90-80-3

A Production-Quality C* Compiler for a Hypercube Multicomputer

- Philip J. Hatcher, Anthony J. Lapadula, Robert R. Jones
Department of Computer Science
University of New Hampshire
Durham, NH 03824

Michael J. Quinn, Ray J. Anderson
Department of Computer Science
Oregon State University
Corvallis, OR 97331-3202

A Production-Quality C* Compiler for a Hypercube Multicomputer

Philip J. Hatcher[a], Michael J. Quinn[b], Anthony J. Lapadula[a], Robert R. Jones[a], and Ray J. Anderson[b]

[a]Department of Computer Science, University of New Hampshire, Durham, NH 03824 U.S.A.

[b]Department of Computer Science, Oregon State University, Corvallis, OR 97331 U.S.A.

Abstract

We describe our third generation C* compiler for a hypercube multicomputer. This production-quality compiler features a full implementation of the language, including general pointer-based communication and support for separate compilation. The compiler incorporates new optimizations and utilizes an improved set of communication primitives. It supports a variety of standard domain decomposition primitives, and it also allows the programmer to specify a custom mapping of data to the distributed memories of the hypercube. The performance of this compiler on benchmark programs demonstrates that high efficiency can be achieved executing SIMD code on multicomputer architectures.

1 Introduction

C* is the data parallel C superset designed by Thinking Machines for the Connection Machine processor array [12]. Many languages that might be classified as data parallel have been proposed, including Blaze [7], Booster [8], Coherent Parallel C [2], Dino [13], The Force [5], Kali [6], the paralation model [14], and Parallel Pascal [10]. The fully synchronous semantics and the local view of the computation distinguish C* from these other languages. Synchronous execution eliminates race conditions and makes C* programs deterministic, greatly reducing the complexity of program debugging. Having a local view of the computation simplifies the introduction of data decomposition directives, which are essential in a distributed memory environment.

Our approach has several advantages over systems that translate imperative sequential languages for execution on parallel machines [1, 11]. When using a sequential language, programmers often introduce unnecessary sequentiality into their programs. Many problems have data parallel solutions that are shorter and more elegant than the corresponding sequential solutions.

C* associates a virtual processor with a fundamental unit of parallelism. The programmer takes a local view of the computation, expressing the algorithm in terms of the operations concurrently performed by each virtual processor. In contrast, traditional sequential programs take a global view of the computation; the algorithm is expressed in terms of operations performed on arrays. Many systems that support the global view require programmers to provide directives that perform data mapping. This requirement detracts from the global view as the programmer must think locally when considering issues such as load balancing and communication costs. If the language is to contain data mapping constructs, it should take a local view of the computation.

Advocates of functional programming languages argue that the property of referential transparency provides many opportunities for parallel execution of functional programs—notably in

the concurrent evaluation of function arguments. Moreover, like C*, the programs of functional programming languages do not suffer from timing problems, are deterministic, and can be executed on sequential as well as parallel architectures. However, parallel implementations are hindered by the difficulty of implementing functional languages on sequential processors. The overhead associated with implementations of functional programming languages makes it extremely difficult for functional programs to be competitive with the performance of programs hand-crafted for a particular parallel machine. C* programs can be competitive with hand-written parallel programs.

In addition, functional programming languages suffer from a lack of transparency. It is difficult for a programmer to understand the reasons for the efficiency, or lack of efficiency, demonstrated by a particular program. The explicit parallelism of C* makes it easier for the programmer to understand the run-time behavior of programs.

This paper describes experience with our third generation C* compiler for a hypercube multicomputer. Our first compiler was based upon a general, but less efficient, control flow model [9]. Our second compiler introduced both efficient flow of control and a powerful communication optimizer, but was implemented in prototype form only [4].

The third, and current, compiler is intended to be production quality. It features a full implementation of the language, including general pointer based communication and support for separate compilation. The compiler includes new optimizations and utilizes an improved set of communication primitives.

In the remainder of this paper we give an overview of C*, describe the design and implementation of our compiler, and present the results of compiling and executing a set of benchmark C* programs on a 64 node NCUBE 3200 hypercube.

2 The C* Programming Language

The original C* language was documented in a technical report issued by Thinking Machines in 1987 [12]. However, the language has evolved differently in our systems and at Thinking Machines. Our primary divergence from the original language design is in the addition of *virtual topologies* (discussed at the end of this section) and in the implementation of pointers (discussed in Section 3.6).

C* is a high-level parallel programming language. It supports a shared memory model of parallel computation. A programmer can assume that all processors share a common address space. C* supports virtual processors. This allows a programmer the convenience of ignoring the actual number of physical processors available. All C* virtual processors execute synchronously. A synchronous execution model eliminates all timing problems, makes programs deterministic, and greatly reduces the complexity of program comprehension and the difficulty of program debugging.

The conceptual model presented to the C* programmer consists of a front-end uniprocessor attached to an adaptable back-end parallel processor. The sequential portion of the C* program (consisting of conventional C code) is executed on the front end. The parallel portion of the C* program (delimited by C* constructs not found in C) is executed on the back end.

The back end is adaptable in that the programmer selects the number of processors to be activated. This number is independent of the number of physical processors that may be available on the hardware executing the C* program. For this reason the C* program is said to activate *virtual* processors when a parallel construct is entered.

Virtual processors are allocated in groups. Each virtual processor in the group has an identical memory layout. The C* programmer specifies a virtual processor's memory layout

```
domain cell {
    double energy, density, temperature, pressure;
};
```

Figure 1: Declaring a domain.

```
#define KDIM 54
#define LDIM 54

domain cell mesh[KDIM][LDIM];
```

Figure 2: Declaring virtual processors.

using syntax similar to the C `struct`. A new keyword `domain` is used to indicate that this is a parallel data declaration. Figure 1 contains a partial domain declaration for the mesh points of a hydrodynamics simulation. As in C structures, the names declared within the domain are referred to as *members*.

Instances of a domain are declared using the C array constructor. Each domain instance becomes the memory for one virtual processor. The array dimension therefore indicates the size of the virtual back-end parallel processor that is to be allocated. Figure 2 contains a domain array declaration. Note that domain arrays can be multidimensional. The number of virtual processors allocated is the product of the array dimensions.

Data located in C*'s front-end processor is termed *mono* data. Data located in a back-end processor is termed *poly* data.

Figure 3 illustrates the C* *domain select* statement. The body of the domain select is executed by every virtual processor allocated for the particular domain type selected. The virtual processors execute the body synchronously. The domain members are included within the scope of the body of the domain select. These names refer to the values local to a particular virtual processor.

The code executing in a virtual processor of a C* program can reference a variable in the front-end processor by referring to the variable by name. A variable that is visible in the immediately enclosing block of a domain select statement is visible within the domain select. The C* compiler is responsible for making mono variables accessible at run time to the virtual processors.

```
[domain cell].{
    double temp1;
    temp1 = calculate_temperature(energy, density);
    temperature = (temp1 > TFLR ? temp1 : TFLR);
    pressure = calculate_pressure(temperature, density);
}
```

Figure 3: Activating virtual processors.

Similarly, the members of a domain instance are accessible everywhere in a program. The members of one domain can be read and written from within a domain select statement for a different domain. Poly data can also be read and written from the sequential portion of the program. The syntax employed is to provide a full domain array reference followed by a member reference.

C*, like C++, has a keyword `this`. In C* `this` is a pointer to the domain instance currently being operated on by a virtual processor. Pointer arithmetic on `this` can be performed to access other virtual processors' members.

C* provides a set of *reduction* operators. These operators accumulate poly values into a mono location. All C assignment operators are overloaded for this purpose. New operators have been added to the language to express reductions that compute the minimum and maximum of a set of poly values.

The sequential portion of a C* program is just C code and executes following the normal C semantics. Conceptually, the parallel sections of a C* program execute synchronously under the control of a *master program counter* (MPC). A virtual processor's local program counter is either active, executing in step with the MPC, or inactive, waiting for the MPC to reach it.

For example, the MPC steps through an `if-then-else` statement by first evaluating the control expression, then executing the `then` clause, and finally executing the `else` clause. A local program counter would also proceed first to the control expression. However, if the expression evaluated to zero (*false* in C), then the local program counter would proceed to the `else` clause and wait for the MPC to reach it. If the expression evaluated to non-zero (*true* in C), then the local program counter would wait at the `then` clause for the MPC.

As well as being synchronous at the statement level, C* is also synchronous at the expression level. No operator executes within a virtual processor unless all active virtual processors have evaluated their operands for the operator. Once the operands have been evaluated, the operator is executed as if in some serial order by all active virtual processors. This seemingly odd use of a serial ordering to define parallel execution is required to make sense of concurrent writes to the same memory location.

Our implementation of C* allows additional information to be provided by the programmer in order to aid the compiler in the mapping of virtual processors to physical processors. The array dimension of the domain array establishes a *virtual topology*. A one-dimensional domain array is considered to be a ring of virtual processors. A two-dimensional domain array is considered to be a two-dimensional mesh of virtual processors. In general, an n-dimensional domain array is considered to be an n-dimensional mesh of virtual processors with wrap-around connections.

These virtual topologies establish a convention of *locality*. Virtual processors that are adjacent in a virtual topology should be mapped by a compiler to physical processors that are "near" each other. On some architectures this information will be of little value and can be ignored by the compiler. On other architectures this can lead to large efficiency gains if the programmer exploits the feature and the compiler effectively implements it.

For the common topologies (low dimension domain arrays), the compiler recognizes macros that provide convenient access to adjacent elements in the virtual topology. The macros take a domain element address and return the address of the appropriate adjacent domain element. In the one dimensional case macros called "successor" and "predecessor" are provided. In the two dimensional case the macros "north," "south," "east" and "west" are provided.

Additional keywords exist to aid the compiler in mapping a larger number of virtual processors to a smaller number of physical processors. The keyword *contiguous* indicates that blocks of adjacent domain elements should be mapped to the same physical processor. The keyword *interleaved* indicates that domain elements should be assigned to physical processors in a

The C* construct:

```
while (condition) {
  statement_list1;
  communication;
  statement_list2;
}
```

is translated into the following C code:

```
temp = TRUE;
do {
  if (temp) {
    temp = condition;
  }
  if (temp) {
    statement_list1;
  }
  communication;
  gtemp = global_or (temp);
  if (temp) {
    statement_list2;
  }
} while (gtemp);
```

Figure 4: Translation of a C* `while` statement.

round-robin fashion. In the two dimensional case, the keywords *contiguous_row*, *contiguous_col*, *interleaved_row*, and *interleaved_col* exist to map rows and columns *in toto*. The keyword *user-spec* indicates that the compiler should utilize user-written macros to implement an arbitrary mapping.

3 Design of the C* Compiler

3.1 Overview

The compiler is a C*-to-C translator. It parses C* input, transforms C* syntax trees into C syntax trees, and then unparses. We use our own port of the GNU C compiler to compile the output C code for the nodes of the NCUBE 3200.

Our implementation consists of four major components, each discussed in its own subsection below: processor synchronization, virtual processor emulation, communication optimization, and a run-time communication library. Our implementation of pointers is discussed in Section 3.6.

3.2 Minimizing the Number of Processor Synchronizations

In C* expressions can refer to values stored in arbitrary processing elements. All variable declarations state, implicitly or explicitly, which processing element holds the variable being declared. Therefore, the location of potential communication points can be reduced to a *type checking* problem. A sufficient set of synchronization points are the locations at which we identify message passing is potentially needed. We incorporate synchronization into the message-passing routines.

Parallel looping constructs may require additional synchronization. If the parallel loop body incorporates virtual processor interaction, the processors must be synchronized every iteration prior to the message-passing step. Of course, a virtual processor does not actually execute

The C* construct:

```
if (condition) {
   statement_list1;
   communication;
   statement_list2;
}
```

is translated into the following C code:

```
temp = condition;
if (temp) {
   statement_list1;
}
communication;
if (temp) {
   statement_list2;
}
```

Figure 5: Translation of a C* `if` statement.

the body of the loop after its local loop control value has gone to *false*. Rather, the physical processor on which it resides participates in the message passing and the computation of the value indicating whether any virtual processors are still active. This means that our C* compiler must rewrite the control structure of input programs. Figure 4 illustrates how `while` loops are rewritten.

Since we have incorporated synchronization into communication, all physical processors must actively participate in any message-passing operation. This forces our compiler to rewrite all control statements that have inner statements requiring message passing. Figure 5 illustrates how an `if` statement is handled.

Communication steps buried inside nested control structures are pulled out of each enclosing structure until they reach the outermost level.

The technique just described will not handle arbitrary control flow graphs. For this reason we have not implemented the `goto` statement. We do, however, support the `break` and `continue` statements.

3.3 Efficiently Emulating Virtual Processors

Once the message-passing routines have been brought to the outermost level of the program, emulation of virtual processors is straightforward. The compiler puts `for` loops around the blocks of code that have been delimited by message-passing/synchronization steps. Since within the delimited blocks there is no message passing, there is no interaction between virtual processors. Therefore, it makes no difference in which order the virtual processors located on a particular physical processor execute.

3.4 Optimizing Communication

The compiler eliminates one class of messages by keeping copies of the sequential code and data on each physical processor. Every physical processor executes the sequential code. Note that this adds nothing to the execution time of the program. Because every physical processor has copies of the sequential variables, it can access sequential data by doing a local memory fetch. In other words, assigning the value of a `mono` variable to a `poly` variable can be done without any message passing.

However, when a virtual processor stores to a mono variable, the value may have to be broadcast to update all physical processors' copy of the mono. In many cases the compiler can eliminate this broadcast operation. If it is known that all virtual processors are simultaneously assigning the same value to the mono, then no broadcast is required—all nodes can perform the assignment locally. The compiler must determine that all the virtual processors will execute the assignment and that the right hand side of the assignment will evaluate to the same value on all processors. To conservatively support this analysis, the compiler tags each expression with a Boolean flag that indicates whether the expression's leaves consist of only mono variables and constants. If so, then the expression is known to evaluate to the same value on all processors and is termed a *strictly mono* expression.

Moreover, if a strictly mono expression is the control expression for a conditional or iterative statement, then it is known that all virtual processors will follow the same path through the statement. If a statement is at the outermost control flow level, or is nested within conditional or iterative statements that are all controlled by strictly mono expressions, then the statement is known to be executed by all virtual processors.

The same analysis is used to eliminate the need to perform a "global-or" calculation in the implementation of a loop construct (see Figure 4). If the loop is controlled by a strictly mono expression, then all virtual processors will execute the loop the same number of times. The time-savings achieved is significant if the communication operation within the loop (which required the loop to be synchronized in the first place) is not powerful enough to perform the "global-or" calculation as a side effect of its execution. In addition, the loop control is *sequentialized*—performed only once per physical processor—to reduce the overhead of the virtual processor emulation.

The knowledge gathered about the control flow of the program—whether a given statement is known to be executed by all virtual processors—can be used to effect another strength reduction optimization. If virtual topologies are being used, and all virtual processors are reading a value from their neighbor, the read can be implemented with the more efficient write communication primitive. Since all processors must be able to compute the address of the value being requested by their neighbor, this optimization will not be performed for the fetching of an array element indexed by a poly.

In another use of knowledge gathered about both the control flow and strictly mono expressions, some read operations can be implemented as broadcasts. If all virtual processors are active, and the value being read is expressed as a domain reference using only strictly mono expressions, then all processors are reading the same value, and the processor holding the desired value can broadcast it to the other processors. If the virtual processors are not all active, this optimization can still be performed if the domain reference does not contain any side-effects. The owner of the value can evaluate the value safely, even if the owner is inactive. We conservatively assume the existence of a side-effect if there is a function call or an assignment operator. The compiler attachs to each expression tree a flag indicating whether or not it may have a side-effect.

Side-effect-free assignments to poly values in sequential code are only performed on the processor holding the target of the assignment. Even if there is a side-effect, no communication is done. All processors evaluate the *lval* and the *rval* of the assignment for their side-effects, but only the processor that holds the target location performs the assignment.

We use special-case analysis to optimize two idiomatic C* expressions. The first is the assignment of a mono from within a conditional statement that selects only one virtual processor. The programmer intends to broadcast a value from a particular processor and we would like to implement it as such (rather than as a reduction and a broadcast). We check for expressions

that involve testing the virtual processor offset (expressed: `this-m`, where `m` is the domain array) against a constant.

The second idiomatic expression is called a tournament.

```
if (ABS(x) == (?>= ABS(x)))
    pivot = this - m;
```

The fragment above is from the calculation of the pivot row in a Gaussian elimination program. The programmer is not interested in the value resulting from the unary maximum (`?>=`) reduction operation. Rather, the programmer is interested in which virtual processor owns the "winning" value. A straightforward implementation would perform a reduction to calculate the maximum absolute value and a second reduction to update the mono variable `pivot`. By detecting this as a special case, we can call a special "tournament" communication primitive that maintains the virtual processor number of the "winning" value as the first reduction is performed. Both the result and the corresponding virtual processor value end up on all processors upon completion of the tournament primitive, saving the cost of the second reduction in the straightforward translation.

Our catalog of strength reduction optimizations is complemented by optimizations that combine communication operations. The goal is to reduce the total message latency. In order to combine communications, the optimizer attempts to move read operations backward through the program (toward the beginning) and attempts to move write operations forward through the program (toward the end). Data can be read earlier if it is known that no statements altering the data will execute on the source processor prior to the reader using the copied data. Write operations can be delayed until the data is actually needed. This message combiner was originally implemented in our prototype compiler and is described more fully in [4].

3.5 Implementing the Run-Time Routing Library

The routing library has undergone significant changes in the past two years. These changes have dramatically improved the communication efficiency of the compiled C* programs. We have added new routines and improved existing routines by giving them the ability to route vectors, by making use of unbuffered communication primitives, by overlapping communications with local data shuffling, and by rewriting key data manipulation loops in assembly language.

We have introduced new functions to the routing library to improve the performance of C* programs executing certain data parallel programming idioms. For example, we have added a new function called `tournament` to support the tournament idiom.

One of the principal weaknesses in the original routing library was the amount of time spent copying data to set up and break down messages. In many cases the data-copying time far exceeded the time spent sending the message to another processor. We have addressed this problem in three ways. First, we changed the format of the data packets. Formerly we had one data packet for each four-byte value. In the new communication library the data packets contain addresses and lengths, not values. If a virtual processor is copying a vector to or from another virtual processor, then the transfer can be represented by a single data packet. Less time is spent assembling the data packets for the routing function, and less time is spent inside the routing function constructing the message. The second change to reduce the amount of copying is the use of unbuffered read and write operations. When the original routing library was written, VERTEX, the NCUBE 3200's node operating system, did not support unbuffered reads and writes. All outgoing messages were copied from user space into a system buffer before being send down the DMA channel. Likewise, all incoming messages were stored in the system

buffer before being copied into user space. Now that VERTEX supports unbuffered reads and writes, we have made use of this capability in several of our functions. Lastly, we have rewritten many of the routing functions so that they now do a better job overlapping communications with internal data movement.

Some of the functions in the routing library do a significant amount of preprocessing. For example, it is not unusual for 25% of the time spent performing a random write to be dedicated to the initial bucket sort of the data packets. We have found that, as a rule, recoding key sections of code in assembly language reduces their execution time by 50%.

Finally, we are exploring ways to implement a true two-way communication facility. Although every pair of adjacent processors on the NCUBE share two bit-serial DMA channels (one for each direction), the three-step handshake protocol built into VERTEX effectively limits communications to one direction at a time. Most of the functions in the routing library are based upon the exchange of data among adjacent processors along the various dimensions of the hypercube. Since VERTEX does not support a simultaneous exchange of data, these routines take longer to execute than they ought to. We plan to modify VERTEX to allow a new exchange communication primitive. In some instances this primitive will allow us to reduce the execution time of the communication routines by 20% or more.

3.6 Implementing Pointers

Our current compiler provides full support for pointers, allowing the convenient use of standard C library routines that accept pointers as parameters or return pointers; and distinguishing, for efficiency, "local" pointers from "non-local" pointers.

The compiler requires that the target of a pointer be categorized at compile time. A C* pointer either points to a domain element, points to a mono, points to a poly, or points locally. A pointer to a local is defined by context: declared within sequential code the pointer must point to a mono; declared within a virtual processor the pointer must point to the memory of that same virtual processor.

The C* keywords `mono` and `poly` are used syntactically like the type qualifiers `const` and `volatile` defined in the ANSI C standard. "Local" is not a keyword in C*, but the default qualifier for a pointer declaration is "local". In a declaration of multiple levels of indirection, the innermost pointer inherits its qualifier from its predecessor in the type chain.

Declaring a pointer to be local informs the compiler that the target of the pointer can always be retrieved using only a local memory fetch. The concept of a pointer to local also allows the convenient use of many standard C library routines in both sequential and parallel contexts within a C* program. For instance, the standard `strlen` routine can always be called and passed a pointer to a local `char`. However, if the string to be examined is not local or mono, then a special C* version of the library routine must be provided and called.

Assignments of pointers are tightly checked to ensure that the compile-time categorization of pointers will still be valid at run time. The only legal pointer conversions are the promotion of a pointer to local to a pointer to poly; and, from within sequential code, the promotion of a pointer to mono to a pointer to local.

4 Execution of C* Programs

To illustrate the performance of our C* compiler, we present benchmark results from five programs: Mandelbrot set computation (*mandel*); computing the number of relatively prime pairs found in an integer range (*relprime*); matrix multiplication (*matmult*); Warshall's algorithm

program	problem size	seq C	C*	speedup
matmult	64 × 64	32,729	3225	10.15
matmult	128 × 128	273,268	15,288	17.87
matmult	256 × 256†	2,664,968	92,114	28.93
matmult	512 × 512†	20,778,216	655,964	31.68
sieve	upto 1.6M†	213,000	6378	33.40
sieve	upto 4.8M†	690,000	14,907	46.29
sieve	upto 8.0M†	1,173,000	22,930	51.16
warshall	64 × 64	16,673	1681	9.92
warshall	128 × 128	132,748	6166	21.53
warshall	256 × 256†	1,086,240	31,444	34.55
warshall	512 × 512†	8,837,760	208,885	42.31
relprime	[1..128]	7554	322	23.46
relprime	[1..256]	37,396	1227	30.48
mandel	512 × 512	789,864	20,656	38.24

Table 1: Execution results for a set of benchmark C* programs running on a 64-node NCUBE 3200. The units are NCUBE 3200 clock ticks—on our system there are 7812.5 ticks per second.

(*warshall*; and the Sieve of Eratosthenes (*sieve*). Table 1 contains speedup data gathered while running these programs with varying problem sizes on a 64-processor NCUBE 3200. Speedup reported for the problem sizes marked with a † is *scaled* speedup [3] (since the problem size is too large to fit in the memory of a single processor).

The speedups reported are in some cases limited by choice of algorithm and in some cases by inefficiencies that remain in our compiler. The point of presenting this data is to demonstrate that we can execute programs written in a high-level parallel programming language on a real parallel machine and get significant speedup.

Moreover, our compiler is designed to be rugged and will accomodate realistic programs. In the next few months we expect to be able to report execution results for a set of nontrivial benchmarks. We are currently working on implementations of irregular mesh computations, the SIMPLE hydrodynamics simulation from Lawrence Livermore Laboratory, and a differential equation solver that uses spectral domain decomposition.

Acknowledgements This work was supported by National Science Foundation grants DCR-8514493, CCR-8814662, and CCR-8906622.

References

[1] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *The Journal of Supercomputing*, 2:151–169, 1988.

[2] E. Felten and S. Otto. Coherent parallel C. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 440–450, ACM Press, 1988.

[3] J. Gustafson, G. Montry, and R. Benner. Development of parallel methods for a 1024–processor hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4), 1988.

[4] L. Hamel, P. Hatcher, and M. Quinn. An optimizing C* compiler for a hypercube multicomputer. In J. Saltz and P. Mehrotra, editors, *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, Elsevier, 1991. In press.

[5] H. Jordan. The Force. In L. Jamieson, D. Gannon, and R. Douglass, editors, *The Characteristics of Parallel Algorithms*, pages 395–436, The MIT Press, 1987.

[6] C. Koelbel, P. Mehrotra, and J. Van Rosendale. Supporting shared data structures on distributed memory architectures. In *Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 177–186, March 1990.

[7] P. Mehrotra and J. Van Rosendale. The BLAZE language: A parallel language for scientific programming. *Parallel Computing*, 5:339–361, 1987.

[8] E. Paalvast. *The Booster Language.* Technical Report PL 89–ITI–B–18, Instituut voor Toegepaste Informatica TNO, Delft, The Netherlands, 1989.

[9] M. Quinn, P. Hatcher, and K. Jourdenais. Compiling C* programs for a hypercube multicomputer. In *SIGPLAN PPEALS 1988, Parallel Programming: Experience with Applications, Languages, and Systems*, pages 57–65, July 1987.

[10] A. Reeves. Parallel Pascal: An extended Pascal for parallel computers. *Journal of Parallel and Distributed Computing*, 1:64–80, 1984.

[11] A. Rogers and K. Pingali. Process decomposition through locality of reference. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 69–80, June 1989.

[12] J. Rose and G. Steele. *C*: An Extended C Language for Data Parallel Programming.* Technical Report PL 87–5, Thinking Machines Corporation, Cambridge, MA, 1987.

[13] M. Rosing, R. Schnabel, and R. Weaver. Dino: Summary and examples. In *Third Conference on Hypercube Concurrent Computers and Applications*, pages 472–481, ACM Press, 1988.

[14] G. Sabot. *The Paralation Model.* The MIT Press, 1988.