearest neighbor operations, no other communication function is used.
A typical feature of this routine is that everything is done four times (in the four directions), which can be quite time consuming.

7. *Convex Hull*
Here, the 'list' of corners from the previous part is used, and the intention is to link those corners together to build a convex hull.
First, the boundary corners are ranked depending on the quotient of the difference of their row and column index to the row and column index of the master pixel of the region. Then, links between the corners are created depending on their rank. For every corner, a determinant is computed, which takes into account the row and column indexes of its two connected corners. Corners with negative determinant are removed from the set of corners, and the links are rebuilt completely depending on the ranking of the remaining corners. Then, the determinant is computed again, and so forth. This continues until all the determinants of the corners turn positive, and no corner is deleted from the corner set.

The main features in this routine are the frequent use of activity functions, the Coterie functions and the front-back end communication functions (*Any* and *Count*). Test functions, (except for *Any* and *Count*) such as greater or less, are used less frequently.
The while loop is the main part of the convex hull. Its termination depends on the outcome of the *Any* and *Count* functions and makes the timings of the convex hull part heavily dependent on the input.

8. *Compute Rectangle Attributes*
The goal here is to compute the attributes of all selected rectangles in the image. The input to this timing section is the corner pixels along with the links between them which were obtained in the 'Convex Hull' routine.
First, the angle of a corner is computed. This time (which is different from the angle in the K-Curvature part), the angle between the two *straight line* segments is

computed which connect the corner to both its linked corners. Angles within some tolerance are selected as right angles. After that, the criterion is applied that some object, possibly occluded, can only be defined as a rectangle if it has at least three connected corners that are right angles. Regions with less than three right angles are discarded. For the remaining valid regions, a while loop determines for every corner if its connected corners are right angles. The master corner is the corner which has a right angle itself and where both its connected corners have right angles as well. Only regions with such a master corner are assumed to build a rectangle. The computation of the rectangle attributes then uses the connected corners of the master corner: The midpoint is computed by taking the average of the pair of connected corners. The length of the minor and major axis are the length of the line segments drawn from the master corner to either of its connected corners. Also, the orientation of the major axis with respect to the X axis is computed.

As in the previous part, there are many Coterie and activity operations. The nearest neighbor operations are not present here, instead there are many *Index* operations. For the test of a right angle corner being connected to two right angle corners as well, a while loop is necessary. This makes the timing dependent on the input again.
A major time-consuming occurrence is the division in the integer square root function, which is used to compute the lengths of the major and minor axis and occurs 36 times. Concerning trigonometric operations, the acos at the end of this timing section is not relevant, because it uses a look-up table.

9. *Access Rectangle Attributes*
   For all the regions where rectangles were found in the above described sense, the master pixel outputs the attributes, which are: coordinates of the midpoint, the angle of the major axis with the X-axis, the major and minor axis length, and the intensity value of the master pixel.
   A while loop goes through all the valid rectangles. The *Access* operation communicates between the back and front end, *SelectMax* searches for the maximum in the whole active image and *Index* accesses the row and column index of the master pixel.

An important factor in the evaluation of the benchmark timings is that it does not use float variables and Route. It uses many short variables, like short and char, for which our retrieved ratios from Table 4 are not really valid, but could give some insights.

5.2.1 The Timings

The SUN

Program Part	time in sec.	percent of total time
Read in Image	0.29	0.72 %
Connected Components	0.06	0.15 %
Select Masters	0.09	0.22 %
Label Regions	0.1	0.25 %
Trace Edges	27.65	68.54 %
K Curvature	7.34	18.20 %
Convex Hull	1.73	4.29 %
Compute Rectangle Attributes	2.98	7.39 %
Access Rectangle Attributes	0.1	0.25 %
Total	40.34	100.00 %

Table 5: Low level portions of benchmark, executed on the **SUN**.

The CM

Program Part	back e. time	front e. time	total time	perc. ba.e.ti.	perc. fr.e.ti.	perc.tot. time
Read in Image	13.89	125.31	139.20	0.10 %	0.59 %	0.39 %
Connected Components	8.36	31.41	39.77	0.06 %	0.15 %	0.11 %
Select Masters	148.17	15.45	163.62	1.04 %	0.07 %	0.46 %
Label Regions	128.11	13.79	141.90	0.90 %	0.06 %	0.40 %
Trace Edges	3840.68	13658.43	17499.11	26.94 %	63.98 %	49.15 %
K Curvature	1216.97	3903.04	5120.01	8.54 %	18.28 %	14.38 %
Convex Hull	6482.47	1705.94	8188.41	45.47 %	8.00 %	23.00 %
Compute Rect. Attrib.	2407.06	1841.02	4248.08	16.88 %	8.62 %	11.93 %
Access Rect. Attrib.	11.25	54.32	65.57	0.08 %	0.25 %	0.18 %
Total	14256.96	21348.71	35605.66	100.00 %	100.00 %	100.00 %

Table 6: Low level portions of benchmark, executed on the **CM**, times in millisec.

The IUA

Program Part	time in millisec.	percent of total time
Read in Image	0.012	0.015 %
Connected Components	0.082	0.098 %
Select Masters	0.475	0.566 %
Label Regions	0.566	0.674 %
Trace Edges	27.432	32.703 %
K Curvature	10.606	12.644 %
Convex Hull	24.075	28.701 %
Compute Rectangle Attributes	19.588	23.351 %
Access Rectangle Attributes	1.048	1.249 %
Total	83.883	100.000 %

Table 7: Low level portions of benchmark, executed on the **IUA**.

Timing Proportions

Program Part	SUN / CM	CM / IUA (back end times)
Read in Image	2.08	1138.52
Connected Components	1.63	101.46
Select Masters	0.55	312.13
Label Regions	0.70	226.50
Trace Edges	1.58	140.01
K Curvature	1.43	114.75
Convex Hull	0.21	269.26
Compute Rect. Attr.	0.70	122.89
Access Rect. Attrib.	1.53	10.74
Total	1.13	169.96

Table 8: Proportions of Execution Times.

5.2.2 Comparison of the Timings

In the tables 5, 6 and 7, the results of the timings for the SUN, the CM and the IUA are presented. For the CM times, several runs with the same input were made, and the result presented is the average of these runs.
In table 8, the ratios of the timings are presented. The SUN/CM ratio is the SUN time divided by the total CM time, and the CM/IUA ratio is the CM back end time divided by the IUA (back end) time. As before, the total times of the SUN and the CM are compared, whereas the back end times of the CM and the IUA are compared.

It is interesting to note that the CM is almost as slow as the SUN, however the IUA is (comparing back end with back end times) 170 times faster than the CM.
It is also striking that, comparing the percent numbers, the SUN percentages are similar to the percentages of the waiting time of the CM back end, and the back end percentages of the CM are similar to the percentages of the IUA.
In the following, the respective parts are investigated separately.

- In the small program segment **Connected Components**, there is only a test for equality, inversion, and query for size of the parallel variable (Size). As seen in the above description, these dyadic or monadic operators are used frequently in the benchmark. The tables show that for Connected Components, the CM is about 1.5 times faster than the SUN, and that the IUA is about 100 times faster than the CM. This clearly minimizes the difference between the CM and the IUA, and reverses the relation between the SUN and the CM from the above described segments.
  The use of the function Size should be in the range of the function *Index*, because it is just an access to a member of the class of the parallel variable. Nevertheless, with the measured times for the basic functions which were tested on integer and float, a timing proportion CM-IUA of 100 is not reached (except for the non-explicable timings).

- In the **Select Masters** part of the benchmark, one Coterie network is built, and the *Index* and *RegionSelectMax* functions are executed within it. This gives a good estimate of how fast the Coterie operation *RegionSelectMax* is with different regions, as the *Index* function is almost negligible on all machines. We see that the SUN is two times **faster** in Select Masters than the CM, and the IUA is about 312.13 times faster in Select Masters than the CM.

- Another program segment, which uses predominantly Coterie operations is the **Label Regions** segment, which builds up a Coterie network and does an *Index* and a *RegionBroadcast* operation. Here, the SUN is 1.5 times faster than the CM and the IUA is 300 times faster than the CM. These proportions are very similar to the Select Masters proportions.
  The timings for the two segments indicate clearly that the Coterie operations are a major bottleneck for the CM, and that the IUA is very fast with integer Coterie operations.

- In **Trace Edges**, the CM is 1.5 times faster than the SUN, and the IUA is 140 times faster than the CM. This is close to the average, but slightly in opposition to the Coterie constellation.
  In Trace Edge, apart from some dyadic and monadic operators, there are also the

nearest neighbor operations. From the basic operations we know that in the nearest neighbor operations, the IUA is 26 times (int) faster than the CM, and the CM is 3.5 (int) times faster than the SUN. This CM-IUA proportion is one of the higher numbers of all functions, and points to a high ratio with neighboring functions on UCharPlane in Trace Edges.
Furthermore, in Trace Edges, the Select active function is sometimes used, whose timings are not reasonable with the IUA simulator.

- For the **K-Curvature** part, the CM is 1.5 times faster than the SUN, and the IUA is about 110 times faster than the CM. In this segment, there are no Coterie operations, and the participating functions and factors are similar to the Trace Edges part.

- In the **Convex Hull** segment, the SUN is roughly 5 times faster than the CM, and the IUA is roughly 270 times faster than the CM. It is very surprising that the SUN is 5 times faster in this segment than the CM, because this difference was not recorded in any of the listed basic functions before. Because the only invariant in the timings of the functions is represented by the Coterie operations, this means that either the Coterie network produces an optimal sequence of storage accesses in the SUN, and/or the Coterie regions have an inherent time-taking shape, like a spiral.
  Looking at the division of the total time into front end and back end time on the CM shows that actually, the back end time is about 4 times greater than the front end time. In the basic functions, the only functions whose back end time was greater than the excess front end time were the Coterie operations.

- In the **Compute Rectangles Attributes** segment, the SUN is 1.5 times faster than the CM, and the IUA is 120 times faster than the CM. The advantage of the SUN compared to the CM is smaller than in the Convex Hull, and also the advantage of the IUA over the CM is below the overall average. The CM-IUA comparison is lower than average and the SUN-CM comparison is still the reverse of most routines. The fast speed of the SUN compared to the CM still can only be explained by the Coterie operations, which are frequent in the routine's loop. In other routines, the influence of the Coterie operations is diluted by operations like the 36 integer divisions in the int square root operation.

- In the **Access Rectangle Attributes** segment, the CM was 1.5 times faster than the SUN, and the IUA was roughly 10 times faster than the CM. Here, the *Access*, the *Any* operation, the *SelectMax* operation, the *Index* operation and the *Select* call appear in the program segment. The functions *SelectMax* and *Any* are global operations, which have a low IUA-CM ratio for integers and float. *Index* has a

high ratio, but is only used once. With shorter Char variables, the ratio might be somewhat higher, but would still be on the low end.

It is surprising that the IUA is 60 times faster than the CM because it is only 1.3 times faster for the *Any* function, 4 times faster for int *SelectMax*, roughly 25 times faster for *Index*, and the *Select* calls are much less than a factor of 60 faster on the IUA than on the CM. The only function which could cause the difference is *Access*.

In summary, the timings for the benchmark confirmed the hypotheses previously stated: The Coterie network is a major bottleneck for the CM. Every time Coterie operations are present, (in Select Masters, Label Regions, Convex Hull and Compute Rectangle Attributes) the SUN is faster than the CM. Also, most of these times, the IUA is roughly more than 300 times faster than the CM.
When there are no Coterie operations present, usually the CM is 1.5 to 2 times faster than the SUN, and the IUA is 100 to 200 times faster than the CM.

5.2.3 The *Lisp Timings for the Benchmark

Also *Lisp times are reported which were obtained by Thinking Machines on the CM-2, with different vp-ratios. In case the vp-ratio was greater than one, (it is 16 with Connected Components + Select Masters + Label Regions and Trace Edges), the timing was divided by the vp-ratio. The used input is different from the one used for the previous timings in the ICCL. The input image also varies in size from the segments used above. N.A. in the table means that the time for a particular routine was not reported.

Program Part	time in millisec.
Read in Image	25.00
Connected Components + Select Masters + Label Regions	13.75
Trace Edges	11.875
K Curvature	15.03
Convex Hull	0.18
Compute Rectangle Attributes	n.a.
Access Rectangle Attributes	n.a.
Total	n.a.

Table 9: Low level portions of benchmark, executed on the **CM in *Lisp**.

It is very difficult to compare these times with the ICCL implementations on the SUN, CM and IUA simulator, first because the input image is different, second because

some timings were recorded with virtualization, and third because some times are missing. The only times directly comparable are the computation of corners in K-Curvature, because it does not depend heavily on the input data (no loop is dependent on the input, and no Coterie operations are used there).
With 15.03 milliseconds, the **K Curvature** takes somewhat longer than the K-Curvature in the IUA simulator (11.21 msec), however, front end time was not calculated for the IUA simulator. This time is 341 times faster than the time recorded for the K-Curvature ICCL implementation on the CM, comparing front end times. This shows that there is a large overhead involved in implementing the ICCL on the CM on top of PARIS.
The segments **Connected Components**, **Select Masters** and **Label Regions** in Tables 5, 6, 7 and 8 seem to be summarized into **Label Components**. Timings can be dependent on input data with all the parts except K-Curvature.
The *Lisp Implementation takes 13.75 milliseconds for the Label Components, whereas the IUA takes 2.88 milliseconds (5 times faster), but the CM version of the ICCL takes 20 times longer than the *Lisp implementation.
The Trace Edges segment is faster than the IUA by the factor 2.3. The Convex Hull is very fast, exactly 134 times faster than on the IUA.
The differences in timing for the program segments except for K-Curvature could have all kinds of causes, from a complicated Coterie network patterns to many corners in the regions. Therefore no statements can be made in comparison.

5.3 The Fast Line Extraction Algorithm

In this subsection, the times on the three machines of the previously introduced fast line extraction algorithm are reported. The timing intervals are equal to the steps discussed in section 5.
Timings were done with the same algorithm on the three machines, however, in order to run the line extraction algorithm on the simulator, some commands were slightly changed. For example, the IUA simulator does not support the *float RoutePlus* operation or the *float RegionSelectMin/Max* operations, so the variables which should be processed by *float RegionSelectMin/Max* operations were defined as int. In this case, it is possible to convert all the *float RoutePlus* operations into Coterie operations. On the CM, as usual, the algorithm is run several times and the times are averaged.

Program Part	time in seconds	perc. of tot. time
Input	0.20	5.99 %
Compute Mask	0.12	3.60 %
Create Bucket Scheme	0.24	7.19 %
Build two Coteries	0.16	4.79 %
Create Counts in Coterie	0.25	7.49 %
Build Final Coterie	0.05	1.50 %
Compute Master Pixel	0.08	2.40 %
Split Regions into three	0.49	14.67 %
Compute Peak Points	0.74	22.16 %
Compute Line	0.57	17.07 %
Compute Endpoints	0.16	4.79 %
Filter Length	0.20	5.99 %
Output	0.08	2.40 %
Total	3.34	100.00 %

Table 10: Fast Line Extraction Algorithm, executed on the **SUN**.

Program Part	time back end	time front end	total time	perc.ti. ba.end	perc.ti. fr.end	perc. tot.ti.
Input	14.83	42.08	56.92	0.61 %	2.56 %	1.39 %
Compute Mask	13.09	50.41	63.50	0.53 %	3.06 %	1.55 %
Create Bucket Scheme	40.12	144.91	185.03	1.64 %	8.80 %	4.52 %
Build two Coteries	23.93	99.93	123.86	0.98 %	6.07 %	3.03 %
Create Counts in Coterie	241.74	67.37	309.10	9.88 %	4.09 %	7.55 %
Build Final Coterie	6.08	22.94	29.02	0.25 %	1.39 %	0.71 %
Compute Master Pixel	81.00	18.01	99.01	3.31 %	1.09 %	2.42 %
Split Regions into three	332.22	190.83	523.05	13.58 %	11.59 %	12.78 %
Compute Peak Points	1147.32	356.84	1504.16	46.90 %	21.68 %	36.75 %
Compute Line	178.90	179.20	358.10	7.31 %	10.89 %	8.75 %
Compute Endpoints	175.83	66.62	242.45	7.19 %	4.05 %	5.92 %
Filter Length	104.41	74.34	178.76	4.27 %	4.52 %	4.37 %
Output	87.06	332.51	419.56	3.56 %	20.20 %	10.25 %
Total	2446.53	1645.98	4092.51	100.00 %	100.00 %	100.00 %

Table 11: Fast Line Extraction Algorithm, executed on the **CM** with times in milliseconds.

Program Part	time in millisec.	perc. of tot. time
Input	0.015	0.037 %
Compute Mask	0.246	0.587 %
Create Bucket Scheme	2.391	5.696 %
Build two Coteries	0.361	0.861 %
Create Counts in Coterie	2.787	6.640 %
Build Final Coterie	0.122	0.290 %
Compute Master Pixel	0.854	2.035 %
Split Regions into three	7.295	17.378 %
Compute Peak Points	10.774	25.668 %
Compute Line	14.399	34.304 %
Compute Endpoints	2.195	5.230 %
Filter Length	0.493	1.175 %
Output	0.041	0.099 %
Total	41.975	100.000 %

Table 12: Fast Line Extraction Algorithm, executed on the **IUA**.

Program Part	SUN / CM	CM / IUA (back end times)
Input	3.51	962.99
Compute Mask	1.89	53.13
Create Bucket Scheme	1.30	16.78
Build two Coteries	1.29	66.23
Create Counts in Coterie	0.81	86.73
Build Final Coterie	1.72	50.00
Compute Master Pixel	0.81	94.81
Split Regions into three	0.94	45.54
Compute Peak Points	0.49	106.49
Compute Line	1.59	12.42
Compute Endpoints	0.66	80.09
Filter Length	1.12	211.70
Output	0.19	2102.90
Total	0.82	58.29

Table 13: Proportions of Execution Times.

It is interesting that by replacing two Routes with *RegionBroadcast* and converting float numbers to integer in order to execute on *RegionSelectMax*, the CM actually runs slower (4092 versus 3357 seconds). This could mean that even with non-trivial regions, the CM prefers Route with collisions over a *RegionSelectMax* function.

With this algorithm, the CM is even slower than the SUN. If the algorithm without the changes due to the IUA simulator is executed, however, the CM is slightly faster than the SUN again. The times of the CM and the SUN are comparable, whereas the IUA

timing is about 60 times faster than the CM (comparing back end times), see also Table 13.
This algorithm works mostly on integer and floating point, so the ratios of Table 4 will be more valid.

1. In the **Compute Mask** part, there are many nearest neighbor, comparison and *Select* operations, because the mask is computed. The nearest neighbor (high ratio) and *Select* operations seem to balance out in the CM-IUA ratio.

2. The **Create Bucket Scheme** part contains many comparisons, arithmetic operations and especially *Select* operations, because in this part, the atan is computed. What is peculiar in is that the excess front end time on the CM is 3.5 times larger than the back end time. In this part, there are many control constructs, which are executed on the front end. The ratio SUN-CM, however is still above the average for this algorithm. The CM-IUA ratio is remarkably low in this part. Arithmetic operations certainly contribute to the low CM-IUA ratio, see Table 4. It is unknown how the Select functions contribute to the ratio, since their execution time could be small.

3. **Build two Coteries** basically runs two connected components algorithms for the two region representations. The main part of the connected components algorithm is again nearest neighbor operations, and besides that, there are test for edge, and, or, test for equality, inversion and shift. The CM is slightly faster than the SUN and the CM-IUA ratio is within the average range.

4. In **Create Counts in Coterie**, there are 4 *RegionBroadcast* on IntPlane, 2 *RegionSelectMin* on IntPlane and 2 *RoutePlus* on IntPlane operations, as far as communication operations are concerned, which make up the main part. It is striking in Table 13 that, as soon as Coterie operations are present in a program, the SUN is faster than the CM. The distance in speed between the IUA and the CM also lies well above average.

5. The timings for **Build Final Coterie** should be similar to *Build two Coteries*, with the difference that in this segment, no threshold and no edge operation was tested. It is noticeable, however, that the CM is faster than before, compared both to the SUN as well as to the IUA. This could be due to the absence of the edge function, which could be similar to the *Index* function. The *Index* function has a fairly low SUN-CM ratio and a fairly high CM-IUA ratio. Other functions, however, like the absence of two 'and' functions could have influenced the outcome as well.

6. In **Compute Master Pixel**, the main operations are one *RegionSelectMin* and one *RegionBroadcast* on IntPlane each. Apart from that, there is an *Index*, a shift and an and operations. The timing of this part shows again that in case a program piece contains Coterie operations, the timings of the other operations can be neglected, the SUN is faster than the CM, and the IUA-CM ratio is much greater than the average factor in the program.

7. The part **Split Regions into three** consists mainly out of Coterie functions (4 *RegionBroadcast*, 3 IntPlane and 1 on BitPlane, 4 *RegionSelectMin/Max* on IntPlane), arithmetic functions and Select functions. The SUN is slightly faster than the CM, but the factor CM-IUA is below average, i.e. the CM is faster compared to the IUA than usual. The arithmetic operations are probably the primary reason for this effect.

8. In **Compute Peak Points**, the differences are very striking: The SUN is 2 times faster than the CM, and the IUA is about 100 times faster than the CM. In this part, there are 18 *RegionSelectMax* operations on IntPlane and 18 *RegionBroadcast* operations on ShortPlane and one *RegionBroadcast* on BitPlane, which explains the differences. This is the most time-taking section of the algorithm.

9. **Compute Line** has only 3 *RegionBroadcast* operations (2 on IntPlane and on BitPlane), but many arithmetic operations and *Select* invocations. Therefore, the CM is again faster than the SUN, and the IUA is only about 10 times faster than the CM.

10. **Compute Endpoints** has 2 *RegionSelectMin/Max* and 2 *RegionBroadcast* operations on integer, and some arithmetic operations. The Coterie operations seem to dominate the timings.

11. **Filter Length** has 2 *RegionSelectMin/Max* and 3 *RegionBroadcast* operations, all are on 8-bit variables. As the Coterie operations in this section are all on 8-bit variables, the CM-IUA ratio is higher than that in the previous section. The section itsself, however, is the second least time-taking.

In order to narrow down the instructions responsible for timing differences in non-Coterie sections, more experiments would have to be executed. The experiments done so far e.g. do not consider different length variables and do not comprise enough functions. Therefore, it is hard to come up with reasons for specific timing sections, except if there are Coterie operations in them. If there are and the timing exhibits lower SUN-CM ratio as usual and higher CM-IUA ratio as usual, the influence of Coterie functions is distinct. It has also been observed that these ratios are more distinct the smaller the field is on which the Coterie function operates.

5.4 The Depth Recovery Algorithm

In this subsection, another vision algorithm ([3]) is briefly introduced and run on the 3 different machines.
In this algorithm, a parallel dense depth map is created, with the input of temporally separated images from a forward-moving sensor. Correspondences between them are established in parallel through correlation. The correspondences are used to determine the translational and rotational motion parameters of the camera through a parallel motion algorithm. This is done by first determining the approximate translational and rotational parameters and then constraining the search for the exact translational and rotational parameters. Finally, the dense map is computed from the image correspondences and the computed motion parameters.

As with the Fast Line Extraction Algorithm, in this algorithm the *float RegionSelectMin/Max* operations are changed to *int RegionSelectMin/Max* operations in order to avoid problems with the IUA simulator.

Program Part	times in seconds	perc. of total time
Correlation_9x9mask	30.16	46.85 %
Correlation_3x3mask	12.90	-
Translation part	1.22	1.90 %
Translation depth	0.58	0.90 %
Translation and rotation	32.41	50.35 %
Total	64.37	100.00 %

Table 14: Depth Recovery Algorithm: Timings run on **SUN** in seconds.

The total result with the original algorithm is also 64.37 seconds.

Program Part	time back e.	time front e.	total time	perc. back end time	perc. front end time	perc. tot.time
Correlation_9x9mask	4.32	11.00	15.32	85.92 %	14.85 %	19.36 %
Translation part	0.47	1.22	1.69	9.35 %	1.65 %	2.14 %
Translation depth	0.16	0.31	0.47	3.18 %	0.42 %	0.59 %
Translation and rotation	0.078	61.56	61.64	0.016 %	83.09 %	77.91 %
Total	5.028	74.09	79.12	100.00%	100.00 %	100.00 %

Table 15: Depth Recovery Algorithm: Timings run on **CM** in seconds.

The total result with the original algorithm is 82.31 seconds.

Program Part	times in seconds	perc. of total time
Correlation_9x9mask	0.22740	92.97 %
Correlation_3x3mask	0.16583	-
Translation part	0.01017	4.16 %
Translation depth	0.00441	1.80 %
Translation and rotation	0.00262	1.07 %
Total	0.24460	100.00 %

Table 16: Depth Recovery Algorithm: Timings run on **IUA** in seconds.

Program Part	SUN / CM	CM / IUA (back end times)
Correlation_9x9mask	1.968	18.997
Translation part	0.722	46.214
Translation depth	1.234	36.281
Translation and rotation	0.526	29.771
Total	0.814	20.556

Table 17: Proportions of Execution Times.

In the first step, the correspondences are found. This requires many nearest neighbor, *plus* operations and one *RoutePlus* operation, which operate all on short variables. Table 17 shows the proportions of execution times. The CM-IUA ratio in Table 17 lies between the CM-IUA ratios from Table 4 for *int plus* and nearest neighbor functions (which is 35) and the CM-IUA ratio for *int RoutePlus* functions (which is 10). Taking into account the execution times and number of occurrences of the functions, the final ratio would be higher than the 19 in Table 17. This indicates that the *RoutePlus* operation with a non route-to-self pattern is not as costly on the CM as it is on the IUA. The SUN-CM ratio for finding correspondences shows that executing the *RoutePlus* operation on the CM with a non-route-to-self pattern is costlier than executing it with a route-to-self pattern. The reason is that despite the higher ratios in Table 4 for nearest neighbor and *int RoutePlus* operations, the SUN-CM ratio is as low as for the *int plus* operation.

The second part computes the approximate foe by the modal hough method from the displacements obtained in the first step. In the translation part, two Coterie operations (*int RegionSelectMin/Max*) and some nearest neighbor operations are executed, apart from front end operations.
Because of the input data dependence of the CM for the Coterie functions, the CM-IUA ratio is much higher than in Table 4 for the Coterie functions, and the SUN-CM ratio is much lower than in Table 4.

In the Translation depth part, the depth map is found from the foe computed in the last step. This is effectively the approximate translational motion. There are mainly

floating point plus, minus, multiply and divide operations in this part. The SUN-CM ratio conforms with the ratio in Table 4, but the CM-IUA ratio is much higher than the CM-IUA ratio for floating point arithmetic operations in Table 4. One explanation for this is that the *int plus* operation with its high CM-IUA ratio helps in raising the overall CM-IUA ratio for this part.

The Translation and Rotation part distributes hypothesized translations along the pixels, and for each pixel finds the optimal rotation, where the foe obtained in the second step is used. Then, the depth map is obtained. This part contains many front end instructions, but also some Select functions for the back end. The fact that there are many front end instructions in this part is proven by Table 16, where the fourth part has the biggest percentage of total time for the front end of the CM, while its back end percentage is very small.

Part two of the depth recovery algorithm is another example which shows that the Coterie functions contribute to the speed up of the IUA versus the Connection Machine. Part one shows that the *RoutePlus* function with non-route-to-self pattern are slower on the CM than the route-to-self structure. Also, part one shows that times for the *RoutePlus* function could increase by a larger factor in comparison to *RoutePlus* function with route-to-self pattern than on the CM.

The algorithm also shows that Coterie functions are useful in the motion field, in this case to find the best displacement vector in each region.

6 Summary

This work described the transportation of the ICCL onto the Connection Machine, the problems connected with the transportation, and comparisons of the performance for the ICCL on the CM, the SUN and the IUA simulator. As there were not many vision algorithms implemented in the ICCL yet, a new line extraction algorithm was developed, which, together with the benchmark and the dense depth retrieval algorithm, served as a good testbed for the performances of the ICCL on the CM.
The different vision algorithms address different operations in the ICCL. The benchmark algorithm uses different size variables, but no floating point operations and no *Route[Plus]* operations. The dense depth algorithm uses mainly nearest neighbor and *RoutePlus* operations, while the line extraction algorithm uses many Coterie, floating point and some *RoutePlus* operations.

Experimental results show that the ICCL implementation on the IUA is fastest for all algorithms, and that the SUN and CM implementations have mostly comparable speeds. The CM lacks speed especially when Coterie operations are used frequently. This is apparent in the line extraction algorithm, where the SUN is always faster than the CM when more Coterie operations are used. Concerning *RoutePlus* operations, the CM gets slower when the pattern is changed from route-to-self to non-route-to-self, while the SUN timings stay the same; evidence is found for this in the first part of the depth retrieval algorithm. With evidence from the 'Compute Mask' and 'Compute Line' part of the line extraction algorithm, the CM is faster in nearest-neighbor functions than the SUN, and slightly faster in arithmetic functions.

The IUA simulator is fastest in all algorithms and operations. As shown in Table 4 and the percentage of time it spends in 'Compute Line' in the line extraction algorithm (Table 12), it has a smaller advantage for floating point 'plus', but is still faster than the CM and SUN. The timings for the benchmark, which are faster than the CM by factors of over one hundred, show that using no floating point operations, smaller plane sizes and no routing operations achieves very high speedups over the CM. This is also due to the missing floating point accelerator in the IUA. The speedups in comparison to the CM are also biggest if many Coterie operations are used on the IUA simulator (see 'Select Masters' in benchmark, 'Compute Peak Points' in line extraction algorithm and 'Translation part' in depth recovery algorithm).

The point in this paper is to show that several vision algorithms need to use the Coterie Networks and that it has no equivalent implementation on the CM, and that therefore vision algorithms become very costly on the CM. The SUN execution times are shown to provide for an 'upper limit' of execution times of the CM.
We have investigated the benchmark, the line extraction algorithm and the depth recovery algorithm, which all benefit from the existence of Coterie Networks. The Coterie

Network, however, has no hardware equivalent in the CM, and therefore an algorithm has to be constructed to implement it. In Section 3.1.4, it was shown that no two-dimensional region defining function exists in PARIS, and that the most useful PARIS function for implementing the Coterie functions in parallel is a one-dimensional scan function. This function has to be implemented in an iterative approach in order to run Coterie functions in parallel on a processor array. Therefore, simulating Coterie functions in parallel is costly.

*Lisp timings have also been reported for the benchmark. The only timing which allows a valid comparison is 1.5 times faster on the IUA with the ICCL versus the CM with *Lisp, which is for parts with no Coterie operations. The implementation of the ICCL on the CM is another 81 times slower than the *Lisp implementation on the CM, which could improve timings of vision algorithms on the CM, if implemented in *Lisp. Because of the scalability problem, however, the advantage of the Coterie operations on the IUA versus the CM would still remain considerably high.

In summary, this paper shows through several example algorithms that the Coterie Networks, a special-purpose hardware feature are of great advantage for programming several vision algorithms, and that in general the ICCL is faster on the IUA than on the CM. The paper also documents the transportation of the ICCL on the CM, which is easy except for the transportation of the Coterie Network.

References

[1] J.B. Burns, A.R. Hanson and E.M. Riseman. "Extracting Straight Lines." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. PAMI-8, No. 4, July 1986.

[2] J.H. Burrill. "The Class Library for the IUA, Tutorial." *AAI Amerinex Artificial Intelligence, Inc.*, 1992

[3] R. Dutta and C. C. Weems. "Parallel Dense Depth from Motion on the Image Understanding Architecture," , IEEE Computer Society Conference on Computer Vision and Pattern Recognition, New York City, New York, June 15–17, 1993, pages 154–159.

[4] P. Kahn, L. Kitchen and E. Riseman. "Real Time Feature Extraction: A Fast Line Finder for Vision-Guided Robot Navigation." *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol. 12(11):1098-1102 (Nov. 1990)

[5] L.J. Kitchen and J. Malin. "The Effect of Spatial Discretization on the Magnitude and Direction Response of Simple Differential Edge Operators on a Step Edge, Part1: Square Pixel Receptive Fields." *CmpSci Technical Report, University of Massachusetts at Amherst*, 1987-34, April.

[6] C.C. Weems, S.P. Levitan, A.R. Hanson and E.M. Riseman. "The Image Understanding Architechture." *International Journal of Computer Vision*, 2, 251-282, 1989.

[7] C. Weems, E. Riseman and A. Hanson. "An Integrated Understanding Benchmark: Recognition of a 2 1/2 D 'Mobile'." *CmpSci Technical Report, University of Massachusetts at Amherst* 1988-34.