Moded Flat GHC for Data-Parallel Programming (extended abstract)

Kazunori Ueda
Department of Information and Computer Science
Waseda University
4-1, Okubo 3-chome, Shinjuku-ku, Tokyo 169, Japan
ueda@ueda.info.waseda.ac.jp

November 25, 1994

Abstract. Concurrent logic languages have been used mainly for the (parallel) processing of rather irregular symbolic applications. However, since concurrent logic languages are essentially general-purpose, they should be applicable to problems with regular structures and their data-parallel processing as well. This paper studies the possibility of massively parallel processing in concurrent logic programming, focusing on arrays and its data-parallel processing.

1 Regular Computation in Concurrent Logic Programming

> *"We hope the simplicity of GHC will make it suitable for a parallel computation model as well as a programming language. The flexibility of GHC makes its efficient implementation difficult compared with CSP-like languages. However, a flexible language could be appropriately restricted in order to make simple programs run efficiently. On the other hand, it would be very difficult to extend a fast but inflexible language naturally."* — [5] (1985)

Concurrent logic languages have focused mainly on the parallel processing of symbolic applications with rather irregular structures [8, 4]. However, real-life parallel symbolic applications (such as machine learning) may involve a lot of numerical computation as well. We anticipate that future symbolic languages should provide certain support of high-performance computing. Whether concurrent logic languages can evolve in this direction deserves studying in depth.

Previous approaches to irregular parallel symbolic processing involving numerical computation were mostly multi-lingual. The whole computation was coordinated by symbolic languages, while numerical computation was programmed in conventional languages and called via foreign-language interface. This may be a promising approach in the short run, with the obvious advantage of the reuse of existing numerical software, but it is partly grounded on an assumption that nonprocedural symbolic languages are unsuitable for efficient regular computation. Procedural languages gain performance in regular computation by destructive assignments and random access to array elements, but it is yet to see whether nonprocedural languages can handle arrays with competitive performance and how they can exploit data parallelism. Some array processing applications can be rewritten naturally using list processing, but others do require efficient support of arrays.

Since irregular programming encompasses regular programming, flexible languages could be made more suitable for regular programming by restricting its descriptive power somewhat and doing aggressive optimization based on static program analysis. The crucial point here is *not* to do so in an ad hoc manner.

We have thus far developed a constraint-based mode system for Flat GHC that enables static dataflow analysis [6, 10]. The resulting language is called Moded Flat GHC. This approach is similar in essence to static type systems found in many languages, and is considered one of the most systematic approaches to syntactic restriction and static analysis. The following observations have been obtained from several years of study:

1. Slight assumptions on the programming style—e.g., instantiation of variables is cooperative rather than competitive—make it possible to analyze dataflow (modes) statically and efficiently.

2. Well-modedness guarantees that all unification body goals are assignments to variables and therefore do not cause failure.[1]

3. As a byproduct of dataflow analysis, the information on the number of access paths to data (i.e., whether each datum has exactly one reader or possibly many) is obtained. (The number of writers is guaranteed to be one in well-moded programs.) This information is fundamental for memory management and the syntactic control of aliasing.

4. The mode system can deal with programs using vectors (one-dimensional arrays). In well-moded programs, arrays with a single reference (access path) can be updated in place without any runtime check on the number of references.

5. Data type information (such as the second argument of `p` being a stream of vectors) can also be analyzed efficiently by imposing slight assumptions on our programming style.

These observations suggest that Moded Flat GHC is a promising platform for the study of efficient array operations in nonprocedural languages. Furthermore, the facts

- that the language does not have the notion of destructive assignments and distinguishes between old and new arrays and

[1]To be exact, the mode system assumes extended occur check [10] that excludes the unification between identical variables, but ordinary programs do not perform such unification.

- that all data created and referenced by concurrent processes should explicitly appear as goal arguments

are considered even beneficial for program analysis and optimization for parallel implementations.

When pursuing the performance of regular computation, it is very important to maintain the simple computational model and the mathematically tractable semantics of nonprocedural languages. Unnatural extensions can lose the *raison d'être* of nonprocedural languages and may well hamper sophisticated optimization. KL1, designed based on Flat GHC, already provided array features based on this principle [7]. At the language level, KL1 provided array operations within the framework of logic programming, while at the implementation level, one-bit reference counting was employed for the efficient update of single-reference arrays [2]. They achieved the same time and space complexities as those of procedural languages. Moded Flat GHC aims to go further and establish language constructs in which we can analyze the number of references statically and efficiently. The purpose is *not* to detect all single-reference cases, which is undecidable, but to find a simple scheme which is safe and covers almost all practical cases.

2 Moded Flat GHC and Array Operations

When a concurrent logic program is executed, the arguments of a goal g are instantiated to constants or data structures as computation proceeds. A mode of Moded Flat GHC is a function that tells whether the value of each particular place in the data structures (including the top level) will be determined by the goal g (output mode) or by some other goal (input mode). Mode analysis means to compute a well-moding of the pair of a program and a goal clause. In addition to the observations in Section 1, the mode system of Moded Flat GHC has the following desirable characteristics:

1. The mode system can deal with (i) complex data structures such as streams of streams and streams of incomplete messages, (ii) difference lists, and (iii) mutual recursion, all with no difficulty.

2. Mode analysis is basically a constraint satisfaction problem formed by the mode constraints syntactically imposed by individual clauses. The constraint satisfaction problems can usually be solved by the unification of feature graphs (feature structures with cycles); generate-and-test search is usually unnecessary.

3. Being constraint-based, the system is inherently amenable to separate analysis of individual program modules. Also, the three aspects of mode analysis—declaration, checking, and inference—can be dealt with within a unified framework.

4. The practical cost of all-at-once mode analysis is just almost proportional to the program size n and the *complexity* of the data structures used (in terms of the size of the grammar to generate them). The *size* of the data structures to be generated

does not matter. The practical cost of separate analysis is worse only by $O(\log n)$ times.

The readers are referred to [10, 9] for mathematical details and basic theorems.

As mentioned before, KL1 supports data structures called vectors (one-dimensional arrays). Vectors can take any terms as elements. Several vector operations are provided, but under the static mode system, the most fundamental operation turns out to be the following:

```
set_vector_element(V, I, E, NewE, NewV).
```

This operation receives an array `V`, the index value `I` and the new element value `NewE`, and returns through `E` the `I`th value of `V` and through `NewV` an array which is identical to `V` except that the `I`th element is replaced by `NewE`.

One may wonder why this operation is more fundamental than just reading or updating an array element. The reason is that it imposes weaker mode constraints on its arguments and is hence more generic. The reason why it imposes weaker constraints is that it keeps unchanged the number of references to the whole array and its elements. For instance, the `I`th element becomes accessible through the variable `E` but inaccessible through the new array `NewV`. All the other elements of `V` are inherited to `NewV` and thus remain accessible.

In contrast, an access operation that simply returns a new reference to the `I`th element of `V`, such as Prolog's `arg`, will impose far stronger mode constraints on `V` and its elements; see [9] for the reasons.

The moral of the above result is that, in the array processing in Moded Flat GHC, array elements should be *removed* once accessed and the resulting blank should be filled with another value. For instance, when performing some operation on an array element and storing the result back to the array, the blank can be filled with a variable which will be instantiated to the result value. An exception to the above principle is that *read-only* arrays, namely arrays whose elements are (or are to be instantiated to) ground terms, can be accessed without removing its elements.

The reason why non-ground elements must be removed is that they have an aspect of *resources* in general, whose access paths should not be copied or removed freely. A typical example of "values as resources" is a bidirectional communication stream. Leaving an accessed element intact in the array does increase the access paths to that element and imposes strong mode constraints to those paths, making the array operation less generic [9].

For arrays with ground elements, accessed values can be left in the array because the mode system guarantees the new reference to the element to be a read-only path that can be safely added. Numerical applications will frequently use arrays in this manner.[2] However, the applications of data-parallel array processing are not limited to numerical computation, and so the observations obtained from generic arrays give us useful guidelines for designing array operations.

As we mentioned in Section 1, mode analysis provides us with the information on the number of access paths. If it can be guaranteed statically that an array `V` has only one

[2]However, we anticipate they also frequently use arrays whose values are accessed only once.

reader, the above `set_vector_element` can destructively update `V` to create `NewV`. This is the most significant application of mode analysis from a practical point of view. `set_vector_element` on an array with (possibly) two or more readers involves the copying of the array, but we expect that most of the arrays are either

- referenced by a single process and updated by `set_vector_element` as computation proceeds, or
- composed of ground elements and referenced by multiple processes but not updated after creation.

They can be distinguished by mode analysis and/or declaration and be implemented quite differently. For instance, distributed-memory parallel computers could hold multiple copies of the latter arrays.

3 Parallel Operations on Arrays

We have so far discussed why we are so confident in handling arrays elegantly in non-procedural languages. However, parallel processing imposes an additional requirement that arrays should be accessible in parallel. How can this requirement be supported at the language level and the implementation level?

The basic operations for the parallel processing of arrays are the splitting and the concatenation of arrays. Consider quicksort. Small elements are gathered to the left part of an array and large elements are gathered to the right. Then the array will be split at the threshold value and the two subarrays will be processed recursively in parallel. If the array is on shared memory, the splitting does not involve copying and yet the two subarrays can be accessed without any interference.

If small and large elements are gathered by destructive operations, it implies that the results of recursive quicksort will also be obtained by destructively permuting the original array elements. In that event, the concatenation of the sorted subarrays will not involve copying, either. To generalize, if the two arrays to be concatenated are guaranteed to reside next to each other, they can be concatenated in constant time.

Thus concurrent processes can efficiently and safely update different parts of an array on shared memory in parallel. This observation is important also for distributed memory implementations, because concurrent processes mapped on the same processor share memory.

Splitting and concatenation of arrays are as fundamental as `set_vector_element` from the viewpoint of modes, because they preserve the number of references to the elements and therefore are compatible with the "values as resources" paradigm. In contrast, copying the whole or a part of an array will increase the number of references and impose stronger mode constraints.[3] Extracting a subarray and discarding the rest will similarly impose strong mode constraints. The use of these operations should be limited to arrays with ground elements.

[3]Sending an array to another processor will involve copying but will *not* increase the number of references as long as the reference within the original processor is discarded.

An n-dimensional array could be defined as a one-dimensional array of $(n-1)$-dimensional arrays. However, if both column vectors and row vectors of matrices are frequently accessed, multidimensional arrays should be supported by the language. Element access, splitting, and concatenation of multidimensional arrays are similar to those for one-dimensional arrays. Splitting a multidimensional array may result in two arrays with interleaving memory areas, but this causes no security problem.

4 Data-Parallel Operations on Arrays

The previous section considered the case where O(1) parallel processes operated on arrays of length n in parallel. Data-parallel processing will involve more processes to operate on the array. This section studies through examples what features should be considered in the implementation of data parallelism.

Study of the language constructs for data parallelism is important as well. For instance, $\Theta(n)$ processes could be created more elegantly using iterative constructs than by recursion. Appropriate notations should be provided for referring to array elements and subarrays in those iterative constructs. However, they could be defined by using simple higher-order constructs (i.e., first-class program codes) and partial evaluation, both of which the current mode system can deal with naturally. So the rest of the paper will focus on the implementation level, leaving surface language design to future research.

4.1 Matrix Multiplication

We take matrix multiplication as the first example [3]. Consider the multiplication of an $l \times m$ matrix A and an $m \times n$ matrix B. The most natural data-parallel modelling will be to create $l \times n$ processes each corresponding to an element of the result matrix C. Let P_{ij} be the process for computing C_{ij}. The creation of the P_{ij}'s can be specified easily using recursion.[4] The mapping of the P_{ij}'s to physical processes can be specified by extending the mapping construct of KL1.

The allocation of data can be determined from the mapping of the processes accessing data (cf. High Performance Fortran) as long as the places of data creation and consumption are the same. The places are different in general, however. Furthermore, the consumer of data may migrate before accessing the data. If the place of consumption can be analyzed statically and the consumer process accesses all the data, the most efficient way will be to send data eagerly to their final destination. However, if the place of consumption cannot be analyzed or not all data are accessed, a more sophisticated strategy should be employed.

How should the matrices A and B be distributed over processors? Suppose we have l processors and map P_{ij} ($j = 1, \ldots, n$) to processor i. Then A can be split into l row vectors and distributed to those processors. For B, a straightforward implementation may copy and distribute the whole matrix to all processors, because each P_{ij} will access the jth column vector.

[4]This does not imply that they are created one by one recursively; optimization is a separate issue.

However, the P_{ij}'s can be executed independently and will discard the column vectors on termination. If the column vectors migrate from one processor to another cleverly enough, processor i needs to hold only $\mathrm{O}(1)$ column vectors at any time. How to achieve this bound (semi-)automatically is an interesting topic of future research.

The result C can be built by concatenating one-element subarrays computed by each process. Static analysis of the size of C will be easy even without array declarations and will simplify memory allocation.

4.2 Prime Number Generation

We take one more example from [3], the sieve of Eratosthenes. Process P_k $(k = 0, 1, \ldots)$ holds a bit vector corresponding to the natural numbers in $[nk, n(k+1))$, and strikes the multiples of primes in parallel. Since the bit vectors held by individual processes are independent, no difficulty arises.

Instead of KL1 vectors and `set_vector_element`, which are too general for implementing bit vectors, strings can be used to implement bit vectors. While a vector can store arbitrary terms (including variables) as elements, a string can store only nonnegative integers of limited size. Operations on strings are all considered special cases of vector operations.

4.3 Relaxation Method

Suppose we are to solve two-dimensional Laplace equations using a distributed-memory computer and a relaxation algorithm. Each processor will take charge of a particular small block of a large two-dimensional array, but the neighboring elements of the block are also necessary to compute the next approximation. If the array is simply divided into disjoint blocks and distributed, the processors must communicate with the neighboring processors and ask the values of those elements. To avoid this, each processor should get hold of the neighboring elements of its own block beforehand. These elements are said to form a *guard wrapper* [3].

In the relaxation method, each process will destructively (at the implementation level) update its own block. This may cause problems in shared-memory parallel implementations in which elements in the guard wrappers are shared among processors. In distributed-memory implementations, however, the guard wrappers will be stored in local memory and hence will not be destroyed by the neighboring processors. Processors should obtain new guard wrappers upon entering the next iteration, but this can be programmed easily.

Techniques such as guard wrappers and sentinels are quite useful in parallel array processing as well as in sequential processing; they absorb exception handling on array borders and thus contribute to simplicity and efficiency. One issue is how a parallel implementation can efficiently accommodate guard wrappers or sentinels around *subarrays* passed as arguments.

In multiprocessing within one processor, a process cannot in principle accommodate sentinels by extending a subarray because this may destroy neighboring areas. However, if it is guaranteed that the process does not access those neighboring areas, it can accommodate sentinels by saving the old contents of the neighboring areas. This guarantee can be obtained easily in some cases; an example is the case where each neighboring subarray is referred to exclusively by some *other* process.

4.4 Histogram Generation

The last example is the so-called histogram problem, in which n processes share a histogram with k elements and increment them in parallel. We assume that n is relatively small while k is large. Since the histogram is a shared resource, it is most natural to model it as a server holding an array and to receive requests through a message stream. However, serializing all the accesses from the client processes creates a bottleneck. An alternative would be to let each process hold its own histogram and sum up all the histograms in the final phase. This avoids interference in the counting phase, but at the cost of $\Theta(nk)$ space.

At the implementation level, the point is that it is unnecessary to lock the whole histogram to update some of its elements. We could instead prepare many local locks to exploit parallelism. In concurrent logic programming, this can be achieved by preparing many local servers and their front-end stream mergers. However, if we prepare many servers and let processes asynchronously update histogram elements, it becomes more difficult to detect the termination of the counting phase and read the final histogram. The same difficulty arises in concurrent object-oriented languages; the histogram could be updated easily by preparing an array of k counters, but under the assumption of asynchronous message passing, the termination detection of the counting phase is not so easy. If each counter holds a reference counter as well, its maintenance will take $\Theta(nk)$ time in total.

We anticipate that in any computational models based on asynchronous message passing (including concurrent logic programming and concurrent object-oriented programming), there exist no fully parallel algorithms that simultaneously achieve (i) $\mathrm{O}(1)$ time for the update of each histogram element, (ii) $\mathrm{O}(n+k)$ time for the final processing (termination detection and the reading of the final histogram), and (iii) $\mathrm{O}(n+k)$ space. If this is the case, the histogram should be divided into an appropriate number of blocks for distributed management. The number will depend on the relative importance between the degree of parallelism in the counting phase and the performance of the final processing.

5 Conclusions

We have discussed how arrays can be manipulated in parallel in concurrent logic languages with a static mode system. We argued that Moded Flat GHC is a good platform for describing and analyzing various array operations and for the parallel and massively parallel processing of various kinds of applications, though a lot of future study is necessary particularly on surface language constructs and implementation on shared- and distributed-memory computers.

Concurrent logic languages have focused on rather irregular problems, but they are expected to be useful also for the description and (massively) parallel execution of regular problems because:

- the mode system provides both programmers and implementation with a useful fundamental information by analyzing the handling of "data as resources", and
- the conceptual clarification they provide (e.g., the separation of concurrency and parallelism [7]) should be useful in any future applications.

In particular, the possibility of being able to control aliasing much more nicely than in procedural parallel programming is worth exploring both for better programming styles and for program optimization.

References

[1] Chandy, M. and Taylor, S., *An Introduction to Parallel Programming*. Jones and Bartlett Pub., Inc., Boston, 1992.

[2] Chikayama, T. and Kimura, Y., Multiple Reference Management in Flat GHC. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, Cambridge, MA, 1987, pp. 276–293.

[3] Hatcher, P. J. and Quinn, M. J., *Data-Parallel Programming*. The MIT Press, Cambridge, Mass., 1991.

[4] Taki, K. (ed.), *Parallel Processing with Fifth Generation Computers*. Kyoritsu Shuppan, Tokyo, 1993 (in Japanese).

[5] Ueda, K., Guarded Horn Clauses. ICOT Tech. Report TR-103, ICOT, Tokyo, 1985. Also in *Logic Programming '85*, Wada, E. (ed.), Lecture Notes in Computer Science 221, Springer-Verlag, Berlin Heidelberg, 1986, pp. 168-179.

[6] Ueda, K. and Morita, M., A New Implementation Technique for Flat GHC. In *Proc. Seventh Int. Conf. on Logic Programming*, The MIT Press, Cambridge, Mass, June 1990, pp. 3–17.

[7] Ueda, K. and Chikayama, T., Design of the Kernel Language for the Parallel Inference Machine. *The Computer Journal*, Vol. 33, No. 6 (December 1990), pp. 494–500.

[8] Ueda, K., The Fifth Generation Project: Personal Perspectives, *Commun. ACM*, Vol. 36, No. 3 (March 1993), pp. 65–76.

[9] Ueda, K., Optimization of Concurrent Logic Language Implementations. Report of the contract research, Institute for New Generation Computer Technology, 1994 (in Japanese).

[10] Ueda, K. and Morita, M., Moded Flat GHC and Its Message-Oriented Implementation Technique. To appear in *New Generation Computing*, 1994.