Crystal is a functional language in which programs resemble concise, formal mathematical definitions, without explicit communication commands. The Crystal compiler transforms this very-high-level description into an assemblage of concurrent programs, each coded in some target machine language (e.g., any of the concurrent versions of C, FORTRAN, or LISP) with communication commands that establish its interaction with programs on other processors of the machine. At the center of these transformations are the concepts of *granulization* and *clustering* which automatically break down a Crystal program into many pieces of independent tasks. Granulization is based on two innovations in the language Crystal: (1) to treat all data as functions over some base domain (or index set), giving the generality and flexibility needed in expressing any parallel operation over any network and allowing the compiler to distribute automatically data structures as well as tasks over the network, and (2) to assign a process to each element in the base domain and exploit parallelism on the data level, as well as the expression level. Crystal provides powerful operators, or parallel programming idioms (a la APL), that have guaranteed efficiency in their parallel implementations. The Crystal compiler is capable of generating efficient code for machines with different granularities of parallelism. The Crystal run-time system [4] further ensures the performance of target programs over a spectrum of machine granularity. Together they resolve the portability issue in parallel programming.

Very-high-level Parallel Programming in Crystal

Marina C. Chen

Research Report YALEU/DCS/RR-506
December 1986

Work supported in part by the Office of Naval Research under Contract No. N00014-86-K-0296.
Approved for public release: distribution is unlimited.

Very-high-level Parallel Programming in Crystal

Marina C. Chen*

Abstract. Crystal is a functional language in which programs resemble concise, formal mathematical definitions, without explicit communication commands. The Crystal compiler transforms this a very-high-level description into an assemblage of concurrent programs, each coded in some target machine language (e.g., any of the concurrent versions of C, FORTRAN, or LISP) with communication commands that establish its interaction with programs on other processors of the machine. At the center of these transformations are the concepts of *granulization* and *clustering* which automatically break down a Crystal program into many pieces of independent tasks. Granulization is based on two innovations in the language Crystal: (1) to treat all data as functions over some base domain (or index set), giving the generality and flexibility needed in expressing any parallel operation over any network of distributed data, and (2) to assign a process to each element in the base domain and exploit parallelism on the data level, as well as the expression level. Crystal provides powerful operators, or parallel programming idioms (a la APL), that have guaranteed efficiency in their parallel implementations. The Crystal compiler is capable of generating efficient code for machines with different granularities of parallelism. The Crystal run-time system [4] further ensures the performance of target programs over a spectrum of machine granularity. Together they resolve the portability issue in parallel programming.

1 Introduction

One of the most critical problems in parallel processing today is that of programming parallel machines. The difficulty lies in task decomposition: how to partition a given task into pieces, one for each processor, so that it can be accomplished by the cooperation of many processors in parallel. There have been two main approaches: (1) programming in a conventional sequential language, and relying on a parallelizing compiler to generate code for parallel machines (as in numerical computing) or relying on a parallelizing interpreter and run-time support for dynamically spawning parallel processes (as in functional programming); and (2) devising a parallel programming language and expressing parallelism explicitly in a program.

The first approach has the advantage that programs can be written in familiar languages and existing programs can be transformed by parallelizing compilers for execution on the new

*Department of Computer Science, Yale University, New Haven, CT 06520. chen-marina@yale. Work supported in part by the Office of Naval Research under Contract No. N00014-86-K-0296.

machines. However, the parallelism discovered this way is limited by the algorithm embodied by the program. It is unlikely that the transformations provided by the parallelizing compiler are sophisticated enough for the task of redesigning programs better suited for parallel processing. Such redesigning is necessary for using parallel resources to their full potential. Take the problem of sorting as an example. Consider parallelizing a quicksort program, which is a very good sequential solution. This can be done by spawning a process for each of the two recursive calls to quicksort. The time complexity is indeed improved from $O(n \log n)$ in the sequential version to $O(n)$ (since $O(n)$ comparisons are needed at the top level and the number of comparisons is halved at every level thereafter) by using $O(n)$ independent parallel processes. However, what can be achieved by various parallel sorting networks (e.g., [17]) with $O(n)$ processors is $O(\log^2 n)$, which is significantly faster for large n. Numerous other good sequential algorithms have the same property that they do not lend themselves to efficient parallel implementations, as exemplified by many of the newly devised parallel algorithms [5] which are considerably different from their sequential counterparts.

This point leads to the second approach — parallel programming, where parallelism is explicitly expressed in a program. This is flexible enough to be applied to either class of parallel machines (shared-memory machines or message-passing machines) as well as any kind of parallel algorithm. However, parallel programming and debugging can be extremely difficult with thousands of interacting processes. Most parallel languages, either proposed or in use, have explicit constructs for parallelism. Programmers specify how tasks should be partitioned and which ones can be run in parallel (e.g., futures in Multilisp [7,8], "in" and "out" in Linda [6]), or how processes are mapped to processors (e.g., annotation in ParAlfl [10]); or they specify explicitly the communication between processes (e.g., "?" and "!" in CSP [9]). But specifying communication is very tricky because it requires the programmer to keep track of both the processor's own state and its interactions with other processes, and explicit task decomposition by the user will yield inefficient code for a large class of problems for which an efficient decomposition cannot be known until run-time. In fact, early experience with programming machines such as the Intel iPSC reveals that the burden of specifying task decomposition and communication can be so great that it discourages extensive experiments on load balancing via different task decompositions.

A critical research question raised here is: can a parallel program be written in a highly abstract form such that the detailed interactions among processes in space and time are suppressed, and yet it is still possible to generate efficient code for an assemblage of communicating processors? The seemingly conflicting goals of *ease of programming* and *efficient target code* can be achieved. In this paper we will give an illustration of programming in Crystal, its parallel interpretation that yields automatic task decomposition, and the organization of the Crystal compiler. The Crystal run-time system is described in another presentation [4] of this proceedings.

The rest of the paper is organized as follows. Section 2 contains some example program segments. Section 3 gives a more formal description of the syntax and semantics of the language. Section 4 addresses the issue of parallelism and how to interpret a Crystal program as a collection of parallel processes. Section 5 describes operators for programming abstraction and the properties these operators must possess in order for them to be efficiently implemented on parallel machines. Section 6 contains a brief description of the Crystal compiler.

2 Example Program Segments

2.1 Operations over a set of elements

In problem solving, one often likes to say "the value `x` associated with some element `p` is defined as the minimum (or maximum, summation, union, etc.) of the set of all values `y(q)` for those elements `q` in set `S` such that the predicate `z(p,q)` is true." In Crystal, the above sentence translates to an equation of the form

```
x(p) = \min { y(q) | q in S, z(p,q)}
```

Below is an example excerpted from a Crystal program for computing the minimum spanning tree of a graph with vertex set `vset` and edge set `E`, where `i` is an index for iterations.

```
minWeight(v, i) =
   \ min { weight({v, u}) | u in vset, ({v, u} in E) and marked({v, u}, i)}
```

2.2 Sorting

Given an input tuple x, this program segment computes a tuple y of elements in sorted order. This program is based on a mesh-of-trees parallel sorting algorithm given in [17], but the Crystal version is portable to other architectures, such as the hypercube.

```
v(i, j) = << x[i] > x[j] -> 1,
             else -> 0
          >>
rank(i) = \+ {v(i, j) | j in 0:n-1}
w(i, j) = << rank(i) = j -> x[i],
             else -> 0
          >>
y = [( \+ {w(i, j) | i in 0:n-1} ) | j in 0:n-1]
```

For instance, if `x = [2, 5, 0, 11, 6, -1]`, then the values of function `v` can be displayed as an array (below left) with, say, row index `i` and column index `j`. The function `rank` has the values 2, 3, 1, 5, 4, 0, for `i`$= 0, 1, \ldots, n-1$. The function `w` is displayed at right.

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 1 & 1 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 0 & 2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 11 \\ 0 & 0 & 0 & 0 & 6 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Finally, the tuple `y = [-1, 0, 2, 5, 6, 11]`.

2.3 Parallel prefix

The **scan** operator is a familiar one in APL. Let tuple `u` be defined as `u = [1, 2, 3, 4, 5]`. Then the tuple `v` defined as `v = scan(u, +)` has the value `[1,3,6,10,15]`. Since **scan** is used extensively and has an efficient implementation, it is supported as a primitive operator in Crystal. However, it can be defined in Crystal as follows:

```
scan=(v, f) = u
  where u[i]= w(i, floor(log(i)))
    where w(i,j) =
            << j=0 -> v[i],
               j>0 -> << j <  2^(j-1) -> w(i,j-1),
                         j >= 2^(j-1) -> f(w(i - 2^(j-1), j-1), w(i,j-1)) >> >>
```

The above program is just a description of the parallel prefix operator [11]. Note that n-ary operations over a set of n elements (Section 2.1) can be done using the parallel prefix operation. The value `x(p)` of Section 2.1 is just the last element of the tuple `scan([y(q) | q in S, z(p,q)], min)`. However, it is more efficient (by a logarithmic factor) to implement these directly.

2.4 Solving linear systems

The following segment of code defines a scalar value **scalar_a** and a vector `q` which are computed iteratively, where `i` is the index for iterations. Matrices `A` and `B`, vectors `q0`, `r`, and `s` are given as inputs. Subroutines for computing inner product **inner**, vector addition **vectoradd**, matrix vector multiplication **mmult** and forward solve **fsolve** are defined in the subprogram after the keyword **where**, in which `i` and `j` are used for indexing matrix and vector elements. The Crystal compiler

will interprets i in the main program as a time index, and i and j in the subprogram as space indices.

```
scalar_a(i)= num/den where (num = inner(r, q(i))
                            den = inner(q(i), q(i)))
q(i)= << i = 0 -> fsolve(B, mmult(A, q0)),
         i > 0 -> vectoradd(s, q(i-1), scalar_a(i-1))
      >>
where (
      vectoradd(vec1, vec2, scalar) = [ vec(i) | i in 1:n ]
      where vec(i) = vec1[i] + scalar * vec2[i]
      inner(vec1, vec2) = \+ { vec1[i] * vec2[i] | i in 1:n }
      fsolve(B, y) = [x(i) | i in 1:n]
            where x(i) = y[i] -
            (\+ { B[i,j] * x(j) | j in 1:i-1 } )
      mmult(A, p) = [ap(i) | i = 1:n]
            where ap(i) = ({ A[i,j]*p[j] | j = 1:n }
      )
```

2.5 Particle membership

Given a list of particles **particleList**, this program segment partitions the list into 4 sublists of particles according to their locations. A particle **k** belongs to the list for box **j** if it is closest to the center of box **j**. The partition is done by first computing the square of the distance from the xy-coordinates of the particle **particleList(k).x** and **particleList(k).y**, to those of the center of the box **box(j).x** and **box(j).y**. The function **membership** computes for each particle **k** which box it belongs to. Conversely, a list of particles for each box is obtained by the function **particleInBox**.

```
distSqParticleToBox(k,j) =
    distSq(particleList(k).x, particleList(k).y, box(j).x, box(j).y)
    where distSq(x1, y1, x2, y2) = power((x1-x2),2.0) + power((y1-y2), 2.0)
membership(k) = \minarg{ [ distSqParticleToBox(k,j), j ] | j = 1:4 }
particlesInBox(j) = inverse(membership, particleList, union)(j)
    where ( inverse(f, D, mergeOp) = g
            where g(y) = \mergeOp  x | x in D, f(x) = y)
```

The binary associative operator **minarg** takes a set of pairs **[value, arg]** and returns the **arg** which has the minimum **value** over the set. The operator **inverse(f, D, mergeOp)** computes the inverse of a function **f** with domain **D**. The binary associative operator **mergeOp** specifies how the value of an inverse image should be computed when the function is not one-to-one. For instance, given a function **f(i,j) = i+j** with domain $\mathtt{D} = \{1:n\} \times \{1:n\}$, its inverse function **inverse(f, D, union)** will give a set of pairs **{(i,j) | i+j = a}** as the image of each element **a** in its domain.

For more examples of Crystal programs, such as dynamic programming, LU-decomposition, matrix multiplication, and numerous toy examples, see [1,2,3].

3 Syntax and Semantics of Crystal

Crystal is a functional language that uses set notation, similar to SASL [16]. Syntactically, it uses infix operators to make programs as readable as familiar mathematical notation. Semantically, it has the standard fixed-point semantics.

$$F_1(\mathbf{v}) = \begin{cases} p_{11}(F_1(\tau_{111}(\mathbf{v})), F_2(\tau_{112}(\mathbf{v})), F_2(\tau_{113}(\mathbf{v})), \mathbf{x}_{114}(\mathbf{v})) \rightarrow \\ \quad \phi_{11}(F_1(\tau_{111}(\mathbf{v})), F_2(\tau_{112}(\mathbf{v})), F_2(\tau_{113}(\mathbf{v})), \mathbf{x}_{114}(\mathbf{v})) \\ p_{12}(F_1(\tau_{121}(\mathbf{v})), F_2(\tau_{122}(\mathbf{v})), \mathbf{x}_{123}(\mathbf{v})) \rightarrow \\ \quad \phi_{12}(F_1(\tau_{121}(\mathbf{v})), F_2(\tau_{122}(\mathbf{v})), \mathbf{x}_{123}(\mathbf{v})) \end{cases} \quad (1)$$

$$F_2(\mathbf{v}) = \begin{cases} p_{21}(F_1(\tau_{211}(\mathbf{v})), F_2(\tau_{212}(\mathbf{v})), \mathbf{x}_{213}(\mathbf{v}), \mathbf{x}_{214}(\mathbf{v})) \rightarrow \\ \quad \phi_{21}(F_1(\tau_{211}(\mathbf{v})), F_2(\tau_{212}(\mathbf{v})), \mathbf{x}_{213}(\mathbf{v}), \mathbf{x}_{214}(\mathbf{v})) \\ p_{22}(F_1(\tau_{221}(\mathbf{v})), F_1(\tau_{222}(\mathbf{v})), F_2(\tau_{223}(\mathbf{v})), \mathbf{x}_{224}(\mathbf{v})) \rightarrow \\ \quad \phi_{22}(F_1(\tau_{221}(\mathbf{v})), F_1(\tau_{222}(\mathbf{v})), F_2(\tau_{223}(\mathbf{v})), \mathbf{x}_{224}(\mathbf{v})) \end{cases} \quad (2)$$

Figure 1: The general form for a system of two recursion equations.

3.1 Functions

There are two ways a given function F can be used in a Crystal program. The first is the conventional one of using functions as a way of abstracting detailed operations, such as **vectoradd**, **scan**, **distSq**, and **inverse** in the examples above. The second way is new, using functions as a way for describing data structures. For example, the functions `w` and `v` in sorting, the functions **q(i)**, and **scalar_a(i)** for representing a vector and a scalar, and function **particleList** for a list of particles. Treating all data as functions over some base domain (or index set) instead of defining data structures separately gives the language the generality and flexibility needed in expressing any parallel operation over any network of distributed data.

3.2 Constructors

Three constructors are used in Crystal: sets, ordered tuples, and records, as shown in the above examples. These constructors are familiar either in conventional mathematical descriptions or in data processing tasks. They are essential for concise, clear, and intuitive expressions.

3.3 Syntactic constructs: a more formal description

Crystal syntax is quite intuitive since it very much resembles conventional mathematical notation. Formally, a Crystal program consists of a system of recursion equations. Figure 1 shows such a system. In describing the syntactical parts of recursion equations, three indices i, j, and k are used: the first index i is used for numbering the equations, the second index j for numbering the conditional branches within a given equation, and the third index for the number of occurrences of functions and/or constants within a given branch.

The system shown in Equations (1) and (2) defines two left-hand side functions F_i where $i = 1, 2$. Function F_i can be a tuple of functions or a field of a record of functions. Similarly, set notation may be used to express all arguments on the right hand side of each equation, as in the example of Section 2.1. Each equation can be defined by several conditional branches, such as the two cases satisfying boolean predicate p_{1j} for $j = 1, 2$ in the first equation. Certainly, each case can be defined by nested levels of conditionals. Any left hand side function value $F_i(\mathbf{v})$ of the i'th equation may depend on some *mutually recursive* function values $F_{i'}(\tau_{ijk}(\mathbf{v}))$ on the right-hand side as defined by functions ϕ_{ij} and τ_{ijk}. It may also depend on some *non-recursive* function values $\mathbf{x}_{ijk}(\mathbf{v})$, where any $\mathbf{x}_{ijk}$ is a function that does not appear on the left-hand side of any equation in the system. For example, there are 3 mutually recursive functional values (for $k = 1, 2, 3$) and a single non-recursive function value $\mathbf{x}_{114}$ in case 1 of Equation 1.

3.4 Semantics

The language Crystal is functional, and has fixed-point semantics [13]. Let V_i and all other value or functional domains over which functions and predicates ϕ_{ij}, τ_{ijk}, p_{ij}, and $\mathbf{x}_{ijk}$ are defined be continuous and complete lattices. Furthermore, let these functions and predicates be continuous. Then the solution of the system of recursion equations is its least fixed-point.

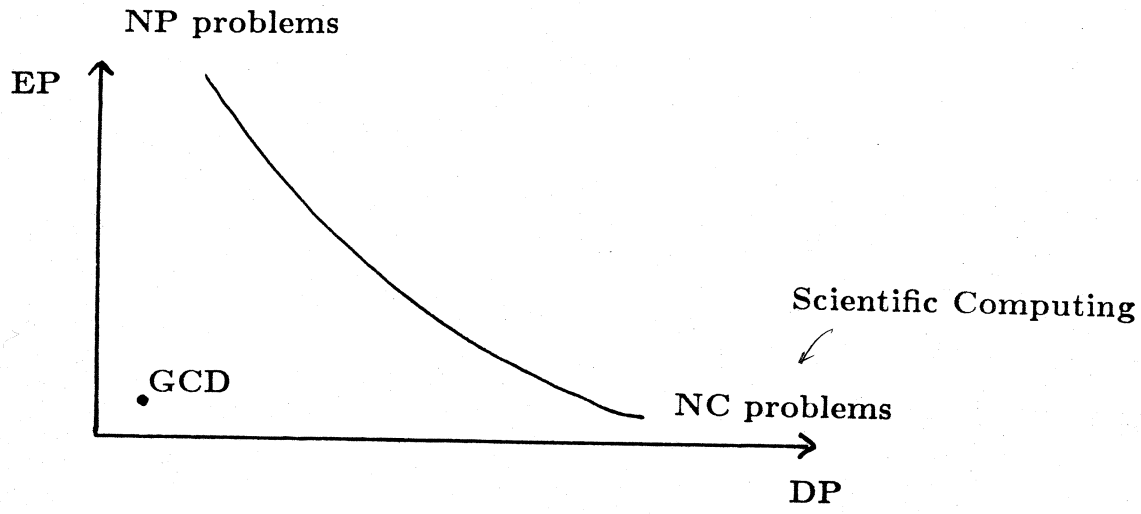

Figure 2: The distribution of problems with varying degree of DP and EP.

4 Parallelism

Parallelism in a Crystal program does not need to be specified explicitly by any particular syntactic constructs. Rather, it is a result of assigning an operational interpretation to familiar mathematical notation.

4.1 Dichotomy of operands/operations and interpretations

As described above, a Crystal function is used to describe a data structure (operands) or an operation (an operator). These two different interpretations of functions correspond to different sources of parallelism and both are supported in Crystal. An example illustrating the two types of parallelism and discussion of their relative merits can be found in [2].

4.1.1 Expression level parallelism (EP)

The first source, called expression level parallelism (EP), or applicative parallelism in [2], is well known in functional languages. This is the parallelism attainable by interpreting each function as a process. As long as two functions are not constrained by a dependency relation, the two processes can proceed in parallel. An analogous interpretation appears in logic programming languages, such as interpreting Prolog's conjuctive goals as a network of processes in Concurrent Prolog [14]. In the example of solving linear systems above, if we spawn two parallel processes that compute the values of `inner(r, q(i)` and `inner(q(i), q(i))` in the definition of `scalar_a(i)`, then we are exploiting EP. However, there is another kind of parallelism "inside of" the inner product operation on two vectors. This is called *data level parallelism.*

4.1.2 Data level parallelism (DP)

What's new in Crystal is the idea of interpreting each element of the domain of a function as a process, rather than the function itself as a process. The amount of parallelism that can be extracted this way is only limited by the size of the domain and eventually the available parallel resources. When the domain is very large, which is often the case with compute-intensive tasks, the available parallelism is correspondingly large. Parallelizing Fortran compilers tap this to a very limited extent, and one can do much better (in theory, anyway) by hand-coding everything. But until now there has no systematic way of fully exploiting such massive parallelism.

Problems in various application domains (strictly speaking, efficient algorithms for these problems), can be broadly categorized by their inherent EP and DP as shown in Figure 2. For a given amount of parallel resource, one needs to exploit as much parallelism as the problem allows. Some algorithms, such as the Euclidean algorithm for computing GCD (greatest common devisor) seem to be inherently sequential in the sense that they contain little EP or DP.

The significance of Crystal's new interpretation of functions lies in its ability to exploit DP, which is essential for virtually all fast parallel algorithms for problems in the class NC [5] (polynomial number of processors and polylogarithmic time complexity).

Crystal supports both DP and EP for a spectrum of problems lying on the continuum: predominantly EP for the class of problems clustered in the upper left corner of the figure, EP on top of DP for those along the middle part of the curve, and predominantly DP for those on the lower right corner. For the current generation of parallel machines, except for the Connection Machine, the potential of DP cannot be fully exploited due to the communication/computation ratio of the processors. As parallel computer technology progresses and the communication latency between processors becomes less, this will change.

4.2 Operational interpretation of DP

Since the idea of EP is well known only DP will be treated here. Consider a system of recursion equations as shown in Figure 1. Each element $\mathbf{v}$ in the domain V_i is interpreted as a process. Each of the mutually recursive functions F_i corresponds to the output of a process. A function ϕ_{ij}, called a *local processing* function, describes a part of the functionality of element $\mathbf{v}$ that produces output value $F_i(\mathbf{v})$. Function τ_{ijk}, called a *communication function*, describes which other element $\mathbf{u} = \tau_{ijk}(\mathbf{v})$, element $\mathbf{v}$ should receive its input. Function $\mathbf{x}_{ijk}$, called an *input function*, describes the input data. Predicates p_{ij}, called *control predicates*, describe the conditions under which a given process executes some particular processing functions ϕ_{ij}, obtains data from other processes defined by particular communication functions τ_{ijk}, and receives some particular inputs described by the input functions $\mathbf{x}_{ijk}$.

5 Programming Abstraction

One of the goals of Crystal is to provide a set of useful idioms for developing fast parallel algorithms. To this end, Crystal borrows several operators from APL, many similar to those used in Connection Machine Lisp [15]. In the context of parallel programming, in particular when different granularities are considered, these operators have two properties of particular interest:

1. *Granulizability.* Each operator can be implemented on machines of logarithmic diameter with near-linear speedup, independent of the machine granularity.

2. *Completeness.* There is a subset of Crystal operators which is complete in the sense that algorithms for all problems in NC can be programmed using these operators. A minimum such set consists of the **scan** operator, loop of polylogarithmic number of iterations, and constant-time local computations .

The granulizability property ensures portability over a spectrum of machines. The completeness property guarantees the power and generality of the language.

6 Organization of the Crystal Compiler

Figure 3 shows the organization of the Crystal compiler. The lexer and the parser are conventional. The (static) mapping generator performs a transformation that maps each index appearing in the source program to either time (a loop index) or space (logical process-id's). For more sophisticated mapping [1], a linear combination of indices in the source program is mapped to the time index in the implementation.

The granulizer performs task decomposition according to DP and EP, regardless of the machine granularity. It generates as many logical processes (represented as a parse tree) and *inter-process*

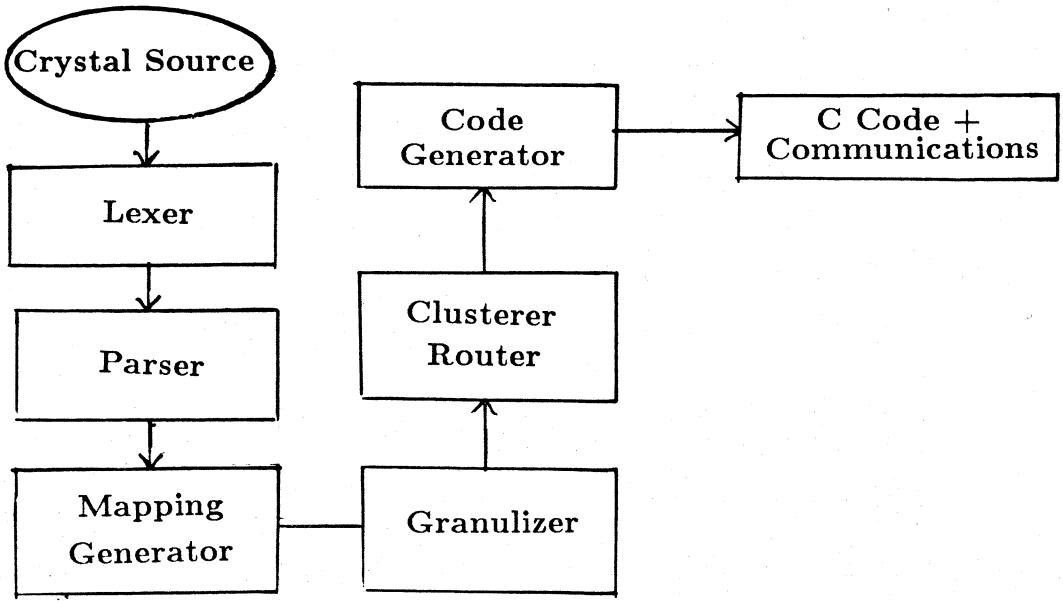

Figure 3: The organization of the Crystal compiler.

communications as possible, which contains the maximum amount of parallelism possible in a given program.

The clusterer then combines many logical processes into a single *nodal parse tree* to fit the particular granularity of the target machine. In the meantime, it eliminates those inter-process communications that are inside the same node and package those that go outside of the node. Furthermore, it determines efficient routing for these outgoing communications [12].

At this point, the original source code has been transformed to a collection nodal parse trees, one for each processor, plus the communications. Compiling a nodal parse tree to target sequential code on each processor requires no more than conventional code generation. Thus completes the compilation process.

One extremely important and interesting issue is how the clusterer chooses the appropriate collection of logical processes to combine into a single node. This is discussed in a companion paper [4] in this proceedings.

7 Concluding Remarks

This paper demonstrates the viability of programming for parallelism without user-specified task decomposition or explicit inter-process communications. The compiler is responsible for automatically breaking down a Crystal program into many pieces of independent tasks and generating efficient target code. At the center of these transformations are the concepts of *granulization* and *clustering*. Crystal also provides a set of high-level operators which can be supported efficiently on parallel machines and are powerful enough for expressing all fast parallel algorithms. Together with conventional mathematical notation, these high-level operators help to bring programming, debugging and testing to a new conceptual level. Before the advent of parallel machines, such operators wouldn't be supported in a sequential language (APL is an exception) because they are too expensive to implement. But now they can be supported very efficiently — polylogarithmic time complexity and always near-linear speedup — on parallel machines. As larger and larger machines become available, languages that exploit only EP will soon reach their limit and languages with

explicit communication will become still more difficult to use, while Crystal will be able to exploit parallelism to the limit of the available DP.

Acknowledgement. I would like to thank Alan Perlis for many inspiring discussions, and Neil Immerman on the subject of the completeness of Crystal operators. My thanks also go to Joe Rodrigue for his comments and suggestions on the manuscript.

References

[1] M. C. Chen. A design methodology for synthesizing parallel algorithms and architectures. *Journal of Parallel and Distributed Computing*, December 1986.

[2] M. C. Chen. A parallel language and its compilation to multiprocessor machines. In *The Proceedings of the 13th Annual Symposium on POPL*, January 1986.

[3] M. C. Chen. Placement and interconnection of systolic processing elements: a new LU decomposition algorithm. In *Proceedings of the IEEE International Conference on Computer Design: VLSI in Computers*, pages 275–281, October 1986.

[4] M. C. Chen and Joel Saltz. A multi-level parallel programming environment. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, September 1986.

[5] Stephen A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, (64):2–22, 1985.

[6] David Gelerter. Linda and friends. *IEEE Computer*, August 1986.

[7] Jr. Halstead, Robert H. Multilisp: a language for concurrent symbolic computation. *ACM Transaction on Programming Language and Systems*, October 1985.

[8] Jr. Halstead, Robert H. Parallel symbolic computing. *IEEE Computer*, August 1986.

[9] C.A.R. Hoare. Communicating sequential processes. *Communication of ACM*, 21(8):666–677, 1978.

[10] Paul Hudak. Para-functional programming. *IEEE Computer*, August 1986.

[11] R. E. Ladner and M. J. Fischer. Parallel prefix computation. *JACM*, (4), October 1980.

[12] Zhijung Mu and M. C. Chen. Communication efficient distributed data structures on hypercube machines. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, September 1986.

[13] D.S. Scott and C. Strachey. Toward a mathematical semantics for computer languages. In J. Fox, editor, *Proceedings of the Symposium on Computers and Automata*, pages 19–46, Polytechnic Institute of Brooklyn Press, New York, 1971.

[14] Ehud Shapiro. Concurrent prolog: a progress report. *IEEE Computer*, August 1986.

[15] Guy L. Steele Jr. and W. Daniel Hillis. Connection machine lisp: fine-grained parallel symbolic processing. In *Proceedings of the 1986 Symposium on Lisp and Functional Programming*, pages 279–297, 1986.

[16] D. A. Turner. *Recursion Equations as a Programming Language*, pages 1–28. Cambridge University Press, 1982.

[17] Jeffrey D. Ullman. *Computational Aspects of VLSI.* Computer Science Press, 1984.