Execution model of three parallel languages: OpenMP, UPC and CAF

Ami Marowka
Software Engineering Department, Shenkar College of Engineering and Design, 12 Anna Frank, Ramat-Gan, 52526, Israel
Tel.: +972 3 6110033; Fax: +972 3 7521141; E-mail: amimar2@yahoo.com

Abstract. The aim of this paper is to present a qualitative evaluation of three state-of-the-art parallel languages: OpenMP, Unified Parallel C (UPC) and Co-Array Fortran (CAF). OpenMP and UPC are explicit parallel programming languages based on the ANSI standard. CAF is an implicit programming language. On the one hand, OpenMP designs for shared-memory architectures and extends the base-language by using compiler directives that annotate the original source-code. On the other hand, UPC and CAF designs for distribute-shared memory architectures and extends the base-language by new parallel constructs.
We deconstruct each language into its basic components, show examples, make a detailed analysis, compare them, and finally draw some conclusions.

Keywords: OpenMP, UPC, CAF, parallel programming models

1. Introduction

The last two decades were a very active period in the research of parallel programming models and languages domain [11]. Despite this enthusiastic research activity, a parallel language is still a challenging goal that attracts many scientists in the high-performance computing community.

In this article, we present an analytical study of three modern language-based programming models: OpenMP-C [12] and Unified Parallel C (UPC) [14] and Co-Array Fortran (CAF) [13]. These languages keep the philosophy of making programs concise while giving the programmer the ability to exploit the underlying hardware to gain more performance.

The motivation that drives us to study these three programming languages has two reasons: (1) the development of these languages is backed by industrial consortiums where the major HPC vendors are involved. Thus, deployment of these languages for the benefits of the HPC community is promised; (2) the future road map of microprocessors' development shows that we are in progress toward multiprocessor on-chip based on multithreading technology. The Hyper-Threading technology implemented in the Pentium processor [16] and the Thread-Execution-Engine implemented in UltraSPARC processor [17] are only the start. Therefore, we believe parallel programming will find its way to the commercial computing based on multithreading paradigm. OpenMP and UPC are two programming models that are based on multithreading and ANSI standards.

The main reason that widespread use of parallel language has been lagging behind is that parallel programming is a complex task. A programmer expects that a parallel language will have three major properties: ease-of-use, high abstraction, and portable performance across a wide range of parallel architectures. The design of a parallel language that meets all these expectations is still far from reach.

However, despite the many obstacles, there has been progress in the parallel programming to bring us closer to the desired parallel language. The state-of-the-art parallel languages are converging toward three foundations: (1) two models – the single-address space and multi-address space; (2) three standards – data-parallel, message-passing, and shared-variable; and (3) parallel languages based on well- known serial languages that are extended for parallelism.

The contribution of this paper lies in its analytic study of three contemporary programming languages: OpenMP, UPC and CAF. The development of these languages started before more than half decade ago and they are still in progress. We deconstruct each programming model to its basic components; show examples, make a detailed analysis, compare them, evaluate how far they reach to achieve the desired expectations mentioned above, and draw some conclusions.

The rest of this paper is organized as follows: In Section 2, we study the modeling aspects of each language in detail. Section 3 concludes the paper.

2. Analytical comparison

In this section, each programming model is decomposed into its major elements. We describe, in detail, the characteristics of each one and compare between them. Table 1 summarizes all the analyzed attributes of the three languages.

2.1. Execution model

Fork-join Model. OpenMP consists of a small but powerful set of compiler directives and library routines. The directives extend the sequential programming model with single program multiple data (SPMD) constructs, work-sharing constructs, and synchronization constructs, which provide support for the sharing and privatization of data.

A program written with the OpenMP API begins execution as a single thread of execution called the *master thread*. The master thread executes in a serial region until the first parallel construct is encountered. In the OpenMP API, the parallel directive constitutes a parallel construct. When a parallel construct is encountered, the master thread creates a team of threads, and the master becomes master of the team. Each thread in the team executes the statements in the dynamic extent of a parallel region, except for the work-sharing constructs. Work-sharing constructs must be encountered by all threads in the team in the same order, and the statements within the associated structured block are executed by one or more of the threads. The barrier implied at the end of a work-sharing construct without a *nowait* clause is executed by all threads in the team.

Upon completion of the parallel construct, the threads in the team synchronize at an implicit barrier, and only the master thread continues execution. Any number of parallel constructs can be specified in a single program. As a result, a program may fork and join many times during execution, each time with a different number of threads.

A UPC program is a collection of threads operating in a single global address space, which is logically partitioned among threads. Each thread has an affinity with a private space and a portion of the shared address space. From the compiler's point of view, UPC is a parallel extension to the C Standard for distributed shared-memory architectures that adopt the SPMD programming paradigm. In UPC, the number of threads is specified at compile-time or at run-time and cannot be changed during the program's execution. This is unlike OpenMP, which allowed one to change the number of threads dynamically and to determine a different number of threads for each parallel region.

Co-Array Fortran adopts the Single-Program-Multiple-Data (SPMD) programming model. A single program is replicated a fixed number of times, each replication having its own set of data objects. The number of replications is fixed throughout execution.

Fortran assumes a single program executing alone with a single set of data objects. Each replication of the program is called an *image*. Each image executes asynchronously and the normal rules of Fortran apply, so the execution path may differ from image to image. With the help of a unique image index, the programmer determines the actual path for the image by using normal Fortran control constructs and by explicit synchronizations. The compiler is free to use all its normal optimization techniques for code between synchronizations, as if only one image is present.

The array syntax of Fortran 95 is extended with additional trailing subscripts in square brackets to give a clear and straightforward representation of any access to data that is spread across images. References without square brackets are to local data, so code that can run independently does not need to be changed. Only where there are square brackets, or where there is a procedure call and the procedure contains square brackets, is communication between images involved. There are intrinsic procedures to synchronize images, return the number of images, and return the index of the current image.

Nested Parallelism. In OpenMP, if a thread in a team executing a parallel region encounters another parallel construct, it creates a new team, and it becomes the master of that new team. Nested parallel regions are serialized by default. As a result, by default, a

Table 1
The major attributes of the OpenMP, UPC and CAF languages

Attributes of Model	OpenMP-C	UPC	CAF
Version	2.0	1.1.1	1.0
Execution Model			
Fork-Join	Multiple	Single	Single
Parallelism			
Explicit	Yes	Yes	No
Multithreading	Yes	Yes	No
Expression	Directives	Extensions	Co-Arrays
Paradigm	SPMD	SPMD	SPMD
Orphaned	Yes	No	No
Threads set	Dynamic	Static	Static
Nesting	Yes	No	No
Data Environment			
Data-Sharing	shared, private firstprivate, lastprivate	shared, private	Co-Array
Array Distribution	None	Round-Robin	Co-Array
Dynamic Mem. Alloc.	Non-Collective	Coll. & Non-Coll.	Non-Coll.
Work-sharing			
Mechanism	for, sections, single	for all	None
Schedule Control	static, dynamic, run-time	None	None
Affinity	None	Thread, Address	None
Synchronization			
Mechanism	Critical, Barrier, Lock, Atomic, Order, Flush	Notify-Wait, Barrier Lock, Fence	Sync all, Sync team Sync Memory
Split-Phase	No	Notify-Wait	No
Implicit Barrier	Yes	No	Yes
Synchrony	Blocking	Blocking & Non-Blocking	Blocking
Lock Allocation	Static	Static, Dynamic	Static
Communication		***	
Data Movement	Copyin, Copyprivate	Exchange, Permute Scatter, Gather Broadcast	Implicit
Aggregation	Reduction	Reduce, Prefix-Reduce	Implicit
Computation	None	Sort	None
Programmability			
Serial2Parallel	Stepwise	Re-design	Re-design
Determinacy	Weak	Steady	Weak
Correctness	User	User	User
Portability	Arch. Independent	Arch. Independent	Arch. Dependent

*** Currently, not part of the standard.

nested parallel region is executed by a team composed of one thread. The default behavior may be changed by using either the runtime library function or the environment variable. UPC and CAF do not support nested parallelism.

Orphaned Directives. Another key feature of OpenMP is the concept of the orphaned directive. The OpenMP API allows programmers to use directives in functions called from within parallel constructs. Directives that do not appear in the lexical extent of a parallel construct but may lie in the dynamic extent are called orphaned directives. Orphaned directives give programmers the ability to execute major portions of their program in parallel with only minimal changes to the sequential program. With this functionality, users can code parallel constructs at the top levels of the program call tree and use directives to control execution in any of the called functions.

Bottom-Line. OpenMP presents a flexible and efficient execution model. It has the ability to change the number of running threads during program execution, thus enabling the program to be adapted dynamically to the underlying architecture by matching the number of threads to the available number of processor, or to adjust the number of threads according to the input size. The ability to change, on- the- fly, the scheduling policy of the running threads for dynamic load-balancing, as we discuss further below; the nested parallelism that reflects the multithreading hierarchy of the underlying modern architecture; and its use of orphaned directives makes OpenMP programs more portable, efficient, and easy to use, in comparison to UPC and CAF, which does not support this diverse functionality. Currently, most commercial compilers do not support nested parallelism. However, experiments with NanosCompiler SGI Origion 3000 show improved performance [18, 19].

However, the flexibility of OpenMP is not achieved without any cost. For example, the Fork-join mechanism can incur high overhead when it occurs many times during program execution. One solution is not to kill the threads at the end of the parallel region, but to keep them alive and use them in the next parallel region. This open issue and its implications will be addressed by a future version of OpenMP.

2.2. Data environment

Data Sharing. OpenMP, UPC and CAF enable one to distinguish between shared and private objects. However, each language presents its variations according to its memory architecture and its execution model.

For example, pointers in UPC can be classified based on the locations of the pointers and of the objects to which they point. Accesses to the private area behave identically to regular C pointer operations, while accesses to shared data are made through a special pointer-to-shared construct. The speed of local shared memory accesses will be lower than that of private accesses due to the extra overhead of determining affinity, and remote accesses in turn are typically significantly slower because of the network overhead. There are three different kinds of UPC pointers: private pointers pointing to objects in the threads own private space; private pointers pointing to the shared address space; and pointers living in shared space that also point to shared objects.

OpenMP provides several clauses that allow a user to control the sharing attributes of variables for the duration of the region. Sharing attribute clauses applies only to variables in the lexical extent of the directive on which the clause appears. The private clause declares variables private to each thread in a team. The behavior of a variable specified in a private clause is as follows. A new object with automatic storage duration is allocated for the construct. This allocation occurs once for each thread in the team. The original object referenced by the variable, which has an indeterminate value upon entry to the construct, must not be modified within the dynamic extent of the construct and has an indeterminate value upon exit from the construct.

OpenMP also supports two types of object privatization clauses: *Firstprivate and Lastprivate*. For a Firstprivate clause on a parallel construct, the initial value of the new private object is the value of the original object that exists immediately prior to the parallel construct for the thread that encounters it. For a Firstprivate clause on a work-sharing construct, the initial value of the new private object – for each thread that executes the work-sharing construct – is the value of the original object that exists prior to the point in time that the same thread encounters the work-sharing construct. When a Lastprivate clause appears on the directive that identifies a work- sharing construct, the value of each lastprivate variable from the sequentially last iteration of the associated loop, or the lexically last section directive, is assigned to the variable's original object. The shared clause declares variables such that all threads within a team access the same storage area for shared variables. CAF presents the concept of Co-Array for data sharing as we explain below.

Data Distribution. UPC designs for distribute-shred memory machines and supports mechanisms for handling data distribution over the distributed memory. UPC divides its memory space into two parts: one is for shared memory and the other is for private memory spaces. The shared space is partitioned such that each thread has a unique association (affinity) with a shared partition. The underlying objective is to allow UPC programmers, using proper declarations, to keep the shared data that are dominantly processed by a given thread associated with that thread. Thus, a thread and the data that has affinity to it can easily be mapped by the hardware onto the same physical node. This way, data locality can simply be exploited inherently in applications.

UPC gives the user direct control over data placement through local memory allocation and distributed arrays. Shared arrays can be distributed on a block per thread basis in a round robin fashion (row after row), with arbitrary block sizes. Block size and THREADS determine affinity where element j of a blocked array has affinity to thread ((j/block size) mod THREADS). For example, a declaration of $shared\,[20]\ \ int\,c[100][THREADS]$ means that array c will be distributed among the threads so that the first 20 elements will be on thread 0, the next 20 on thread 1, and so on.

OpenMP is designed as a shared-memory programming model for scientific and engineering applications. Therefore, data distribution mechanisms are not part of the standard model. UPC, likes OpenMP, provides dynamic memory allocation in shared memory but includes an option to choose between *collective and non-collective* allocations. If multiple threads call to a non-collective function then all threads that make the call get different allocations. A collective function has to be called by every thread; it will make a single shared allocation and returns the same pointer value on all threads.

Co-Array Fortran extension to Fortran language allows the programmer to express data distribution by specifying the relationship among memory images in syntax very much like normal Fortran array syntax. Each image has its own set of data objects, all of which may be accessed in the normal Fortran way. Some objects are declared with *co-dimensions* in square brackets immediately following dimensions in parentheses or in place of them.

Unless the array is allocateable, the form for the dimensions in square brackets is the same as that for the dimensions in parentheses for an assumed-size array. The set of objects on all the images is itself an array, called a *co-array*, which can be addressed with array syntax using subscripts in square brackets following any subscripts in parentheses (round brackets), for example:

```
REAL, DIMENSION (N) [*]:: X,Y   X (:) = Y (:)
[Q]
```

declares that each image has two real arrays, X and Y, of size N. The Q in the square brackets attached to array Y is the image reference. In other words, Y (:) [2], is the array Y of image number 2. Therefore, if Q has the same value on each image, the effect of the assignment statement is that each image copies the array Y from image Q and makes a local copy in array X.

Array indices in parentheses follow the normal Fortran rules within one memory image. Array indices in square brackets provide an equally convenient notation for accessing objects across images and follow similar rules. Bounds in square brackets in co-array declarations follow the rules of assumed-size arrays since co-arrays are always spread over all the images. The programmer uses co-array syntax only where it is needed. A reference to a co-array with no square brackets attached to it is a reference to the object in the local memory of the executing image. Since most references to data objects in a parallel code should be to the local part, co-array syntax should appear only in isolated parts of the code. If not, the syntax acts as a visual flag to the programmer that too much communication among images may be taking place. It also acts as a flag to the compiler to generate code that avoids latency whenever possible.

Bottom-Line. Scientific applications usually require mechanisms for distributing and re-distributing arrays of two and three dimensions in different shapes. UPC and CAF currently support only symmetric distribution of one-dimension arrays in a round-robin fashion. Higher functionality of array distribution is essential to achieve scalable performance from large-scale applications. Such functionality permits the ability to adapt the data distribution to the geometry of a NUMA machines.

OpenMP is a shared-memory programming model designs for high-end scientific applications. However, the HPC community uses high-performance parallel machines and large-scale clusters based on NUMA architectures that use distributed-shared memory (DSM). The main obstacle to the adaptation of OpenMP on NUMA architecture stems from the absence of facilities for data placement among processors and threads to achieve data locality. The absence of such a mechanism causes remote memory accesses and inefficient cache memory use, both of which lead to poor performance [5].

SGI, Compaq, and PGI provide high-level directives to specify data distribution and thread scheduling in OpenMP programs [8–10]. A major component in SGI and Compaq directives is the DISTRIBUTION directive. This specifies the manner in which a data object is mapped onto the system memories. Three distribution kinds, namely BLOCK, CYCLIC, and *, are available to specify the distribution required for each dimension of an array. They also offer the DISTRIBUTION RESHAPE directive to perform data distribution at element granularity. Both vendors supply directives to associate computations with the location of data in storage. Compaq provides the NUMA directive to indicate that the iterations of the immediately following PARALLEL DO loop are to be assigned to threads in a NUMA- aware manner. The ON HOME directive informs the compiler exactly how to distribute iterations over memories, and ALIGN is used for specifying alignment of data. Similarly, SGI provides AFFINITY, a directive that can be used to specify the distribution of loop iterations based on either DATA or THREAD affinity. PGI has a different execution model of its HPF-like data distribution directives. Since PGI mainly targets distributed memory systems, it relies on one-sided communication or MPI libraries to communicate among the nodes that may contain more than one processor.

The diversity of the approaches used by the vendors and the different syntaxes harm the portability and the simplicity of the model. Moreover, since OpenMP is hard to realize efficiently on clusters, the current approach is to implement OpenMP on top of software that provides virtual shared memory on a cluster, so-called software distributed shared memory system, such as Omni [6] or TreadMarks [7].

2.3. Work-sharing

Most of the parallelism in OpenMP and UPC is done by work-sharing constructs *for and for all* respectively.

OpenMP distributes the execution of the associated statement among the members of the team that encounter it. The work-sharing directives do not launch new threads and there is no implied barrier on entry to a work-sharing construct. The sequence of work-sharing constructs and barrier directives encountered must be the same for every thread in a team. The *for* directive identifies an iterative work-sharing construct that specifies that the iterations of the associated loop will be executed in parallel. The iterations of the for-loop are distributed across threads that already exist in the team executing the parallel construct to which it binds.

OpenMP also permits associating a schedule policy to the work-sharing construct to allow the user to specify how iterations of the loop will be divided among threads of the team. The iterations are divided into chunks and can be scheduled in one of the following: static, in which the chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number; dynamic, in which each chunk is assigned to a thread that is waiting for an assignment. The thread executes the chunk of iterations and then waits for its next assignment, until no chunks remain to be assigned; guided, in which the iterations are assigned to threads in chunks with decreasing sizes. When a thread finishes its assigned chunk of iterations, it is dynamically assigned another chunk, until none remains; and run-time, in which the decision regarding scheduling is deferred until runtime. The schedule type and size of the chunks can be chosen at run time by setting special environment variables. UPC does not support such functionality.

The *upc forall* loop behaves like a C for-loop, except that the programmer can specify an affinity expression. It has four fields, the first three of which are similar to those used for a regular C for-statement. The fourth field is called affinity and is used to indicate that the thread, which has the element v [i], will be executing the i-th iteration. Thus, unnecessary remote accesses can be avoided. The fourth field can also be an integer type. In this case, the thread number (i mod THREADS) will be executing the i-th iteration. OpenMP does not support thread affinity control.

Co-Array Fortran does not provide work-sharing constructs. The work distribution is done by setting a different thread of execution to each one of the program's images. The programmer determines the actual path for the image with the help of a unique image index by using normal Fortran control constructs and by explicit synchronizations.

Bottom-Line. On the one hand, OpenMP is designed for shared-memory architecture. In practice, it is used also on top of distribute-shared machines. Adding the option to control the threads affinity can change dramatically the performance of the applications. On the other hand, UPC can run on top of shared-memory machines and thus the load balancing of the applications can be improved by adding scheduling policies associate to the work-sharing construct. In CAF, the programmer does the work sharing. CAF also lacks of mechanism like scheduling policies to control the workload to improve performance.

2.4. Synchronization model

OpenMP, UPC and CAF are shared-memory programming environments. As such, they support high-level synchronization mechanisms that are used to solve conflicts when multiple threads have access to shared objects. They introduce classic synchronization constructs together with special variations of these constructs. The classic synchronization mechanisms include look-unlock functions, barrier synchronization, and flush operations.

The *flush* operation denotes a sequence point where a thread creates a consistent view of memory. That is, all memory operations (both reads and writes) defined prior to the sequence point must be completed. All memory operations (both reads and writes) defined after the sequence point must follow the flush, and variables in registers or write buffers must be updated in memory.

In addition to these mechanisms, OpenMP offers *critical, atomic, and ordered* directives. The *critical* directive identifies a construct that restricts execution of an associated structured block to a single thread at a time. The *atomic* directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple simultaneous writing threads. The *ordered* directive, which must be within the dynamic extent of a *for or parallel for* construct, forces the structured block following the directive to be executed in the order in which iterations would be executed in a sequential loop.

UPC, on the other hand, presents the concept of split-phase barriers. The *upc notify and upc wait* pair provides a split-phase barrier, which allows the notify phase to be separated from the wait phase with local work. This allows asynchronous computation, which increases the efficiency and the performance of scalable applications.

Co-Array Fortran adds a global barrier synchronization, *sync all*(), which requires all operations before the call on all images to be completed before any image advances beyond the call. In practice, it is often sufficient, and faster, to only wait for the relevant images to arrive. *Sync all*(*wait=list*) provides this functionality. There is also *sync team*(*team=team*) and *sync team*(*team=team*, *wait=list*) for cases where only a subset, team, of all images are involved in the synchronization. The intrinsic pairs *start critical and end critical* provide a basic critical region capability. It is also possible to write your own synchronization routines, using the basic intrinsic *sync memory.* This routine forces the local image to both complete any outstanding co-array writes into "global" memory and refresh from global memory any local copies of co-array data it might be holding (caches, registers etc.). A call to sync memory is rarely required in Co-Array Fortran, because there is an implicit call to this routine before and after virtually all procedure calls including Co-Array's built in image synchronization intrinsic. This allows the programmer to assume that image synchronization implies co-array synchronization.

Bottom-Line. OpenMP, UPC and CAF implementations are not required to check for dependencies, conflicts, deadlocks, race conditions, memory consistency, or other problems that result in incorrect program execution. The user is responsible for ensuring that the applications using these languages execute correctly. Neither frees the programmer from the most tedious and cumbersome tasks of parallel programming.

2.5. Communication model

Generic collective operations are essential mechanism for parallel computations where all the threads are joined together to execute a specific task. The efficient realization of these operations is crucial for the scalability of large-scale scientific applications.

OpenMP supports *reduction* operation and two more collective functions: *copyin, and copyprivate*. The *copyin* clause provides a mechanism to assign the same value to threadprivate variables for each thread in the team executing the parallel region. For each variable specified in a copyin clause, the value of the variable in the master thread of the team is copied, as if by assignment, to the thread-private copies at the beginning of the parallel region. The *copyprivate* clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members. It is an alternative to using a shared variable for the value when providing such a shared variable would be difficult.

The current standard of UPC does not support collective functions. However, a proposal draft in-progress will be part of the spec in the next major version [2]. According to this proposal, UPC will provide two sets of functions: *Re-localization Operations and Computational Operations*. *Re-localization operations* include the functions broadcast, scatter, gather, exchange, and permute. *Computational operations* include the functions reduce, prefix-reduce, and sort. Reduce and prefix-reduce functions are reduction operations over an associative operators ($+, *, -$, &, ˆ, $|$, &&, $\|$). The function sort takes shared array elements and sorts them in place in ascending order.

In CAF, the communication among images, which rise on different processors, is done implicitly when co-array statements are used. For example, the statement X = Y [PE] is actually a *get* operation from Y [PE]; Y [PE] = X is a *put* operation into Y [PE]; Y [:] = X is a *broadcast* operation on X; and S = MINVAL(Y [:]) is a MIN *reduction* operation over all Y.

Square brackets attached to objects in an expression or an assignment is a sign of communication between images. However, CAF cannot use overlapping of computation and communication because the compiler does the communication in CAF implicitly and synchronously and it is largely outside the programmer's control.

Bottom-Line. OpenMP supports a small set of collective operations. Operations like broadcast, scatter, and gather are not supported because they are less used in shared memory programming environments. However, as we have already mentioned, many OpenMP applications run on top of distribute-shared memory machines where such operations are needed. The UPC proposal presents a rich set of collective operations. However, it remains to be seen how effectively they will be realized.

2.6. Programmability

OpenMP, UPC and CAF languages were designed with the same principles in mind: simplicity and portability. They start with standard language as a base-language and add parallel functionality on top of it. OpenMP parallelism is done by annotating the code with compiler directives, while UPC and CAF add new parallel structures and type qualifiers.

One advantage that OpenMP has is that if an OpenMP program is designed to work when there is exactly one thread, it is then also a legal program. In such a case, the serial compiler ignores the compiler directives. Moreover, although UPC is a small set of extensions to C, C programmers unfamiliar with UPC have to understand what UPC programs do. For example, a programmer has to be aware that adding a for-loop into the body of a program could change the multi-threading behavior of that program. In contrast, adding a for-loop, or making any other modifications outside a parallel region, to OpenMP is identical, in effect, to making the same change to the serial program without the directives.

CAF looks and feels like Fortran and requires Fortran programmers to learn only a few new rules. The few new rules are related to two fundamental issues that any parallel programming model must resolve: work distribution and data distribution. Co-Array Fortran can in principle also work on shared nothing systems. Since co-array syntax is incorporated into the language, it is more flexible and more efficient than any library implementation such as MPI or SHMEM. The synchronization in Co-Array Fortran is simple to use as compared to other process-based SPMD programming models. However, the notation of square brackets for representing images can be very complex, hard to understand, and error prone. The programmer may avoid using complex descriptions of co-arrays and instead adopting simple structures and statements wherever it is possible.

OpenMP is primarily designed for fine-grained loop parallelization, which is typically appropriate for small node counts. Therefore, the lack of layout control is less of an issue for OpenMP in its primary domain of interest, though it is a concern for SPMD programs. In Co›Array Fortran, all memory is associated with a local image, so memory placement on NUMA systems is simple to arrange and less affects scalability. Thus, CAF has clear advantages on systems with large number of nodes.

OpenMP supports programs that execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library). However, it is possible and permissible to develop a program that does not behave correctly when executed sequentially. This leads to non-deterministic programming where different degrees of parallelism may result in different numeric results because of changes in the association of numeric operations. For example, a serial addition reduction may have a different pattern of addition associations than a parallel reduction. These different associations may change the results of floating-point addition

OpenMP, UPC and CAF implementations are not required to check for dependencies, conflicts, deadlocks, race conditions, or other problems that result in incorrect program execution. The user is responsible for ensuring that the applications using these languages execute correctly. Neither frees the programmer from the most tedious and cumbersome tasks of parallel programming.

3. Conclusions

OpenMP, UPC and CAF are new based-languages programming models that are still in development. On the one hand, they present relatively simple programming styles: annotating a serial code by compiler directives (OpenMP) or by using simple parallel constructs (UPC); explicit parallelism; SPMD programming paradigm; high abstraction due to separation of logical parallelism from physical execution environment; and high-level of machine portability. On the other hand, the responsibility of the programmer to ensure the correctness of the program is burdensome: checking for dependencies, conflicts, deadlocks, race conditions, or other problems that result in incorrect program execution.

OpenMP stands at a crossroad. It is designed for shared memory architectures and scientific applications that demand high-speed computations. This can be achieved only by extending OpenMP for large-scale shared-distributed memory machines. There are two tested-ways to achieve this goal. On the one hand, HPF-like compiler directives for supporting data-distribution can extend OpenMP. A few vendors have already extended their compilers by such mechanisms [8–10]. This is the easy way. On the other hand, OpenMP can be integrated with MPI for creating a coherent hybrid model [15]. This is the hard way, but a much more promising one. In addition, OpenMP has many open issues that will be addressed in the future [20]. Among them are adding semaphores into OpenMP, better handling of threads group, support of wavefront loop nests, and precedence relations [21].

UPC was designed a priori for NUMA architectures. It introduces a small set of powerful constructs. However, there are major proposals that are waiting for realization. These proposals deal with a new memory consistency model, a library for collective communications, and a library for parallel I/O.

CoArray Fortran can take full advantage of a hardware global memory, but it can also be used on shared-nothing systems with physically distinct memories connected by a network. It requires the fewest changes and should be as fast as SHMEM and MPI, but is not widely available. Cray T3E, T3D and X1 are the only machines with a CoArray Fortran compiler today and it implements only a subset of the language. Currently, Rice University is working to create an open-source, portable, high-quality CAF compiler [23]. Co-Array Fortran has a simple syntax as compared to other alternatives for SPMD Fortran, but without a portable compiler or support from several major vendors, it is not a viable portability tool.

OpenMP, UPC and CAF are another phase in the evolutionary process of the parallel programming models and languages. On the one hand, OpenMP simplicity and its nested parallelism, that match the hierarchical design of future microprocessor, are more ready to the commercial parallel programming. On the other hand, UPC and CAF can be the parallel programming of choice for the scientific community. While only time will tell which of these scenarios will become real, the above study offers a glimpse into the technological potential and the evolving state of the art.

References

[1] T. El-Ghazawi, W. Carlson and J. Draper, *UPC Language Specifications*, version 1.1, 2003. http://www.gwu.edu/up/documentation.html

[2] E. Wiebel, D. Greenberg and S. Seidel, *UPC Collective Operations Specifications*, pre4V1.0, April 2, 2003.

[3] A.J. Wallcraft, A Comparison of Co-Array FORTRAN and OpenMP FORTRAN for SPMD Programming, *Journal of Supercomputing* **22**(3) (Jul 2002), 231–250.

[4] A.J. Wallcraft, A Comparison of Several Scalable Programming Models. Technical Report, Naval Research Laboratory, October 9, 1998.

[5] A. Marowka, Z. Liu and B. Chapman, OpenMP-Oriented Applications for Distributed Shared Memory Architectures, *Concurrency & Computation: Practice & Experience Journal* **16**(3) (March, 2004).

[6] M. Sato, S. Satoh, K. Kusano and Y. Tanaka, *Design of OpenMP compiler for an SMP cluster*, Proc. EWOMP 99, Lund, 1999.

[7] C. Amza, A. Cox et al., TreadMarks: Shared memory computing on networks of workstations, *IEEE Computer* **29**(2) (1996), 18–28.

[8] Silicon Graphics Inc. Parallel Processing on Origin Series Systems MIPSpro 7. FORTRAN 90 Commands and Directives Reference Manual.

[9] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J. Harris, C.A. Nelson and C.D. Offner, Extending OpenMP for NUMA machines, *Scientific Programming* **8**(3) (2000).

[10] J. Merlin, D. Miles and V. Schuster, *Distributed OMP: Extensions to OpenMP for SMP clusters*, In EWOMP 2000, Second European Workshop on OpenMP, Edinburgh, Scotland, U.K., September 14–15, 2000.

[11] D.B. Skillcorn and D. Talia, Models and Languages for Parallel Computation, *ACM Computing Surveys* **30**(2) (June 1998).

[12] OpenMP C and C++ Application Program Interface, http://www.openmp.org.

[13] R.W. Numrich and J. Reid, Co-array Fortran for parallel programming, *Fortran Forum* **17**(2) (1998), 1–31.

[14] T.A. El-Ghazawi, W.W. Carlson and J.M. Draper, *UPC Language Specifications,* V1.0 (http://upc.gwu.edu). February, 2001.

[15] R. Rabenseifner and G. Wellein, *Communication and Optimization Aspects of Parallel Programming Models on Hybrid Architectures*, proceeding of EWOMP 2002, Sep. 18–20, Roma, Italy.

[16] R. Yung, S. Rusu and K. Shoemaker, *Future Trend of Microprocessor Design*, Invited paper, In Proceeding of 28th European Solid-State Circuits Conference, September 24–26, 2002, Firenze, Italy.

[17] UltraSPARC-IV Processor Architecture Overview. Technical Whitepaper Version 1.0, February 2004.

[18] G. Jost, J. Labarta and J. Gimenez, *What Multilevel Parallel Programs Do When You Are Not Watching*, Presentation in WOMPAT Workshop, June 16–18 2004, Houston, Texas, USA.

[19] G. Jost, H. Jin, J. Labarta, J. Gimenez and J. Caubet, *Performance Analysis of Multilevel Parallel Applications on Shared Memory Architectures*, International Parallel and Distributed Processing Symposium (IPDPS 2003), Nice (France), April 2003.

[20] OpenMP Version 3.0 Proposed Features, personal communication.

[21] A. Marowka, Extending OpenMP for Task Parallelism, *Parallel Processing Letters* **13**(3) (2003), 341–352.

[22] D. Hisley, G. Agrawal, P. Satya-Narayana and L. Pollock, Porting and performance evaluation of irregular codes using OpenMP, *Concurrency Practice and Experience* **12**(12) (Oct. 2000), 1241–1259. Publisher: Wiley, UK.

[23] Co-Array Fortran at Rice University, http://hipersoft.cs.rice.edu/caf/.