Data parallelism arises from the application of a single operation to a large data set. The objective of this paper is to illustrate how data parallelism can be successfully integrated into a very high-level, machine independent functional language. At the foundation, a macro-parallel abstract machine is devised and it is combined with the substitution model of functional languages. Features such as data parallelism, functional composition, abstraction, and recursion, that give the full power of parallel algorithm specification are unified and supported in a single language, Crystal. A new procedure call mechanism for programming languages, called *hyper-stack*, is devised for calls to data parallel modules.

Can Parallel Machines be Made Easy to Program?
A Data-parallel Model for Functional Languages

Marina C. Chen

Research Report YALEU/DCS/RR-556
August 1987

Work supported in part by the Office of Naval Research under Contract No. N00014-82-K-0184 and N00014-86-K-0564. Approved for public release: distribution is unlimited.

Can Parallel Machines be Made Easy to Program? A Data-parallel Model for Functional Languages

Marina C. Chen
Department of Computer Science, Yale University
New Haven, CT 06520.
chen-marina@yale.edu

Abstract. *Data parallelism* arises from the application of a single operation to a large data set. The objective of this paper is to illustrate how data parallelism can be successfully integrated into a very high-level, machine independent functional language. At the foundation, a macro-parallel abstract machine is devised and it is combined with the substitution model of functional languages. Features such as data parallelism, functional composition, abstraction, and recursion, that give the full power of parallel algorithm specification are unified and supported in a single language, Crystal. A new procedure call mechanism for programming languages, called *hyper-stack*, is devised for calls to data parallel modules.

1 Introduction

The rapid development in large scale parallel systems presents a new challenge in software: how to program an ensemble of hundreds, thousands, or even millions of autonomous, yet closely coupled processors? Can parallel programming be made at least as easy as sequential programming? The diverse architectures of existing multiprocessors — SIMD versus MIMD, message-passing versus shared memory, fine-grained versus coarse-grained, mesh or ring versus hypercube or butterfly interconnection topologies, complicate the matter further: is portability of parallel software ever going to be possible?

Functional languages are high-level and do not seem to be biased, in particular, towards sequential implementations. Much effort has been made in the realm of functional and data flow languages (see summary in [26]) for discovering latent parallelism in programs. Due to the freedom from side-effects, functional calls that are not interdependent can be executed in parallel. But whether by combinator reduction [10] or data flow processing, parallelism is obtained by decomposing a program into functional blocks (serial combinators, adders, multipliers, merge operators, etc.), with different threads of control. However, this type of parallelism, called *control parallelism*, is orthogonal to the parallelism arising from the application of an operation to a large data set. Hence languages based on this principle do not parallelize problems with large quantities of data well without resorting to some kind of explicit specifications by programmers (such as the use of annotations in [11]).

Since it is the processing of a large quantity of data that results in large-scale parallelism, as illustrated not only by problems in scientific applications, but by every problem in the class NC [6] where a parallel solution to a problem uses a polynomial number of processors and poly-logarithmic time, *data parallelism* [8] is central to parallel processing. Two quite different programming approaches to data parallelism are currently in use: the CSP[9]-based languages, which are essentially sequential programming languages augmented with communication or synchronization commands, for a variety of shared-memory or message-passing multiprocessors, and fine-grain data parallel languages including *Lisp, C*[22], and CM-Lisp[25] for the Connection Machine.

The objective of this paper is to illustrate how data parallelism can be successfully integrated into a very high-level, machine independent functional language. At the foundation, a macro-parallel abstract machine is devised and it is combined with the substitution model of functional languages. Features such as data parallelism, functional composition, abstraction, and recursion, that give the full power of parallel algorithm specification are unified and supported in a single language, Crystal. A new procedure call mechanism for programming languages, called *hyper-stack*, is devised for calls to data parallel modules.

The paper is organized as follows: Section 2 discuss two types of data parallel models: *micro-parallel* and

macro-parallel, and the relationship between macro-parallel model and the substitution model of functional languages. Section 3 describes a macro-parallel abstract machine, and issues concerning its implementation by a variety of different architectures. Section 4 says a few words about Crystal. The interpretation of the macro-parallel sub-language of Crystal in terms of the abstract machine is presented. The interpretation is a truthful implementation of the mathematical definition of the least fixed-point of system of recursion equations. Next, the hyper-stack mechanism for general procedure calling is described. In Section 5, example Crystal programs which illustrate the interpretation of very-high-level programs in terms of the macro-parallel/substitution model are presented. Section 6 gives a few concluding remarks.

2 Model

2.1 Micro-parallelism vs. Macro-parallelism

One approach to programming parallel systems builds upon conventional sequential programming by augmenting explicit communication or synchronization commands to a given sequential language, as illustrated by Hoare's CSP [9], and later a variety of parallel C and Fortran supported on various multiprocessors. With this approach, a given parallel system will be described by a collection of sequential programs, each written for a given processor, with communication commands for passing messages to and from other processors or synchronization commands for accessing a shared memory. This approach to parallel programming takes a microscopic view of a system: a system is described from the standpoint of each individual processor, and therefore is called *micro-parallel programming*.

Micro-parallel programming requires that a programmer, at the time of programming, have in mind the detailed behavior of each individual processor and its interaction with others, either by message passing or via shared memory. Since composition of programs occurs between processors in micro-parallel programming, the complexity of dealing with such interactions grows combinatorially with the number of processors. Hence this viewpoint may be appropriate in the case of tens of coarse-grained, loosely coupled processors, but rapidly becomes ineffective as the number of processors grow larger into the range of thousands or beyond.

In contrast, another style of parallel programming, called *macro-parallel* programming, solves a problem by reducing it to a sequence of steps where each step involves a parallel operation performed by an aggregate of processors. In each step, the macroscopic behavior of the entire system as a whole is specified, with a global coordinate system suitable for describing the problem at hand. This"horizontal"partitioning of a problem into a sequence of parallel steps is the most natural because human reasoning works step by step; parallel reasoning only occurs within each step which is designed to be simple and straightforward enough to be dealt with.

The power of the macro-parallel model is best illustrated by programming in *Lisp or C* on the Connection Machine; experience has shown that it is much easier and less error-prone than programming multiprocessors where the programming model is micro-parallel. Clearly, although micro-parallelism may be appropriate when the number of processors are small and the interactions are infrequent, macro-parallelism wins in the case of large scale parallelism.

2.2 Macro-Parallel Model and Functional Substitution Model

But the macro-parallel model by itself is not enough for general-purpose programming; we need abstraction. We want to abstract a macro-parallel program into a function so that it can be used as a step of computation in a higher-level macro-parallel program, or so that it can be used in the same way any function is used in a functional language. For instance, a program may consist of three levels: (1) at the top level, the problem is defined with recursion, composition, etc., of a few functions operating over vectors and matrices; (2) at the middle level, each of these functions is defined in a macro-parallel fashion by scaler multiplications and additions of floating-point numbers; (3) finally, at the bottom level, the scaler multiplications and additions are implemented by a pipelined architecture, again in the macro-parallel fashion but manipulating even lower data representations, namely, bits.

To fit the macro-parallel model into the framework of the substitution model of functional languages, we need (1) to differentiate whether an argument to a function shall be interpreted by the macro-parallel model or the substitution model, and (2) to provide a black-box functional abstraction for a macro-parallel program and, (3) a substitution mechanism for calls to macro-parallel modules. In the description of Crystal below, we will show that a macro-parallel program can be described by a system of recursion equations over an index set, and thus from the type of the arguments to the functions appearing in the recursion equations (they must be indices), we can differentiate the two different ways an argument shall be interpreted, and hence determine the model. Secondly, the functional abstraction of a system of recursion equations is its least fixed-point. Hence the black-box abstractions of a parallel module can be made, and once abstracted, the module behaves as a function that can be invoked irrespective of its implementation. The substitution mechanism, called *hyper-stack*, boils down to establishing communications between the group of processors implementing the caller function to those of the callee.

As opposed to the graph reduction approach where only control parallelism is exploited, each function call in Crystal eventually reduces to calls to data-parallel modules at the lower levels. If available resources permit two control-independent data-parallel modules to proceed at the same time, then control parallelism is exploited.

3 A Macro-parallel Machine

To make sure the compiler technology we are developing can be applied to a variety of different parallel architectures and to ensure the portability of parallel software, we define an abstract machine as an intermediate target to which source programs are compiled. The abstract machine is then implemented by a given target parallel machine.

The macro-parallel machine defined below uses some suitable global coordinate system for naming processors, which in turn are mapped to the processor space of a given target machine. The values of coordinates are taken from an index set defined below:

Definition 3.1 *Let A_i be a Cartesian product $A_{i1} \times A_{i2} \times \cdots \times A_{iq}$ of finite sets A_{ij}, $1 \leq j \leq q$ and q a positive integer. The set $(A_{ij}, \sqsubseteq)$ with the binary relation "approximate" ([24]) on the elements of A_{ij} is a flat lattice. An index set A is a sum $A_1 + A_2 + \cdots + A_r$ of Cartesian products $(A_i, \sqsubseteq)$, for some positive integer r, and any subset of an index set is also an index set.*

A macro-parallel machine consists of processors each of which has a local memory space. It operates by executing a sequence of macro-parallel instructions. Instructions are of two types, local computations and communications. For convenience, a few "high-level" instructions consisting of a sequence of communication and computation steps are made available. They include instructions for allocating and deallocating processors, parallel prefix computation, etc. The components of a macro-parallel machine are defined as follows:

1. *Host:* A special processor that initiates the first instruction of a program.

2. *Processor space:* An index set A.

3. *Time steps:* The concept of time step is a logical one that corresponds each time step with either a local computation which can go on without any communication with other processors, or a communication for delivering arguments from some source processors to one or more destination processors. The duration of each time step in real time will vary depending on the particular local computation or communication involved.

   The abstract model here is a synchronous one in that all processors are operating in locked steps, where the length of a particular cycle is the maximum of the time needed for completing a local computation or a communication over all processors. The implementation of the abstract machine, however, is not necessarily synchronous; it can be an asynchronous system where each processor has its own clock and

communicates with other processors with some asynchronous protocol. In that case, time stamps can be passed along with communications to derive the implicit global time steps.

4. *Memory space within a processor:* For every $\mathbf{a} \in A$, there is a set of *local variables* $S(\mathbf{a})$.

5. *Constants distributed to processors:* Each processor may contain values c that are used by the processor but cannot be altered by it.

6. *Local computation:* For every $\mathbf{a} \in A$, there is a set of *processing functions*, where a processing function $\Phi_{\mathbf{a}}$ takes two types of arguments: *local arguments* that are values of local variables in $S(\mathbf{a})$ and *remote arguments* r that are values of variables $S(\mathbf{a}')$ of some other processor $\mathbf{a}' \neq \mathbf{a}$. A processing function results in values that update the variables in $S(\mathbf{a})$, but it cannot change the value in any other $S(\mathbf{a}')$. Let $\mathbf{s}(\mathbf{a},t) = [s_0,\ldots,s_{n-1}]$ be the tuple containing the values of all local variables of processor $\mathbf{a}$ at time step t. Similarly, let $\mathbf{c}(\mathbf{a},t)$ be the tuple of constants in processor $\mathbf{a}$ at time step t and $\mathbf{r}(\mathbf{a},t) = [r_0,\ldots,r_{k-1}]$ be the tuple containing the remote arguments of processor $\mathbf{a}$ at time step t. Then we use the notation

$$\mathbf{s}(\mathbf{a},t) = \Phi_{\mathbf{a}}(\mathbf{s}(\mathbf{a},t-1),\mathbf{r}(\mathbf{a},t-1),\mathbf{c}(\mathbf{a},t-1))$$

to describe a processing function Φ executed in processor $\mathbf{a}$ at time step t that takes the values of its local variables and remote arguments and updates the values of its local variables.

7. *Communication:* For every $\mathbf{a} \in A$, there is a set of *communication functions*, where a communication function $\Psi_{\mathbf{a}}$ takes the values of some of the local variables in $S(\mathbf{a})$ as arguments and produces i) a merge operator, and ii) *remote references*: a tuple of index and *accessing function* pairs $[\mathbf{a}_i,\sigma_i]$:

$$[merge\_op, remote\_ref] = \Psi_{\mathbf{a}}(\mathbf{s}(\mathbf{a},t-1)), \qquad \text{where } remote\_ref = [[\mathbf{a}_0,\sigma_0],\ldots,[\mathbf{a}_{m-1},\sigma_{m-1}]].$$

Each communication of processor $\mathbf{a}$ at time step t results in a remote argument r. It is obtained by applying the merge operator to a tuple of values where each value is obtained from applying accessing function σ_i to the local variables of processor $\mathbf{a}_i$ at time step $t-1$:

$$r = merge\_op([\sigma_0(\mathbf{s}(\mathbf{a}_0,t-1)),\ldots,\sigma_{m-1}(\mathbf{s}(\mathbf{a}_{m-1},t-1))]).$$

In the special case when $remote\_ref$ is a singleton $(\mathbf{a}_0,\sigma_0)$, no merge operator needs to be specified and $r = \sigma_0((\mathbf{a}_0,t-1))$.

In the above description, a communication is viewed from the processor that is to receive a remote argument. Similarly, a communication can be formulated from the standpoint of a sender. In that case, the accessing function σ_i of a remote reference $(\mathbf{a}_i,\sigma_i)$ is instead a value, and the merge operator is a copying function [1] which copies the value σ_0 to all other σ_i for $i = 1,2,\ldots,m-1$. Only the value σ_0 needs to be specified, all others are copies of the same value and the copying is done in such a way that the fan-out problem is taken care of. Similarly, in the case when remote references is a singleton, no merge operation, or copying in this case, takes place. For a communication of the sender's form, the action of the second half of a communication step that retrieves remote arguments does not happen, but a write-to-local-memory to store the value σ_i is performed.

A processor may have several communication ports which are active at the same time; in that case, there will be more than one commmunication function executed at each time step, each resulting in a remote argument. We use the convention that the number of remote arguments k should not exceed the number of communication ports which can be active at the same time. Thus all components of the remote argument tuple $\mathbf{r}(\mathbf{a},t)$ are received (or send) in one time step.

The remote references (or remote values to be sent) can be of higher-order types, i.e., they can be referring to code segments.

[1] This merge operatiion will be a copy-scan in PARIS on the Connection Machine.

Note that the activity of communication among processors in a parallel machine is not only determined by the computation of the communication function, but also is sometimes carried out with a merge of many remote arguments into a single argument (or copying a value and send it to many places).

8. *Processor allocation and de-allocation:* A processor performs a communication to a group of processors, announcing them to be allocated to execute a sub-program. Similarly, a processor can de-allocate a group of processors previously allocated.

9. *Parallel prefix operation:* Guy Blelloch [2] has proposed the scan-model as an alternative to the EREW or CRCW parallel computation models. Scan-operator, or parallel-prefix computation [16,17], takes a tuple T and a binary associative operator $\oplus$ as arguments and computes a new tuple S such that $S[i] = T[i] \oplus S[i-1]$. This operation is a powerful abstraction of what can be implemented by a sequence of communication and computation steps on the macro-parallel model. For convenience, we may consider it as a primitive operation, and denote it by $\backslash\backslash\oplus$ for any binary associative operator $\oplus$.

10. *Bounded iterations:* A given parallel computation is then a sequence of interleaved local computations and communications. If an *a priori* bound t_f for the time steps is given, then the computation starts at initial time step t_i and continues until the final time step t_f. The result of computation is contained in the final state of the machine, i.e., $\mathbf{s}(\mathbf{a}, t_f)$ for all $\mathbf{a} \in A$.

11. *Unbounded iterations:* Suppose the computation has no known *a priori* bound, and its completion depends on some termination condition P which in turn depends on the state $\mathbf{s}(\mathbf{a}, t)$ for $\mathbf{a} \in A$ of the machine — then the computation goes on until P is satisfied. The test of such condition can be realized by, say, a communication from all processors to a given processor with logical AND as the merge operator[2]. The result of computation is contained in the final state of the machine, i.e., $\mathbf{s}(\mathbf{a}, t_P)$ for all $\mathbf{a} \in A$ where t_P is the minimum time step such that P is satisfied.

3.1 SIMD versus MIMD implementations

The processing function $\Phi_{\mathbf{a}}$ and the communication function $\Psi_{\mathbf{a}}$ for every processor $\mathbf{a}$ of the abstract macro-parallel machine might be different for different processors. Their implementation on a Single Instruction Multiple Data (SIMD) multiprocessor architecture requires them to be time-multiplexed. The number of possible different processing functions and communication functions over the processors depends a great deal on applications at hand and the programming paradigms in use. In the case where there are only a few different functions and each function is fine-grained (consisting of only a few number of machine instructions), it is more efficient to step through in sequence all different functions for a given logical cycle. Otherwise, it will be more efficient to time-multiplex the complete machine instruction set for each logical machine cycle. In this case, each processing function, though maybe different from others, is broken down into a sequence of machine instructions, each of which is guaranteed to be executed at some time-slice in a given logical machine cycle. Consequently, there will be at most a constant (the size of the machine instruction set) slow-down factor for implementing the abstract machine by an SIMD machine.

To implement the abstract macro-parallel machine by an MIMD architecture, synchronization must occur at the end of each abstract machine cycle when each processor has completed its processing and communication functions. Such a synchronization mechanism is well known and the choice of a particular synchronization method depends on the characteristics of the target multiprocessor.

3.2 Message-passing versus shared-memory implementations

Upon executing its communication function, a processor of the macro-parallel machine is ready to retrieve its remote argument(s), or the result of a merge of these arguments. The macro-parallel machine is described

[2]This operation may be very expensive on some multiprocessors, hence may significantly degrade the performance of applications where a global termination condition is required. On the Connection Machine, on the contrary, this operation can be implemented by the "wire-or", which is fast, and takes a constant amount of time regardless of the number of processors involved.

as an architecture with local memory, and processors communicate by accessing other processor's memory. This model can be implemented by a message-passing machine where remote arguments of a processor are sent to it via, say, a fan-in tree of processors which merges these arguments. In order to support the communications of the abstract machine efficiently, there must be an efficient routing system [28,19,20] for the message passing machine. The way in which the abstract network of processors is embedded into a given target message-passing machine can also plays a part in the communication efficiencies[14].

For a shared-memory machine to implement the local memory model of the abstract machine is simpler: merely partition the memory so that each processor has a piece to itself. Certain data that must be duplicated in the message-passing model would not need to be duplicated in this case. The access of remote arguments by each processor is simply accessing the shared memory, while the merge of these arguments must be implemented by, for instance, a fan-in tree of processors that completes the merge in a logarithmic number of steps.

4 Parallel Interpretation of Crystal Programs

Having established a parallel programming model, the next step is to design a suitable language which matches the macro-parallel/substitution model. It turns out that we need not look very far for suitable language constructs; unconventional interpretations of familiar constructs to the macro-parallel model serve the purpose. Crystal has taken many of the constructs that are around in existing programming languages such as APL[13], SETL[7], and functional languages such as FP[1] and KRC[27], and, more broadly, conventional mathematical notations. But the language is interpreted with the macro-parallel/substitution model instead.

High-level data structures such as tuples and sets from SETL are natural for denoting a collection of ordered or unordered data objects, each of which corresponds to a piece of data distributed to an individual processor. APL has numerous aggregate operators that apply over arrays; they match the macro-parallel model well, only that the APL array structure must be made more flexible. Recurrences are widely used mathematical notations; they describe succinctly the local behavior of each point of a system as well as the interactions between different points over the entire system. This property allows recurrences, made over with a functional syntax, to be exactly the natural notation for describing the computation at each processor as well as the communication between processors of a macro-parallel model.

The above-mentioned constructs, data stuctures, and notations are unified into Crystal [4,5] with functional syntax and fixed-point semantics. A macro-parallel program is described as a system of recursion equations over an index set. Sets and tuples generated from some index set are just special forms of recursion equations. A system of recursion equations is a natural language for the macro-parallel abstract machine we have defined. Its interpretation in terms of the abstract machine is described by induction on the structure of recursion equations.

4.1 Recursion Equations over an Index Set

Below is a system of two recursion equations defined over domain V of indices:

$$
\begin{aligned}
F_1(\mathbf{v}) \text{ over } V = &\begin{cases} p_{11}(F_1(\mathbf{v}_{111}), F_2(\mathbf{v}_{112}), F_2(\mathbf{v}_{113}), \mathbf{x}_{114}(\mathbf{v})) \rightarrow \phi_{11}(F_1(\mathbf{v}_{111}), F_2(\mathbf{v}_{112}), F_2(\mathbf{v}_{113}), \mathbf{x}_{114}(\mathbf{v})) \\ p_{12}(F_1(\mathbf{v}_{121}), F_2(\mathbf{v}_{122}), \mathbf{x}_{123}(\mathbf{v}) \rightarrow \phi_{12}(F_1(\mathbf{v}_{121}), F_2(\mathbf{v}_{122}), \mathbf{x}_{123}(\mathbf{v})) \end{cases} \\
F_2(\mathbf{v}) \text{ over } V = &\begin{cases} p_{21}(F_1(\mathbf{v}_{211}), F_2(\mathbf{v}_{212}), \mathbf{x}_{213}(\mathbf{v}), \mathbf{x}_{214}(\mathbf{v})) \rightarrow \phi_{21}(F_1(\mathbf{v}_{211}), F_2(\mathbf{v}_{212}), \mathbf{x}_{213}(\mathbf{v}), \mathbf{x}_{214}(\mathbf{v})) \\ p_{22}(F_1(\mathbf{v}_{221}), F_1(\mathbf{v}_{222}), F_2(\mathbf{v}_{223}), \mathbf{x}_{224}(\mathbf{v}) \rightarrow \phi_{22}(\backslash \oplus [F_1(\mathbf{v}_{22k}) | 0 \leq k < m]) \end{cases}
\end{aligned}
\tag{1}
$$

In discussing various syntactic parts of a system of recursion equations, we use three indices i, j, and k: i for numbering the equations, j for numbering the conditional branches within a given equation and k for numbering terms within a given conditional branch.

A system is made up of any constant number of functions defined over the domain V, which is an index set. For practical purposes, it suffices to consider V as a set of integer tuples, however, for convenience, symbolic names are often used as indices. A domain V will be defined by a domain specification *domain_id* =< *domain_expr* > which specifies the ranges of each of the indices. It also serves the purpose of a shorthand for not repeating the range of elements of V in the predicates of the equation that follow.

Each equation can be defined by several conditional branches, and each branch by nested levels of conditionals. Applying F_i to any argument not in the domain, or which does not satisfy any of the predicates, will yield an undefined value.

Any function value $F_i(\mathbf{v})$ may depend on *mutually recursive* function values $F_l(\mathbf{v}_{ijk})$ and on *non-recursive* function values $\mathbf{x}_{ijk}(\mathbf{v})$, where $\mathbf{x}_{ijk}$ is a function that does not appear on the left-hand side of any equation in the system.

4.2 Semantics of Recursion Equations

A system of recursion equations over an index set has standard fixed-point semantics. Let V_i and all other value or functional domains over which functions and predicates Φ_{ij}, Ψ_{ijk}, p_{ij}, and $\mathbf{x}_{ijk}$ are defined to be continuous and complete lattices. Furthermore, let these functions and predicates be continuous. The least fixed-point of the system of such a system is taken to be the "black-box" abstraction of the macro-parallel module.

4.3 Macro-parallel Interpretation of Recursion Equations

Every element of the index set is interpreted as a processor, and for every function F, there corresponds a variable F_{local} of specified type, together with a *change* bit indicating whether its value has been changed from one iteration to the next. The system is interpreted as a loop of unbounded iterations where in each iteration, a change bit gets set if its corresponding variable is assigned a new value; it terminates when the change bit of all F_i becomes false. Note that the macro-parallel interpretation follows exactly the way the least fixed-point is defined and that the evaluation proceeds in the opposite direction as a system of recursion equations would be evaluated in conventional functional languages.

The abstract machine code at each iteration t may consist of two or more macro-parallel instructions, depending on how complex the right-hand side definition is. We describe the interpretation inductively on the structure of the right-hand side of a system of recursion equations, starting at the base case where there are no recursive calls on the right-hand-side:

1. In the system of equations (1), suppose all recursive terms were not present. Then the equations are translated into two macro-parallel instructions: (1) all non-recursive values $\mathbf{x}_{ijk}(\mathbf{v})$ are distributed to every processor $\mathbf{v}$ as constants, (2) a local computation for all processors using the constants and updates variable F_{local} and change bit gets set accordingly.

2. The next simplest form of recursion equations is that which has recursive calls, but the calls neither appear in predicates (in a conditional expression or appearing in the filter expression of a set or tuple), nor do they appear as *nested* recursive calls. In System (1), suppose all recursive terms in the predicates were not present, then the equations are translated into three macro-parallel instructions:

   (i) For each index $\mathbf{v} \in V$, constants $\mathbf{x}_{ijk}(\mathbf{v})$ are distributed and predicates are evaluated. We say that index $\mathbf{v}$ is selected by a predicate p when the right-hand side predicate p evaluates to true. When predicates are not mutually exclusive, it is up to the particular programming language to determine exactly which predicates are to select a given index.

   (ii) For each index $\mathbf{v} \in V$, it generates remote references to obtain, say, a remote argument $r = F(\mathbf{u})$, which is the value of the variable corresponding to F in processor $\mathbf{u}$ at time $t-1$. A remote argument can also be the result of a merge as in $r = merge\_op(F(\mathbf{u}_0), \ldots, F(\mathbf{u}_{m-1}))$.

   (iii) For each index $\mathbf{v} \in V$, the local variable F_{local} is modified according to the result of executing the processing function, which is applied to local variables, constants, and remote arguments of $\mathbf{v}$.

3. In general, the right-hand side of a system can be a complex structure of calls to functions, but as long as these function calls are not recursive calls to F, they are part of a computation step and are to be interpreted by the substitution model, which will be addressed by the hyper-stack mechanism in Section 4.5.

   The only interesting case we are concerned with here is when calls to F are nested. Each level of nested recursive function calls requires a communication step. Simple flow analysis will tell the sequence of communications needed in order to fetch the required remote arguments. For example, in the following Crystal program that defines Ackerman's function, two consecutive communications on the macro-parallel machine will be required if the index pair (x, y) is selected by the **else** clause: the first communication for obtaining remote argument $r_1 = A(x, y-1)$ and the second for obtaining $r_2 = A(x-1, r_1)$.

```
A(x,y) = << x=0 -> y+1,
            y=0 -> A(x-1,1),
            else -> A(x-1, A(x, y-1))  >>
```

4.4 Space-time Optimization

Definition 4.1 *A system of recursion equations defined over an index set V contains a loop index t if there exists a component t of the index tuple $\mathbf{v} \in V$ such that the value t_1 of component t of each index on the right-hand-side is always less than the value t_0 of the corresponding component on the left-hand-side, and furthermore, if the maximum difference $t_0 - t_1$ over all such pairs of indices is a constant c.*

Note that if a system of recursion equations contains a loop index, then the value of $F_i(\mathbf{v})$ won't change once it becomes defined by the iterative process to reach the least fixed-point of the system. If the difference of the left-hand side and right-hand side t values is bounded by c, then once the computation arrives at $t = t_1$, all values before $t_1 - c$ will never be accessed again. Let U be the index set with the time component removed. Hence there is no need to allocate one processor for each index tuple in V as described above, it is only necessary to allocate one processor for each index tuple in U, where each processor has c storage elements for retaining the value $F_i(\mathbf{v})$ for c time steps.

The interpretation of a system that has a loop index by the macro-parallel machine can be either a bounded iteration where the upper bound is defined by the range of the loop index t, or an unbounded iteration whose termination depends on some specified conditions. More generally, a system of recursion equations may have an implicit loop index which can be made explicit by a *space-time mapping* [4,3] of the original index set to a new index set.

The above discussion illustrates that recursion equations defined over an index set allows side-effects to the state of variables distributed in parallel processors to be described in a functional way: either by including a loop index in the description as illustrated by the example in Section 5.2, or by the implicit inductive procedure that finds the least fixed-point as illustrate by the following program for the transitive closure [12] of a binary relation [3]:

```
closure(x,y) =
    x=y or E[x,y] or exists z in V: (closure(x,z) and closure(z,y)),

!!! input data
!!! vertex set
V = { 0, 1, 2, 3, 4, 5 },

!!! binary relation represented by an adjacency matrix
```

[3] Crystal's recursion equations over an index set is closely related to the first-order inductive definitions used in [12] where the complexity of a program is captured by the language.

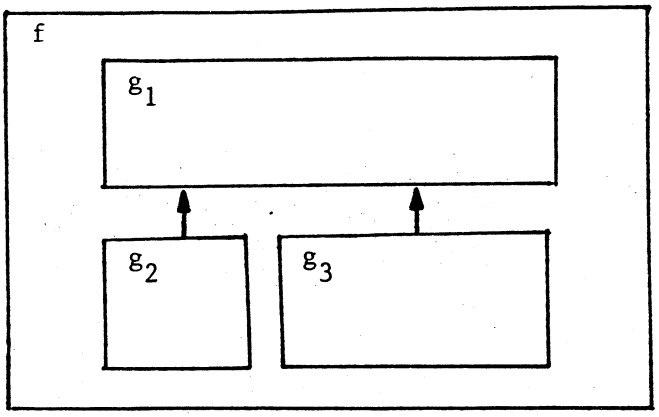

Figure 1: The hierarchy of hyper-stack frames.

```
E =
  [ [ 0, 1, 1, 0, 0, 1]
  , [ 0, 0, 1, 0, 0, 1]
  , [ 0, 1, 0, 1, 0, 0]
  , [ 0, 1, 1, 0, 0, 1]
  , [ 0, 1, 1, 0, 0, 1]
  , [ 1, 1, 1, 0, 0, 0]
  ],
```

In either case, the language processor determines the loop indices and generates a sequence of macro-parallel instructions that changes (or causes side-effect in) the state of the processors. In this sense, space-time mapping, in its own right, becomes a method of overcoming the difficulty in implementing array operations in functional languages.

4.5 Hyper-stack: a procedure calling mechanism for macro-parallel modules

Perhaps the easiest way to visualize how macro-parallel modules can work within the functional substitution model is by simulating a calling sequence. The mechanism used for procedure calls is called ***hyper-stack***, which is a generalization of *distributed* stack with a concrete spatial dimension to each stack frame.

For convenience, we describe the procedure-calling mechanism to macro-parallel module in the upward and downward phases. In fact, both the upward and downward phases can go on in different parts of the machine at the same time:

1. *Downward phase.* Each time a function f is called, a *frame* for that function is created, where the frame is flexible and can grow to the right size later. We then go into the body of f where other functions might in turn be called. For each function g invoked within function f, a frame for g is created. Graphically, g's frame is drawn within f's frame as shown in Figure 1. In f's definition, if $g_1(g_2, g_3)$ appears, then three non-overlapping frames, one for each g_i, are drawn within f's. We also draw arrows between frames at the same level for keeping track the results of one to the other. For instance, both g_2's and g_3's frame must be glued to g_1's so that result of evaluating the former can be passed to the latter. This process continues and a hierarchy of frames within frames will be created until no more recursive calls are made. In the downward phase, no processors other than those that keep track of the frames and direction of data flow between frames are allocated. Thus for a program with tree recursion, the number of processors used for this purpose is as many as the number of tree nodes. In this phase, the model described here is similar to the graph reduction model.

2. *Upward phase.* The power of this model lies in the upward phase where the number of processors allocated is proportional to the amount of data to be processed by the macro-parallel module, not

just those correspond to the tree nodes mentioned above. A frame at the bottom level is some macro-parallel module (sequential module is just a special case), and the size of the module will be specified by its domain specification which may depend on, say, the size of its input argument. We now grow this frame to reflect the size of macro-parallel module, i.e., allocating processors to make the frame concrete. At this point, the frame is set and the computation of the macro-parallel module is initiated. For frames at the same level, they must be properly "glued together" for passing the results of one to the other, i.e., actual communications are established according to the arrows drawn in the downward phase. A frame disappears after a procedure has been completed and has sent off all of its results to the caller. Processors within the frame will be deallocated. Practically, these processors are often re-used by the caller module so that the result can be kept in the same processors to minimize the communication time.

5 Examples

5.1 Almost Ascending Sequence

Having discussed the abstract machine and in general, and the way in which recursion equations are interpreted by the abstract machine, we now use an example to illustrate how a program can be"compiled" to the abstract machine. The first example is structurally very simple in that it does not really contain recursion. However, it illustrates, in particular, how sets and tuples are interpreted. It is the problem of longest almost ascending sequence [21] described as follows: Given an integer array `A` of n elements `A[0], ..., A[n-1]`: A sequence `A[p], A[p+1], ..., A[q]` is ascending if for all `i, p < i < q, A[i-1]` is less or equal to `A[i]`. An almost ascending sequence is a sequence in which there is no more than *one* possible `i, p < i < q`, such that `A[i-1]` is greater than `A[i]`. We want to find what is the length `q-p` of the longest such sequences.

A first Crystal program that solves this problem is a simple, straightforward specification of the problem:

```
! Program LAAS (Longest Almost Ascending Sequence)
! input data
A = [3, 4, 5, 3, 4, 1, 2, 3, 4, 5, 4, 5, 6, 7, 7, 5, 6, 7, 8, 9, 9],
n = ||A||,

L = \max{ q-p+1 | 0 <= p <= q < n : aas(p,q) },
aas(p,q) = || { i | p < i <= q: A[i-1] > A[i] } || <= 1,
```

In a Crystal program, the order in which definitions are given does not matter, as long as they are at the same level; subprograms are indicated by the keyword **where**. The input sequence of numbers is given as a tuple `A` whose length is `n`. The number of elements of a set or the length of a tuple is denoted by enclosing it by double bars as shown. In Crystal, a *generated* set is written in the form

```
{ exp(v) | domain(v) :  filter(v)}
```

where `exp(v)` can be an arbitrary expression of `v`, `domain(v)` specifies the domain of variable `v`, and `filter(v)` is a predicate that selects elements in the domain. An ordered tuple is defined similarly:

```
[ exp(v) | domain(v) :  filter(v)]
```

where `domain(v)` has a specified order. Any binary associative operator can be prefixed by the reduction operator `\` and results in an n-ary operator over a set or a tuple of elements. When `\op` is applied to a set, `op` is assumed to be commutative. Whereas `\op` over a tuple of elements means that the operator must be applied to the elements in the order of left to right (or first element to the last element). No commutativity is assumed in this case.

Now we want to interpret this program by the macro-parallel/substitution model. Note that the order of evaluations of the various components of a reduction operator over a generate set or tuple is *domain* →

filter → *expression* → *reduction op*. In the following, we use the convention that the local variables created in each processor will be in italics, while accessing function for retrieving its value will be the same name in boldface font. The implementation of Program LAAS consists of the following sequence of instructions:

1. The host processor allocates `0 <= i < n` "Constant" processors C_i for holding the input tuple `A`, whose length `n` is pre-evaluated by the host machine. Each processor C_i now contains the constant ai =`A[i]`.
2. The host processor allocates $n(n+1)/2$ processors R_{pq} according to the domain specification `0 <= p <= q < n`.
3. Now start evaluating `aas(p,q)` for each processor R_{pq}:
   (a) Each processor R_{pq} now allocates `q-p` processors S_{pqi} according to the domain specification `p < i <= q`.
   (b) Predicate `A[i-1] > A[i]` is evaluated in all processors S_{pqi}, stored as a boolean value `dip?`. This step takes a communication and a computation step. First, each processor S_{pqi} generates two remote references $(C_i, \mathbf{ai})$ and $(C_{i-1}, \mathbf{ai})$ to obtain the values of variable ai of processors C_i and C_{i-1}, i.e., `A[i]` and `A[i-1]`, one on each communication port respectively. Next, all processors S_{pqi} compute the predicate and store the result in `dip?`.
   (c) For those processors S_{pqi} whose `dip?` is true, evaluate the expression `i`.
   (d) Each processor R_{pq} generates remote references $(S_{pqi}, \mathbf{dip?})$ for $p < i \leq p$ with merge operator "+". The length of the set `{ i | p < i <= q: A[i-1] > A[i] }` is achieved by the merge operation, which is a summation over the value *dip*?, now converted to an integer value[4], of processors S_{pqi}. Let the result be stored in variable *len* of R_{pq}.
   (e) In each processor R_{pq}, compute the predicate $len \leq 1$ by first distributing the constant 1 to all such processors, followed by a local computation that results in the boolean value *aas*?, thus completing the phase of evaluating `aas(p,q)`. Processors S_{pqi} are de-allocated.
4. For those processors whose *aas*? is true, evaluate the expression `q-p+1` and store it in variable *len_as*.
5. The host processor now does a communication to obtain remote arguments from the variable *len_as* of those processors R_{pq} whose *aas*? is true with merge operator `max`, and assigns this value to variable L. All processors R_{pq} may be de-allocated at this point, and the program is completed.

This example illustrates perhaps the simplest form of calling a data-parallel module within a substitution model when `aas(p,q)` is called in the definition of `L`.

Program LAAS is a straightforward problem specification that is easy to write and to understand, but it is not an efficient program by the macro-parallel interpretation, because it needs a total of $O(n^3)$ processors during its course of computation. However, the computation is fast; it needs only a constant number of basic macro-parallel instructions. A much better solution to the problem follows from the observation that an almost asending sequence can be constructed by two adjacent "locally-maximum" ascending sequences, each of which is the sequence from one "dip" to the next. In the Crystal program below, a tuple `K` that keeps the indices where the dips occur is defined. The length of an almost ascending sequence is then the difference of the indices of two dips that are two positions apart, i.e. `K[i]-K[i-2]`.

```
! Improved Solution of LAAS

L =  << m <= 2 -> n,
        else -> \ max {K[i] - K[i-2] | 1 < i < m} >>
     where  m = ||K||,

K = [ i | 0 <= i <= n : i = 0 or i = n or A[i] < A[i-1]],
```

[4]As in APL, boolean value "true" is converted to integer value 1 and "false" to 0 when type conversion is needed. Similarly, all non-zero integer values are converted to "true" and zero is converted to "false" when necessary.

The parallel interpretation of the improved solution (input tuple A is not repeated here) is as follows:

1. The host processor allocates $(n+1)$ processors R_i according to the domain specification `0 <= i <= n`.
2. Predicate `i=0 or i=n or A[i-1] > A[i]` is evaluated in all processors R_i, and stored as a boolean value **dip?**. All processors R_i, except R_0 and R_n, must evaluate `A[i-1] > A[i]` because the two preceding predicates are false. Its evaluation takes a communication and a computation step similar to the ones described above in Item 3b.
3. Next, evaluate the expression `i` and store the result in *dip_pos* for those processors where *dip?* is true.
4. Now the tuple K is to be constructed by a parallel prefix operation, an allocation step, and a communication step:
   (a) A parallel prefix operation `\\+` [5] is applied to the tuple of values *dip?* in processors R_i for all $0 \leq i \leq n$. Let the components of the resulting tuple of the parallel-prefix operation be stored one by one, from left to right, in variable *k_index* of each R_i for $0 \leq i \leq n$.
   (b) The host allocates m processors S_j, where m is the value of *k_index* of processor R_n plus 1.
   (c) Every R_i with *dip?* true generates a remote reference (S_{k_index}, dip_pos). Note that this remote reference is of the "sender" form, where the value of *dip_pos* instead of the accessing function **dip_pos** is sent. This value is now stored in the variable *position* of processor S_j. This completes the construction of tuple K, which is implemented by processors S_j, $0 \leq j < m$.
   Note that an optimization of the the tuple construction would avoid the allocation of new processors S_j but simply perform step 4c on processors R_i, where each R_i allocates one more variable *position*.
5. The host computes the predicate `m=2`. If it is true, the program terminates. Otherwise:
6. Processors S_j, for $1 < j < m$, generate a remote reference $(S_{j-2}, \textbf{position})$, and assign it to *pre*.
7. Processors S_j, for $1 < j < m$, peform *position* minus *pre*, and assign it to *len*.
8. The host generates remote references $(S_j, \textbf{len})$ with merge operator **max**. The remote argument is assigned to L which holds the result.

The improved version uses only a linear number of processors with respect to the length of the input sequence in a constant number of steps, although the parallel prefix operation may be expensive on some architectures.

5.2 Example: Bitonic Sort

The second example we choose is a sorting network made up of bitonic sorters. It illustrates how recursive calls to macro-parallel modules are made by using the hyper-stack mechanism. The following is a version of the bitonic sort algorithm (see for example, [15]), as shown in Figure 2, written in Crystal. The idea behind the program is the following: suppose we know how to sort a bitonic sequence; then we can sort an arbitrary sequence by building progressively larger bitonic sequences of length n from two sorted sequences of length $n/2$. The program sorts the input sequence recursively, concatenating two sorted sequences (one in increasing order and one in decreasing order) into one bitonic sequence at each step. The function **b_sort** takes a bitonic sequence and produces a sorted sequence in increasing order if a reverse bit **rev?** is false, and in decreasing order if the bit is true. Each data item in the bitonic sorting network is defined by a net, called **b_net**, defined over the indices `i` for the elements of the input sequence, and `d` for the stages of the network, with `d=log(n)` being the initial stage and `d=0` being the final stage. Each stage of the net serves

[5] This will be a plus-scan in PARIS on the Connection Machine.

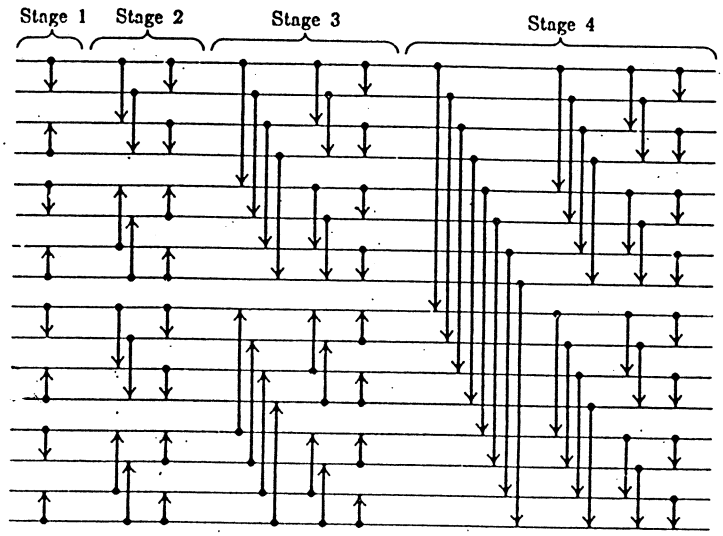

Figure 2: A sorting network consisting of bitonic sorters.

the purpose of breaking up a bitonic sequence into two halves, each a bitonic sequence. In the non-reversal mode, the first bitonic sequence consists of the lower half of the input sequence, and the second one consists of the upper half[6].

```
! Program Sort
! The input sequence must have a length which is a power of two.
sort(r, rev?) =
    << n <= 1 -> r,
       else -> ! sort the concatenation of two sorted sequences
       b_sort(sort(a, rev?) :: sort(b, not rev?), rev?) >>
    where (
      n = ||r||,
      a = r[ 0   <= i < n/2 ], ! lower half of the input sequence
      b = r[ n/2 <= i < n ]),  ! upper half of the input sequence

b_sort(r, rev?) =
    [b_net(i, 0) | 0 <= i < n]
    where (
      B = 0 <= i < n, 0 <= d <= log(n),
      b_net(i, d) over B =
        << d = log(n) -> r[i], ! initial stage
           else ->
             << xor(p?, rev?) -> larger(a, b), ! a pair-wise comparison
                else -> smaller(a, b) >>
        >>
        where (
          p? = i mod 2^(d+1) < 2^d,
          a = b_net(i, d+1), ! i'th element from previous stage
          b = b_net((i - 2^d) mod 2^(d+1), d+1)
              ! element to be compared with the i'th element
              ),
      n = ||r||),
```

[6]We can write a even shorter Crystal program by noticing that a **b_net** itself has a recursive structure, i.e, it consists of a bitonic separator that separates a bitonic sequence into two bitonic sequences, each of which in turn can be sorted by a "half-size" **b_net**.

```
[smaller, larger](x, y) =
    << x < y -> [x, y],
       else -> [y, x] >>,
```

The interpretation of Program Sort can be described in two parts: first, the interpretation of the system of two equations that defines **b_net** and **n** within the scope of **b_sort** by the macro-parallel/substitution model, and next, function **b_sort** and function **sort** by the substitution model.

The keyword **over** declares B as the domain of function **b_net**. An optimization on the allocation of processors is performed: indices of the recursion equation that defines function **b_net** are examined to detect any possible existence of a *loop index*. Roughly speaking, we want to detect if there is any index turns out to be a built-in counter for the iterative steps towards reaching the least fixed-point of the system. In this case, index **d** of the **b_net** is a loop index because we can define a new index d_1 such that it relates to index **d** by $d_1 = \log(n) - \mathbf{d}$. Now substitute **d** by d_1 throughout the equation, and the result is $d_1 - 1$ appearing on the right-hand side while d_1 appears on the left-hand side. Thus d_1 actually counts up the inductive steps. Thus **d** can be interpreted as a loop index for the macro-parallel program; moreover, the loop is bounded and it begins at **d=log(n)** and terminates at **d=0**.

The macro-parallel instructions for the system **b_net** and **n** consists of a loop indexed by **d** where each iteration consists of the following steps:

1. *Allocating processors.* Indices other than the loop index are for processors. The caller of **b_net** allocates **n** processors N_i according to the domain specification. The local variable *bnet* that corresponds to the function **b_net** is allocated. We also assume that the argument **r** resides in some ensemble of processors R_i.

2. *Communications and computations.* The interesting part of interpreting a system of recursion equations lies in the part of recursive calls. Since **b_net** is called in definition **a** and **b**, this means that two remote arguments are needed. A remote reference $(N_{(i-2^d) \bmod 2^{d+1}}, \mathbf{bnet})$ will be generated and the value *bnet* of processor $N_{(i-2^d) \bmod 2^{d+1}}$ will be retrieved. A remote reference $(N_i, \mathbf{bnet})$ is also generated, but it is simply an access to processor N_i's own local variable *bnet*.

   We will not describe step by step the instructions, since simple flow analysis would tell the order in which various communication and computation steps should occur.

Each system of recursion equations is encapsulated and abstracted as a function, in this case, function **b_sort**. Now we turn to the discussion of how substitution of macro-parallel modules work.

Figure 3 illustrate the hierarchy of frames for function **sort** when it is applied to an input sequence **r** of length 8. The recursion in function **sort** grows as a tree. A tree of processors are allocated in the downward phase to keep track of the distributed procedure calling sequence. Physically, only as many processors as leaf nodes are needed because each processor can use its own stack for storing the history of the calls and be re-used at every level of the tree.

In the upward phase, processors are actually allocated for the function **sort**, and the number of such processors is proportional to the amount of data to be processed. The upward phase starts at the leaf nodes of the tree built in the downward phase where each leaf node is responsible for a **sort** function that sorts a single element. The total number of the leaf nodes is the length of the input sequence **||r||**. Since **sort** does not call any other function defined by a system of recursion equations in the case of single element input, only one processor is allocated for each leaf node. At this point, as many as **||r||** hyperstack frames are set in concrete in the sense that macro-parallel processors are allocated. After this step of the computation is completed by processors for each pair of leaf nodes (there is a hidden communication for synchronization here), the hyperstack frame is poped and control is given to the tree nodes at the next higher level. The result of each processor is to be sent by a communication to the processors allocated for **sort** at the next higher level. Since each function **sort** at this level calls **b_sort**, which in trun calls **b_net**, two processors are allocated for each call to funtion **sort**. At this point, as many as **||r||/2** hyperstack frames each containing 2 processors are set in concrete. The process continues going up the hierarchy of frames as shown in the

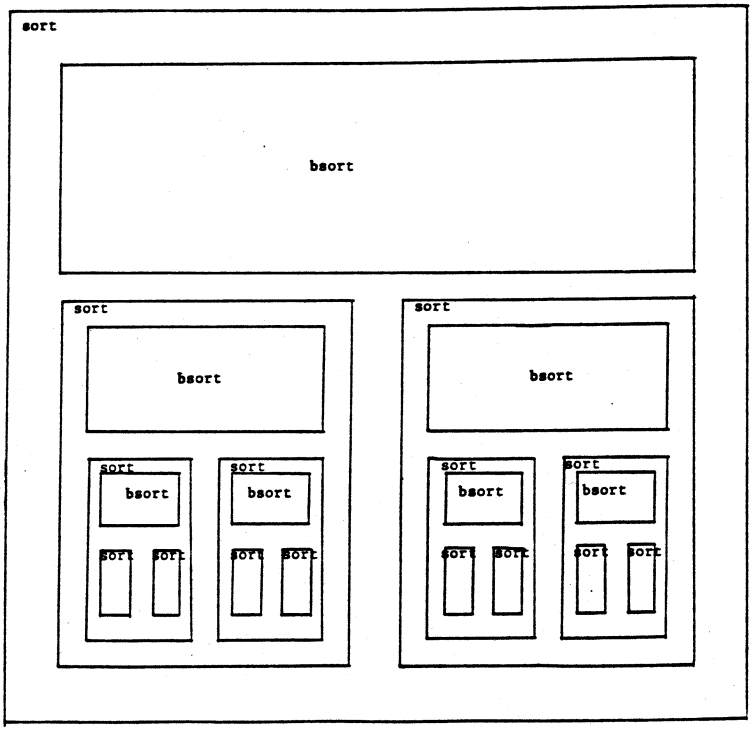

Figure 3: The hierarchy of stack frames at the end of the downward phase.

figure. Since processors can be re-used at every level of the hierarchy both for storing the input arguments from its immediate lower level and for sorting at that level, only a total of n processors are needed.

The essential difference of the macro-parallel/substitution approach from the graph reduction approach can be seen here: besides allocating a logical process (the tree node) for each function call as in the graph reduction model, a group of processors are allocated for each tree node, and its size depends on the size of the input argument to the call. Hence parallelism is used at a much greater scale in our approach.

6 Concluding Remarks

We have chosen a very high-level, mathematical-notation-like language for programming parallel machines, not just because it is easy to use and to understand but because it contains *more information* for parallelization. A specification written in a conventional programming language may have lost certain information due to the inherent extraneous dependencies in the data structures or language constructs, such as lists, stacks, sequential loops, etc. We also believe that adopting existing programming language constructs and conventional mathematical notations and enlivening them with a new parallel model has a distinct advantage over creating new parallel language constructs: once the model is understood by a programmer, the programming itself follows a familiar route.

One pronounced problem in programming general purpose multiprocessors which have fixed interconnections and sizes is that unless each program is written for a given problem of a fixed size range, a programmer must take into account how a large problem with unknown size can be fitted onto a machine with a smaller number of processors. In general, not only problem size causes the necessity of re-distributing work among many processors; also, the machine characteristics and the course of computation could dominate how a task should be assigned to processors. Programming at the microscopic level is not only error-prone and time-consuming but subject to the necessity of customizing each problem for various input size ranges, input data sets, or dynamic behavior of a computation, which often is not known until run time. Hence hand-written, micro-parallel code for multiprocessors can get very bad performance without explicitly programming for load balancing, and this increases the difficulty further. Our approach of macro-parallel programming not only makes programming easier, but allows the control of the *granularity* of computation by playing with the index set: for example, the original index set is partitioned into subsets so that a physical processor is

assigned with a subset of logical processors, one for each index tuple. Moreover, this granularity control can be done at run-time [23] when more information about the work load is available so that the aggregation of index tuple into subsets reflects this knowledge.

Another related issue in parallel programming is that the design space for parallel programs has been enlarged from the one-dimensional space of sequential programming into multi-dimensional space in which processors can be arranged in a variety of different ways. The number of possible solutions to a given problem can be different not only in the algorithmic sense: also, the same algorithm can have many different realizations in multi-dimensional space and time, each with different control flow, data flow, granularity, and space-time tradeoffs. These different space-time realizations can sometimes be determined systematically by optimization procedures [18] and are often best selected on the basis of the target machine characteristics. Hence a problem specification must be high-level enough so that there is room for such optimizations. The combination of the macro-parallel and functional nature of Crystal helps programmers to cope with these increased degrees of freedom for programming, simply by allowing higher-level specifications.

The potential of a functional language for parallelism becomes greater when higher-order functions are used in combination with parallel programming methodologies that improves on, say, more straightforward source programs. A program optimization rule is specified in Crystal as a high order function that takes a source program as its argument. A programmer may apply an optimization rule according to some programming methodology (e.g. that in [3]), or the optimization can be automated and be made transparent to the programmer. In the Crystal programming system, program optimizations are performed as much as possible at the source level where symbolic substitution of program definitions are made. This approach has the advantage that users with more knowledge about the problem can always use that knowledge, still at the source level, to obtain a better program.

The technology of language processing described above is currently under development. Our current goal is that the system will be able to perform language processing for a variety of machines at a reasonable cost, and that the target codes will have reasonably good performance. A higher goal than just making programming parallel machines easier is to illustrate that very high-level programming languages can indeed be made practical by parallel hardware.

7 Acknowledgement

I would like to thank Neil Immerman for many helpful discussions.

References

[1] John Backus. Can programming be liberated from the von neumann style? a functional style and its algebra of programs. *Communications of the ACM*, 21(8):613–641, August 1978.

[2] Guy Blelloch. *Parallel Prefix vs. Concurrent Memory Access.* Technical Report , Thinking Machines, Inc., October 1986.

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

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

[5] M. C. Chen. Very-high-level parallel programming in Crystal. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, September 1986.

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

[7] R.B.K. Dewar et al. Programming by refinement, as exemplified by the setl representation sublanguage. *ACM TOPLAS*, :27–49, January 1979.

[8] W. Daniel Hillis and Guy L. Steele Jr. Data parallel algorithms. *IEEE Computers*, 29(12):1170–1183, 1986.

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

[10] P. Hudak and B. Goldberg. Distributed execution of functional programs using serial combinators. *IEEE Transaction on Computer*, October 1985.

[11] Paul Hudak. Para-functional programming: a paradigm for programming multiprocessor systems. In *The Proceedings of the 13th Annual Symposium on POPL*, January 1986.

[12] Neil Immerman. *Expressibility and Parallel Complexity.* Technical Report 538, Yale University,, April 1987.

[13] Kenneth E. Iverson. *A Programming Language.* John Wiley and Sons, New York, 1962.

[14] S. Lennart Johnsson and Ching-tien Ho. *Spanning Graphs for Optimum Broadcasting and Personalized Communication in Hypercubes.* Technical Report 500, Yale University,, November 1986.

[15] Donald Knuth. *Sorting and Searching. The Art of Computer Programming*, Addison-Wesley, 73.

[16] P. M. Kogge and H. S. Stone. A paralle algorithm for the efficient solution of a general class of recurrence equations. *IEEE Transaction on Computers*, (8), August 1973.

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

[18] J. Li, M.C. Chen, and M.F. Young. *Design of Systolic Algorithms for Large Scale Multiprocessors.* Technical Report 513, Yale University, 1986.

[19] Abhiram G. Ranade. *Constrained Randomization for Parallel Communications.* Technical Report 511, Yale University,, January 1987.

[20] Abhiram G. Ranade. *Equivalence of Message Scheduling Algorithms for Parallel Communications.* Technical Report 512, Yale University,, January 1987.

[21] M. Rem. Small programming exercises. *Science of Computer Programming*, (3), 1983.

[22] John R. Rose and Guy L. Steele Jr. *C*: An Extended C Language for Data Parallel Programming.* Technical Report PL87-5, Thinking Machines, Inc., April 1987.

[23] Joel Saltz and Marina Chen. Automated problem mapping: the crystal run-time system. In *The Proceedings of the Hypercube Microprocessors Conf., Knoxville, TN*, September 1986.

[24] 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.

[25] 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.

[26] P.C. Treleaven, D.R. Brownbridge, and R.P. Hopkins. Data-driven and demand-driven computer architectures. *Computer Survey*, 14(1):93–143, March 1982.

[27] D. A. Turner. Recursion equations as a programming language. In J. Darlington, editor, *Functional Programming and Its Applications*, pages 1–28, Cambridge University Press, 1982.

[28] Hillis W.D. *The Connection Machine.* MIT Press, 1985.