A Declarative Approach to Connection Machine 2 Programming

Giuseppe A. Marino¹, Giancarlo Colla¹

Giancarlo Succi²

¹DIST-Università di Genova
Via Opera Pia 11a, I-16145 Genova, Italia
tel: +39-10-3532997
e-mail: {giancarlo.colla,joy}@dist.unige.it

²Laboratorio Ingegneria Informatica
Studi Aziendali Università di Trento
Via Zeni 8 I-38068 Rovereto
e-mail: giancarlo.succi@ii.unitn.it

Abstract

Writing programs that really exploit the advantages of parallel computers is not easy. The programmer must know their architecture details and choose an algorithm that is suited to the kind of parallelism implemented by the machine. Imperative languages greatly feel the effect of the above mentioned problems. They lay completely upon the programmer the responsibility of choosing the proper algorithm and of implementing it. Declarative languages does not define the operation sequence needed to solve a problem, but they define the problem itself. It is up to compiler and executor to divide job into different processes and to distribute data on machine processors. Therefore they seem to be well suited to be implemented on parallel architectures. We have chosen to implement a subset equational language (SEL) on the Connection Machine 2. The possibility of using the set data structure is fundamental to exploit the data parallelism. This paper describes the SEL abstract machine. This hides to the user the architecture of the CM2 using the Connection Machine as an Active Memory where data belonging to sets are stored and processed. A particularly structure, called CM Data Array, is used to address CM2 processors.

1 Introduction

Nowadays we witness the development of data parallel and process parallel architectures. They have a great computational power, but it is difficult writing programs that really exploit it. The reasons that can explain this situation are the following. First of all, only programmers that know architecture details of parallel computers are able to achieve the best performance from them. Second, the choice of adequate parallel algorithms is also very important to exploit the parallelism of the machine. For example a keypoint fundamental to develop a real good and optimized parallel software is the assignment of data and processes to the machine processors. A good programmer is able to minimize the execution overhead that inter-process communication and shared variables access may cause. At the opposite side, a bad programmer can not only develop an inefficient code, but also to slow down the performance of all the programs running on that host. A good solution is to identify a set of programming languages that both help programmers to develop parallel software, and minimize damages that unoptimized software solutions can cause. Unfortunately the programming languages that are widely used are the imperative ones, they greatly feel the effect of the above-mentioned problems. In fact these languages declare explicitly the operations that the processors have to do to reach the solution (i.e. it is defined the control of the program). This approach lays completely upon the programmer the responsibility of choosing the proper algorithm and of implementing it. We think that declarative languages are most suited to be used on parallel architectures because they define explicitly the logic of the program, leaving the control implicit. The "only" worry of the programmer is to describe what the problem really is and not the
operation sequence needed to solve it. It is up to compiler and executor to divide the job into different processes and to distribute both them and data on machine processors. It is also for this reason that the number of logic and functional languages implemented on parallel architecture is growing. However most of them only exploit either data-parallelism or process-parallelism. We have chosen to implement a logical language based on the set data structure, the Subset Equational Language (called SEL) because it is also suited to be implemented both on a data parallel machine and on a process-parallel one. In this paper we describe the data-parallel implementation of SEL on a Connection Machine 2.

2 From Subset Assertions to Data-Parallel Execution

The Subset Equational Language was developed by Jayaraman et al. [JN88] at UNC/Chapel Hill at SUNY/Buffalo. The exhaustive description of SEL can be found in [Suc91].

A SEL program is a collection of equational and subset assertions:

\[ f(\text{terms}) = \text{expression.} \]
\[ f(\text{terms}) \supset \text{expression.} \]

Their execution may be divided into two parts.

1. the matching operation: the variables that are located in the head are replaced with the ground terms used by the calling assertion,

2. the expression resolution: the right side of the assertion is translated into ground terms (constants, lists, functors or sets).

When the resolution operation is completed, the assertion \( f \) is equal or contains (if subset assertion) the ground instances of expression. In the case of subset assertions, they can be declared by multiple definitions, each one having the same identifier and producing a single subset. The final result is the union of each set. In this paper the discussion is concentrated on this kind of assertions, because they are used to exploit the data parallelism intrinsic in programming languages based on sets.

An example of SEL code using subset assertions is shown by the following declarations, where it is defined the permutation of the elements belonging to a set.

\[
\begin{align*}
\text{perms(\{}\} &= \{\emptyset\}. \\
\text{perms(x|t}) &\text{ contains distr(x,perms(t)).} \\
\text{distr(x,\{t|\}) contains \{x|t\}.}
\end{align*}
\]

The query \(?-\text{perms(A,B,C)}\) starts the SEL execution that produces all the existing permutations of the elements \((A,B,C)\), i.e. \([A,B,C],[A,C,B],\ldots,[C,B,A]\). The first line declares that the permutation of an empty set is an empty list (this is trivial). The second line is more complex. The left side of the assertion defines the matching operations that must be executed and the right side declares that the permutations are obtained applying the distr assertion to the object \(x\) (extracted from the set by the matching operation) and to all the permutations of the subset \(t\) (i.e., starting Set except for \(x\)). The third line defines the distr assertion.

2.1 The Multiple Matching of Subset Assertions

A set can be declared using three different definitions:

\(\{A,B,C\} \quad \{x|\_\} \quad \{x|t\}\)

The first item defines the set (ground term) that contains the elements \((A,B,C)\). The second appears only at the left side of an assertion and it is used to perform the set matching operation. Its behaviour is what we call implicit iteration, viz., selecting the elements of a set one at a time. The assertion \(\text{distr}(x,\{t|\})\) contains \(\{x|t\}\) has as second argument this kind of set definition. When the \(\text{perms}\) applies the query \(?-\text{distr}(A,\{A,B\},B,A))\) to distr, the variable \(t\) is in turn matched with every element belonging to the \(([A,B],[B,A])\) set (multiple matching).

- \(x \leftarrow [B,C]\)
- \(x \leftarrow [C,B]\)
The third item is used also by set matching operation in the same way of \( \{x|\_\}\). However, in this case, the matching between \( \{x|t\} \) produces not only a multiple matching for the variable \( x \), but also for \( t \) at which it is bound the set containing all the elements of the starting set except the one that is bound to \( x \).

- \( x \leftarrow A, t \leftarrow \{B,C\} \)
- \( x \leftarrow B, t \leftarrow \{A,C\} \)
- \( x \leftarrow C, t \leftarrow \{B,A\} \)

The \texttt{perms} assertion is executed for all possible matching and calls \texttt{distr} passing everytime different arguments. In the same way the set definition present in the \texttt{distr} assertion causes multiple execution of the distribution operation.

The implicit iteration on the elements belonging to a set is very useful to implement SEL on a data parallel architecture. In fact we can notice that the same operations are applied on different data. This implicit data parallelism may be exploited by distributing elements belonging to sets on the processors of a SIMD machine and performing parallel execution.

The principle that we use to exploit parallelism is simple, however we had to face some more difficult problems. When an assertion must be applied to a couple of sets, the parallel execution can be performed only on one set, while the execution on the other set must be serialized. The same situation happens when a subset assertion uses another subset assertion. It is again important to have a mechanism that serializes computation on the elements of the \textit{Active Memory}; this mechanism uses the CM Data Array and the parallel environments (these will be explained later).

3 An Overview of the Subset Abstract Machine

The SAM belongs to the WAM [AK90] family, since its general structure resembles that of the WAM. However there is an important difference: while the core of the WAM is the handling of unification capabilities, the SAM core is the handling of sets. For this reason some WAM structures are useless and new memory areas are needed.

The Subset Abstract Machine is a software layer that models the machine architecture so it hides the implementation details of the parallel computer. The programmer writes a source code that is a collection of equational and subset assertions, besides he defines the elements (sets) on which the same operations are applied. The abstract machine distributes data belonging to sets into the CM2 processors, synchronizes parallel executions and handles the data communications through inter-processor links.

The SAM is made by different modules, each one with a particularly task. Figure 1 outlines the main elements making up the virtual machine.

3.1 SAM Registers

Registers are used to control the execution of SEL programs. The PC (program counter) addresses the next SEL assembly instruction that has to be executed. The CP register is used to save the value

![Figure 1: General Structure of the SAM](image-url)
of the PC when a new assertion is called. CE (continuation environment), LCP (last choice point) and CPE (continuation parallel environment) locate the part of memory that contains the informations indispensable to restart stopped executions. The H register (heap pointer) identifies the section of heap that is yet used. A1,A2,...,An are the argument registers: they contain the arguments that are passed between the calling and the called assertion. X1,X2,...,Xn are temporary registers: they are used to store the content of variables that are local to an assertion. As in the WAM, the A and X registers are identical, but they are used for different purposes and, for this reason, they are called in different ways according to their use.

3.2 The SAM Emulator and the Code Area

To execute a SEL program, it is necessary to compile it into a sequence of assembly instructions. For example \texttt{father(John) = Andreas} corresponds to the following SEL assembly code (SAM code).

\begin{verbatim}
get_constant A1 John   \% father(John) 
store_ind_const A2 Andreas \% = Andreas
\end{verbatim}

These assembly instructions are saved into the code area and are addressed by the \texttt{program counter}. The emulator fetches the instruction that is pointed by the PC, checks the instruction identifier (GET_CONSTANT, GET_VALUE, ...) and uses it to execute a particular sequence of operations.

3.3 The Stack: Environments and Choice Points

The stack is fundamental to manage computation, because environments and choice points are stored onto it.

Environments are necessary to save the machine state when nested assertions must be executed to obtain the solution. The next example can underline the importance of the environments.

\begin{enumerate}
\item \[f(X,Y) = g(h(X),l(X,Y)).\]
\item \[f(X,Y) = v1 :- h(X)=v2, l(X,Y)=v3, g(v2,v3)=v1.\]
\end{enumerate}

Line [1] contains the definition of the assertion \(f\): the matching operation binds two ground terms to the variables \(X\) and \(Y\), then the solution is obtained applying the assertion \(g\) to \(h(X)\) and \(l(X,Y)\). Line [2] contains the \texttt{flatten} form that specifies the operations sequence which the emulator must follow to compute \(f(X,Y)\): the assertion \(h(X)\) and \(l(X,Y)\) must be executed before the \(g(v2,v3)\). When the instruction call \(g\) has to be executed, the values of \(X\) registers must be saved into the stack, because the execution of the assertion \(g\) can change their values. When the execution of \(g\) is finished, the old values of registers can be restored.

The Choice Points are similar to the environments, but they are used to execute subset assertions that are defined by difference rules. For example the \texttt{union} of two sets can be defined as:

\begin{enumerate}
\item \texttt{union(S1,..) contains S1.}
\item \texttt{union(.,S2) contains S2.}
\end{enumerate}

The solution of the \texttt{union} assertion is built by the union of the two different solutions (line [1] and line [2]). When the first line has to executed, the Subset Abstract Machine saves only the registers that are necessary to compute the next subset assertion with the same name (line [2]).

3.4 Sequential Data Structures: Lists and Functors

Different data structures are used in the SEL language; they are functors, lists and sets. While the elements belonging to sets are stored into the Connection Machine, lists and functors are built into the memory of the CM2 front end (a normal sequential processor). In this section we give an overview of the modules that are used to store and process lists and functors.

The heap\footnote{The stack and the heap are placed into the same memory area; the stack grows from low to high addresses and the heap from high to low ones. This management optimizes his memory usage because a stack overflow or an heap overflow occurs only when the total memory is assigned to data.} is an array of cells where complex data structures are stored. It is constituted by cells, each one contains an elementary term ground (i.e. a number, a literal or a string). If the functor \(f(a,g(b,c))\) and the list \([a,b]\) have to be built onto the heap, the SAM writes the heap cells so they appear as in figure 2.
When lists and functors are elements belonging to a set, they are stored into the memory of the CM2 processors using a structure (called local heap) that is similar to the above-mentioned model. However, in this case, a new module is necessary to perform the parallel pattern matching: the Push Down List. The SEL assembly instructions build onto the PDL a skeleton of the list or functor to be matched and, then, broadcast its contents to all the CM2 processors that are involved in the execution.

4 Modelling the CM2 as an Active memory

The active memory is a collection of memory cells that have a computing power, i.e. they not only store data, but also perform a data parallel execution. Therefore, as be seen in figure 1, the active memory is implemented on the CM2 where a physical parallel execution is possible (while other machine modules, e.g. the stack, the heap and so on, resides on the front-end because their executing model is intrinsically sequential). Every single set data is bound to a single active memory cell that has a computing capability.

The version of the CM2 that we use here has 16K processors and the set dimension of some source code may exceed this number. Therefore, it is necessary binding more cells to a single physical processors. This operation may be performed following three different approaches:

1. to leave everything up to the machine,
2. to rebuild entirely the mechanism of virtualization, tailoring it to this problem
3. to use a mixture of the first two.

The first approach is classical in sequential implementations, however here the issue is more critical since the communication overhead is not under control and could lead to inefficiencies. The second is really hard to handle, since everything has to be taken care of, and there is the risk (somehow paradoxical) that the difficulty of this task lead to a slow and buggy implementation. The best solution appears to be the third one which is a kind of compromise between (1) and (2) and it has been taken here. We have taken note that the operations on a set produce some intermediate data (on which a data parallel execution is possible). However migrating these new elements toward other cells is a useless operation because they are only transitory data. Therefore they are stored into the cells from which they have origin. This handling minimizes the inter-processors communication into the CM2. Only the final set elements are copied into new cells leaving the routing mechanism up to the machine. As a matter of fact the elements belonging to sets may be separated in two classes:

(a) those generated as intermediate results of a computation,
(b) those produced as a final result of an assertion.

To have a better understanding of this problem, consider the following assertions, where doubleIncrSet increments the elements of the argument set by 2.

\[ \text{doubleIncrSet}(\{x\}) \text{ contains } \{\text{incr}(\text{incr}(x))\} \]

\[ \text{incr}(x) = x + 1. \]

The next SEL assembly code illustrates the lines of the SAM code generated for this assertion that executes the increment of \( x \) in parallel on all the processors on which the set is stored.\(^2\)

\(^2\)This is just a scheme of the real code: not all the SAM instructions needed are placed and the arguments are left over in order to keep the presentation simpler.
Here both [8] and [11] act on a whole set: they increment the value of each element stored inside the set. However there is a difference between the two situations, viz. while the result of [8] is needed only locally, the result of [11] is a global result which must be kept for further computation. In other words, the result of [8] pertains to class (a) while the result of [11] to class (b). The choice of this design is to store a set of class (a) in the same cells as the original set and to store a set of class (b) in a new cluster of cells. Since the abstract design of the SAM associates to each set element its own cell, this kind of implementation amounts to handling explicitly the virtualization for sets of class (a) and leaving this task to the underlying system for sets of class (b).

Such a choice yields an efficient and fast way of generating intermediate sets, which are needed only locally and not globally, therefore they do not need to be burdened by communications. Furthermore this choice keeps a simple structure for global sets: their elements maintain the one to one correspondence with an AM cell, possibly a virtual one, making easy the task of bookkeeping the system, identifying global sets, dereferencing their elements and so on, as it is explained in the next section.

The implementation of the AM on the Connection Machine (CM-AM) is built upon two major entities:

1. the Active Memory itself, which contains the sets elements in its cells,
2. the CM-Data-Array, which is used to identify the various sets in the Active Memory.

Figure 3 illustrates the structure of the AM: it is the usual array of cells; in this implementation a register of each cell, the tag register, is used to identify a permanent set. Other sets’ elements can be stored in this cell, but they belong to temporary sets. The element of the permanent set is stored in the first register of the cell, therefore a permanent set in the AM is represented as a collection containing what is in the first registers of the cells that have the same value in the tag register. Therefore a permanent set is identified just by a tag.

The elements of temporary sets are placed in the other registers of the cells, in the second, in the third, and so on. Therefore a temporary set is identified by two numbers: a tag plus a number, specifying in which registers of the cells pertaining to that temporary set its elements are stored.

While set does not contain duplicated elements, temporary set may contain them, because the duplicate check is performed only when the final set is created. Such design is due to the fact that the extra computation performed here are in data, a complete description of the problem can be found in [SC92].
5 Addressing Active Cells: the CM Data Array

The CM-Data-Array is a data structure stored in the front-end that is used by the main processor to address the sets stored in the Active Memory. It is a table (figure 4) and (in the present version of the CM-SAM) each entry of it identifies a set. Therefore any reference to sets in the front end are references to entries in this table which provides all the information that are needed to completely identify a set in the Active Memory. Each entry of the CM-Data-Array contains the following fields:

- **type**, which specifies whether a set is a permanent set, PER, or is an intermediate result, DDT (The reason for this name is explained in the section about mapping);

- **tag**, which is the tag used in the tag register of the cells of the set in the Active Memory to identifies which cells belong to a set;

- **register number**, which determines in which register of the cells the element of an intermediate set are stored, it is 0 for permanent sets.

The use of the CM-Data-Array has many advantages with respect to a straight usage of the tag number (and of the register number for temporary sets). First of all the design is much cleaner and it is possible to have a higher degree of modularization. Then there is a higher degree of flexibility of the whole structure since the code never addresses directly objects in the hypercube. Furthermore, there may be different entries in the table pointing to the same set in the AM; this can be achieved when ad hoc abstract analyzers -which are under development now [SM91a]- determine that two sets are identical: then it is possible to allocate the set just once in the Active Memory and to refer to it by means of two different entries of the table. There is one more reason for the usage of the CM-Data-Array which includes all the previous ones: it is the way in which the mapping operations are performed. The next section is about this problem.

6 The Parallel Environment

The Connection Machine is a SIMD machine, therefore only one instruction at a time can be executed on all the processors. This peculiarity needs some special care, since there are data parallel flows requiring MIMD structures. If a single operation must be performed on two sets whose elements are allocated in different active cells, then the execution may be parallel only on one set while a sequential execution must be performed on the other. A mechanism, called parallel environment, has been devised in order to take care of it. The next assertion presents a situation where such parallel environment is used.

\[
\text{oneProducts}(X, \{Y_i\}) \text{ contains } \{X \cdot Y\}. \\
\text{allProducts}(\{X\}_1, S) \text{ contains oneProduct}(X, S).
\]

The SAM code for the clauses oneProducts and allProducts is the following.

![Figure 4: Structure of the CM-Data-Array.](image-url)
The allProduct execution cannot be parallel, because a nested parallel assertion exists (the oneProduct assertion). There is therefore the need of sequentializing the data parallelism on the first set and exploiting it only on the second set. The end_map_over of line [12] will restart the data parallelism on \{1,2,3\}. Consequently there is the need of providing a mechanism for stopping a data parallel execution on a set to resume it later. Since this situation can be nested (e.g., if oneProduct called an assertion with data parallel execution on another set) a stack of such information must be saved. The parallel environment is used for this purpose: a frame containing information about the set in use and which elements of it has already been analyzed is stored on the stack each time a mapping occurs inside the execution of another mapping and it is popped at the end of the mapping.

So at line [6] a parallel environment for \{1,2,3\} is placed on the stack, an element of it is selected in order to start the sequenzialization (for instance 1) and it is marked as "already analyzed" in the environment. Any reference to Y1 is now a reference to 1. Therefore line [7] put a 1 in register A1. Then the standard flow is followed in lines from [8] thru [11]. In line [12] there is the end of the mapping. At this point the parallel environment is popped from the stack and it is analyzed to see if the processing has been completed. This is not the case here, since elements 2 and 3 of \{1,2,3\} has not yet been processed. A new element is then picked out from the parallel environment, for instance 3, and the environment is saved back on the stack. Any reference to Y1 is now reference to the number 3. Again lines [8] thru [11] are executed and line [12] is reached. The parallel environment is again popped from the stack, element 2 is selected and then the environment is placed back on the stack. The execution jumps back to line 7, where 2 is placed in A1. Then lines [8] thru [11] are executed and line [12] is reached. Now all the set \{1,2,3\} has been processed therefore the environment can be popped from the stack, the original data parallelism on \{1,2,3\} can be resumed and lines [13] and [14] are executed. Then the execution goes back to allProduct which is then completed in the normal way.

The need for parallel environment is present not only when there are nested calls to clauses inside mappings but any time there is a nested mapping, therefore also in situation like the one of cartProduct which computed the cartesian product of two sets.

cartProduct(\{X\}_, \{Y\}_) contains \{pair(X,Y)\}.

7 Conclusions

Several are the issues still open in this research. Abstract analyzers and garbage collectors are under study and development. A different way of mapping the Active Memory on the hypercube by means of hash tables is almost completed. Network optimizations and compilation enhancements are also hot points.

Acknowledgments

This work has been partly supported by the italian ministry of university and scientific research (40% foundings). Most of the ideas presented in this paper has been discussed with B. Jayaraman. The authors thank researchers of DI/Parma for their support in the usage of the Connection Machine.

nocitebapopra:92 nocitebapo:92

References


