Analytic Comparison of Two Advanced C Language-Based Parallel Programming Models

Ami Marowka
School of Computer Science and Engineering
The Computer Aided Design Laboratory
The Hebrew University of Jerusalem
Jerusalem, Israel, 91904
Email: amimar@cs.huji.ac.il

Abstract— **There are two main approaches for designing parallel language. The first approach states that parallel computing demands new programming concepts and radical intellectual changes regarding the way we think about programming, as compared to sequential computing. Therefore, the design of such a parallel language must present new constructs and new programming methodologies. The second approach states that there is no need to reinvent the wheel, and serial languages can be extended to support parallelism. The motivation behind this approach is to keep the language as friendly as possible for the programmer who is the main bridge toward wider acceptance of the new language.**

In this paper we present a qualitative evaluation of two contemporary parallel languages: OpenMP-C and Unified Parallel C (UPC). Both are explicit parallel programming languages based on the ANSI C standard. OpenMP-C was designed for shared-memory architectures and extends the base-language by using compiler directives that annotate the original source-code. On the other hand, UPC was designed for distribute-shared memory architectures and extends the base-language by new parallel constructs.

We deconstruct each parallel language into its basic components, show examples, make a detailed analysis, compare them, and finally draw some conclusions.

I. Introduction

The last two decades were a very active period in the research of parallel programming models and languages domain [16]. Despite this enthusiastic research activity, a parallel programming language is still a challenging goal that attracts many scientists in the high-performance computing community.

The main reason 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.

But despite the pessimism, there has been enough progress in the parallel programming to bring us closer to the desired parallel language. The state-of-the-art parallel languages are based on three firm foundations:

1) The native models are converging toward two models - the single-address space and multi-address space;
2) the high-level parallel programming models are converging toward three standards - data-parallel, message-passing, and shared-variable; and
3) the parallel languages are based on well-known serial languages that are extended for parallelism.

In this article, we present an analytical study of two modern language-based programming models: OpenMP-C [17] and Unified Parallel C (UPC) [21]. Both capitalize on the experience gained from their predecessor languages such as Split-C [18], AC [19], C// [22] and PCP [20]. These languages keep the C philosophy of making programs concise while giving the programmer the ability to exploit the underlying hardware to gain more performance.

OpenMP and UPC are explicit parallel programming languages based on the ANSI C standard. OpenMP-C was designed for shared-memory architectures and extends the base-language by using compiler directives that annotate the original source-code. In contrast, UPC was designed for distribute-shared memory architectures and extends the base-language by new parallel constructs. In this paper we present a qualitative evaluation of these two contemporary programming models. We deconstruct each model to its basic components; show examples, make a detailed analysis, compare them, and draw some conclusions.

The rest of this paper is organized as follows: In Section 2, we present a brief overview of the languages and show a sample application. In Section 3, we study the modeling aspects of each language in detail. Section 4 concludes the paper.

II. Language Overview

In this section, the parallel languages considered in this paper are briefly reviewed, followed by simple examples to demonstrate the main features of the languages.

A. *Unified Parallel C*

Unified Parallel C (UPC) is a parallel extension of the C programming language. Its aim is to support scalable scientific applications running on distribute-shared memory architectures. The language is based on the SPMD programming paradigm, where every thread runs the same program but keeps its own private local data. Each thread has a unique integer identity expressed as the MYTHREAD variable, and the THREADS variable represents the total number of threads used by the application. These two variables are predefined identifiers. In addition to each thread's private address space, UPC provides a shared memory area to facilitate communication among threads, and programmers can declare a shared object by specifying the shared type qualifier. While a private object may only be accessed by its owner thread, all threads can read or write to data in the shared address space. Because the shared memory space is logically divided among all threads, from a thread's perspective the shared space can be further divided into a local shared memory and remote one. Data located in a thread's local shared space are said to have *affinity* with the thread, and compilers can utilize this affinity information to exploit data locality in applications to reduce communication overhead.

UPC language is mainly characterized by four sets of constructs and type qualifiers: pointer type qualifiers to distinguish between local and remote memory accesses; upc_forall work-sharing construct to exploit data parallelism and affinity control; array type qualifiers for data distribution; and memory consistency models directives.

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 thread's own private space, private pointers pointing to the shared address space, and pointers living in shared space that also point to shared objects.

Data Distribution. UPC gives the user direct control over data placement through local memory allocation and distributed arrays. When declaring a shared array, programmers can specify a block size in addition to the dimension and element type. The system uses this value to distribute the array elements block by block in a round- robin fashion over all threads. For example, a declaration of *shared [2] int array[10]* means that the compiler should allocate the first two elements of the array on thread 0, the next two on thread 1, and so on. If the block size is omitted the value defaults to one (cyclic layout), while a layout of [ ] or [0] indicates indefinite block size, i.e., that the entire array should be allocated on a single thread.

Work-sharing. To simplify the task of parallelization, UPC also includes a built in upc_forall loop that distributes iterations among the threads. The upc_forall loop behaves like a C for loop, except that the programmer can specify an affinity expression whose value is examined before every loop's iteration. The affinity expression can be two different types: if it is an integer, the affinity test checks if its value modulo THREADS is the same as the id of the executing thread; otherwise, the expression must be a shared address and the affinity test checks if the running thread has affinity to this address. The affinity expression can also be omitted, in which case the affinity test is vacuously true and the loop behaves as if it is a C *for* loop. A thread executes iteration only if the affinity test succeeds, and the upc_forall language construct thus provides an easy-to-use syntax to distribute the computation load to match the data layout pattern.

Memory Consistency Model. UPC provides a hybrid, user-controlled consistency model for the interaction of memory accesses in shared memory space. Each memory reference in the program may be annotated to be either strict or relaxed. Under strict behavior, the program executes in a sequential consistency model. This implies that the user can be sure that it appears to all threads that the strict references in a thread appear in the order they are written, relative to all other accesses. To implement this, the compiler must take into account all memory accesses in all threads of the program when determining that a strict reference in a thread may be reordered with respect to other shared references. Under relaxed behavior, the program executes in a "local" consistency model. This implies that the user can assumes that it appears to the issuing thread that all shared references in that thread occur in the order they were written. Here the compiler needs only analyze the shared memory accesses in the local thread to allow reordering. Note that because each reference may be annotated, a number of models between local and sequential consistency are available to the user.

The general method of using these models is that the programmer will first establish a default environment by including either $< upc\_strict.h >$ or $< upc\_relaxed.h >$. These files contain, respectively, $\#pragma\_upc\ strict$ and $\#pragma\_upc\ relaxed$ directives. All non-annotated references are within functions defined after these directives operate under the selected model. The programmer then may annotate more explicitly those references to be handled differently. One method for annotating references is to declare shared variables and pointers with the type qualifiers strict and relaxed. All references to annotated variables and through annotated pointers will operate under the selected model, regardless of the default behavior enforced. Other notable features of UPC language include dynamic allocation functions and synchronization constructs [1]. Features to be added in the next versions of the language include a Parallel I/O API [2], Collective Operations [3], and extension to the UPC Memory Model definition [4].

B. OpenMP

OpenMP is a parallel programming model for writing multi-threaded applications in a shared memory environment. It consists of a set of compiler directives and library routines. The compiler generates a multi-threaded code based on the specified directives. OpenMP is essentially a comparatively recent standardization SMP (Symmetric Multi-Processor) development and practice. By using OpenMP, it is relatively easy to create parallel applications in FORTRAN, C, and C++.

Compiler and third-party applications support is becoming more common.

An OpenMP program begins with a single thread of execution called the *master* thread. The master thread spawns teams of threads in response to OpenMP directives, which perform work in parallel. Parallelism is thus added incrementally: the serial program evolves into a parallel one. OpenMP directives are inserted at key locations in the source code. These directives take the form of $\#pragmas$. The compiler interprets the directives and creates the necessary code to parallelize the indicated tasks/regions. The *parallel region* is the basic construct that creates a team of threads and initiates parallel execution. Most OpenMP directives apply to structured blocks, which are blocks of code with one entry point at the top and one exit point at the bottom.

The number of threads created when entering parallel regions is controlled by the value of the environment variable OMP_NUM_THREADS. The number of threads can also be set by a function call from within the program, which takes precedence over the environment variable. It is possible to vary the number of threads created in subsequent parallel regions. Each thread executes the block of code enclosed by the parallel region. In general, there is no synchronization between threads. Different threads may reach the end of the parallel region at different times. OpenMP does provide constructs for synchronization, but the code should not be written in such a way that its output depends upon different threads executing statements at particular times. OpenMP provides a number of constructs for thread synchronization and coordination, among them critical, atomic, barrier, and master. These are sufficient for many needs, but OpenMP also provides a set of runtime thread-locking functions that can be used for fine control. When all threads reach the end of the parallel region, all but the master thread go out of existence and the master continues alone. The OpenMP directive clauses - Private, Shared, and Default - control whether the listed variables are shared among different threads, or are private (local) to each thread.

OpenMP provides several constructs for sharing work among threads in a team. These are: Parallel for/DO, Parallel Sections, Work-share, and Single directives. These constructs are placed inside an existing parallel region. The result is to distribute execution of associated statements among the existing threads. A number of environmental variables may be set to control aspects of OpenMP execution; such aspects are the number of threads, loop scheduling, enabling of nested parallelism, and dynamic adjustment of the number of threads.

OpenMP also provides a number of routines that may be called from within one's code. These may be used to get and set the number of threads, enable or disable dynamic thread allocation, and to check whether the code is executing in parallel. Changes in the runtime environment made by these routines take precedence over the corresponding environment variables.

C. A Sample Application

This section present a solution to a simple numerical integration problem using each language. The parallel program computes an approximation to π by using numerical integration to find the area under the curve $4/(1 + x^2)$ between 0 and 1.

The interval [0, 1] is divided into num_steps subintervals of width 1/ num_steps. For each of these intervals the algorithm computes the area of a rectangle whose height is such that the curve $4/(1 + x^2)$ intersects the top of the rectangle at its midpoint. The sum of the areas of the num_steps rectangles approximates the area under the curve. Increasing num_steps reduces the difference between the sum of the rectangle's area and the area under the curve.

Listing 1 contains a UPC program to compute π using numerical integration. The default memory consistency behavior, defined in line 1, is set to relaxed mode. Under a relaxed policy the compiler can order data accesses in shared space in any way that can optimize the execution time. We can do so because the algorithm is data-parallel and the areas of all rectangles can be computed simultaneously. Lines 2-3 declare a lock variable lk and a shared variable pi respectively. The parameter num_steps, declared in line 4, is the number of subintervals. In line 10 we compute the steps of each subinterval.

The parallelization of the program is done in line 11 by using the upc_forall work- sharing construct. This construct explicitly distributes the iterations among the threads. Upc_forall has four fields; the first three are similar to those found in the for loop of the C programming language. The fourth field is called *affinity* and in this case it indicates that the thread (i mod THREADS) executes the i-th iteration. This guarantees that iterations are executed without causing any unnecessary remote accesses. In line 12 we compute in parallel the midpoint of each of the subintervals, and in line 13 we compute in parallel the height of the function curve at each of these points. Line 15 is a critical section which adds the heights of all of the rectangles. The boundaries of the critical section are defined by the pair functions upc_lock and upc_unlock and the lock variable lk in lines 14 and 16 respectively. Finally, we compute in line 17 the total height by the rectangles' step to yield the area under the curve.

We now turn to the OpenMP-C example. Listing 2 contains an OpenMP-C program to compute π using numerical integration. The semantic behavior of the program is the same as the previous UPC program. The compiler directive inserted into the serial program in line 8 contains all the information needed for the compiler to parallelize the program. The compiler interprets the directive as shown below. $\#pragmaomp$ is the directive's sentinel. The *parallel* keyword defines a parallel region, lines 9-12, that is to be executed by NUM_THREADS threads in parallel. The NUM_THREADS is defined in line 3 and the omp_set_num_threads function sets the number of threads to use for subsequent parallel regions in line 7. There is an implied barrier at the end of a parallel region. Only the master thread of the team continues execution at the end of a parallel region.

The *for* keyword identifies an iterative work-sharing construct that specifies the iterations of the associated loop that will be executed in parallel. The iterations of the *for* loop are distributed across the threads in a round-robin fashion in the order of the thread number. The private(x) clause declares the variable x to be private to each thread in the team. A new object with automatic storage duration is allocated for the construct. This allocation occurs once for each thread in the team.

The *reduction (+:sum)* clause performs a reduction on the scalar variable *sum* with the operator +. A private copy of each variable sum is created, one for each thread, as if the private clause had been used. At the end of the region for which the reduction clause was specified, the original object is updated to reflect the result of combining its original value with the final value of each of the private copies using the + operator. The reduction operator + is associative, and the compiler may freely re- associate the computation of the final value. The value of the original object becomes indeterminate when the first thread reaches the containing clause and remains so until the reduction computation is complete. The computation will be completed at the end of the construct.

Listing 1. UPC program to compute π using numerical integration.

```
#include <up_relaxed.h>
upc_lock_t lk;
shared double pi;
static long num_steps = 100000; double step;
void main (void)
{
int i; double local_pi , x;
step = 1.0/ (double) num_steps;
upc_forall (i=1; i<=num_steps; i++; i % THREADS)
  {x = (i-0.5)*step;
   local_pi = local_pi + 4.0 / (1.0+x*x) ;}
upc_lock (&lk);
pi += local_pi;
upc_unlock (&lk);
pi *= step;
}
```

Listing 2. OpenMP-C program to compute π using numerical integration.

```
#include <omp.h>
static long num_steps = 100000; double step;
#define NUM_THREADS 2
void main ()
{ int i; double x, pi, sum = 0.0;
step = 1.0/ (double) num_steps;
omp_set_num_threads (NUM_THREADS)
# PRAGMA OMP PARALLEL FOR
# PRAGMA OMP REDUCTION (+:SUM) PRIVATE(x)
for (i=1; i<= num_steps; i++) {
     x = (i-0.5)*step;
     sum = sum + 4.0/ (1.0+x*x);
}
pi = step * sum;
}
```

III. Analytical Comparison

In this section, each programming model is decomposed to its major elements. We describe here the characteristics of the execution model, the memory consistency model and the programmability of each one and compare between them. The full version of this paper contains more details [24]. Table 1 summarize all the analyzed attributes of the two languages.

A. Execution Model

OpenMP uses the *fork-join* model of parallel execution. Although this fork-join model can be useful for solving a variety of problems, it is somewhat tailored for large array-based applications.

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.

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 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.

If a thread modifies a shared object, it affects not only its own execution environment, but also those of the other threads in the program. The modification is guaranteed to be complete, from the point of view of one of the other threads, at the next sequence point (as defined in the base language) only if the object is declared to be volatile. Otherwise, the modification is guaranteed to be complete after the modifying thread, and then (or concurrently) the other threads, encounter a flush directive that specifies the object (either implicitly or explicitly). When the flush directives that are implied by other OpenMP directives are not sufficient to ensure the desired ordering of side effects, it is the programmer's responsibility to supply additional, explicit flush directives.

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.

OpenMP consists of a small but powerful set of compiler directives and library routines. The directives extend the C 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. Although, OpenMP has been designed in mind for SPMD programs, the current standard has some missing features that inhibit building viable and portable SPMD application, as was outlined by Wallcraft [6], [7]. For example, the behavior of the exit statement in the body of a SPMD program has not been defined and such an exit is in fact illegal in some implementations.

Another key feature of OpenMP is the concept of the *orphaned* directive. The OpenMP C 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.

UPC. From the user's perspective, 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.

A UPC program is translated under either a ”static THREADS” environment or a ”dynamic THREADS” environment. Under the static THREADS environment, the number of threads to be used in execution is indicated by the translator in an implementation-defined manner. If the actual execution environment differs from the number of threads, the behavior of the program is undefined. In other words, the number of threads is specified at compile-time or at run-time and cannot be changed during the program's execution, unlike OpenMP, which allowed one to change the number of threads dynamically and to determine a different number of threads for each parallel region.

A UPC program has a single fork-join pair. There is an implicit UPC barrier at program startup and termination. Except as explicitly specified by UPC barrier operations or by certain library functions, there are no other barrier synchronization guarantees among the threads. Each thread has local data storage on which it operates and which are logically divided into a private portion and a shared portion. The logical association of a portion of the shared address space with a given thread is called *affinity*. Thus, each thread may access shared data that have an affinity to any thread. For access to the memory space, UPC supports private and shared pointers. A shared pointer can reference all locations in the shared space while a private pointer may reference only addresses in its private space or addresses in its own portion of the shared space.

B. Memory Consistency Model

The aim of the memory consistency model in a language is to define the order in which the results of write operations may be observed through read operations [5]. The behavior of UPC and OpenMP programs may depend on the timing of accesses to shared variables, so a program defines a set of possible executions, rather than a single execution. Such a model affects the performance, correctness, programmability, as well as the portability of parallel applications.

OpenMP supports only the *weak consistency* model - The programmer uses synchronization operators to enforce sequential consistency. OpenMP is 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 OpenMP application executes correctly.

UPC supports two memory consistency models to determine the order of data accesses in the shared space. Under a strict memory consistency, accesses take place in the same order that would have resulted from sequential execution. Under a relaxed policy, however, the compiler can order such accesses in any way that can optimize the execution time.

UPC provides memory consistency control at the single object level, statement block level, and full program level. The programmers are responsible for using the correct consistency model. References to shared objects may be either strict or relaxed. For relaxed references there is no change to the C standard execution model as applied to an individual thread. This implies that translators are free to reorder and/or ignore operations (including shared operations) as long as the restrictions found in the base language are observed. A further restriction applies to strict references. For each strict reference, the restrictions found in the base language must be observed with respect to all threads if that reference is eliminated (or reordered with respect to all other shared references in its thread).

Equally, the behavior of strict shared references can be defined as follows. Label each access to a shared object S(i, j) or R(i, j), where S represents a strict shared access (read or modify), R represents a relaxed shared access (read or modify), i is the thread number making the access, and j is an integer which monotonically increases as the evaluation of the program (in the abstract machine) proceeds from startup through termination. The ”abstract order” is a partial ordering of all accesses by all threads such that an access x(a,b) occurs before y(c,d) in the ordering if $a == c$ and $b < d$. The ”actual order (k)” for thread k is another partial order in which x(a,b) occurs before y(c,d) if thread k observes the x access before it observes the y access. A thread observes all accesses present in the abstract order which affect either the data written to files by it or its input and output dynamics as described in the base language. The least requirements on a conforming implementation are that:

Attributes of Model	OpenMP-C	UPC
Version	2.0	1.1.1
Execution Model		
Fork-Join	Multiple	Single
Parallelism		
Explicit	Yes	Yes
Multithreading	Yes	Yes
Expression	Compiler Directives	Language Extensions
Paradigm	SPMD	SPMD
Orphaned	Yes	No
Threads set	Dynamic	Static
Nesting	Yes	No
Data Environment		
Data-Sharing	shared, private fi rstprivate, lastprivate	shared, private
Array Distribution	None	Round-Robin
Dynamic Memory Allocation	Allocation Non-Collective	Collective & Non-Collective
Work-sharing		
Mechanism	for, sections, single	for_all
Schedule Control	static, dynamic, guided, run-time	None
Affi nity	None	Thread-based, Address-based
Synchronization		
Mechanism	Critical, Barrier, Lock, Atomic, Order, Flush	Notify-Wait, Barrier Lock, Fence
Split-Phase	No	Notify-Wait
Implicit Barrier	Yes	No
Synchrony	Blocking	Blocking & Non-Blocking
Lock Allocation	Static	Static, Dynamic
Memory Consistency		
Model	Weak	Strict & Relaxed
Control level	None	Object, Block, Program
Communication Model		***
Data Movement	Copyin, Copyprivate	Exchange, Permute, Broadcast Scatter, Gather
Aggregation	Reduction	Reduce, Prefi x-Reduce
Computation	None	Sort
Parallel I/O		***
Namespace	Single & shared	Multiple & shared
W/R Operations	Non-deterministic	Deterministic
Parallel functions	None	open, close, seek, read, write
Special I/O	None	Asynchronous, List
Programmability		
Serial2Parallel	Stepwise	Re-design
Determinacy	Weak	Steady
Correctness Responsibility	User	User
Portability	Architecture Independent	Architecture Independent
*** Currently, not part of the standard		

TABLE I
THE MAJOR ATTRIBUTES OF THE OPENMP-C AND UPC LANGUAGES.

- x(a,b) must "occur before" y(c,d) in actual order(e) if $a == c$ and $a == e$ and $b < d$
- x(a,b) must "occur before" y(c,d) in actual order(e) if $a == c$ and $b < d$ and ((x == S) or (y == S)), for all e

unless such a restriction has no effect on either the data written into files at program termination or the input and output dynamics requirements described in the base language. References to shared objects, either directly or via pointer-to-shared indirection.

C. Programmability

OpenMP-C and UPC languages were designed with the same principles in mind: simplicity and portability. Both start with standard C 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 adds new parallel structures and type qualifiers.

OpenMP-C and UPC 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. Both do not free the programmer from the most tedious and cumbersome tasks of parallel programming.

OpenMP is intended to support programs that will 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.

Another issue related to the non-deterministic behavior of OpenMP is *thread-safe*. All standard C library functions and built-in functions (that is, functions of which the compiler has specific knowledge) must be thread-safe. Unsynchronized use of thread- safe functions by different threads inside a parallel region does not produce undefined behavior. However, the behavior might not be the same as in a serial region. (A random number generation function is an example.)

OpenMP is a shared memory model and is intended for the scientific community. However, this community uses high-performance parallel machines that are multiprocessor machines with distributed-shared memory (DSM) based on NUMA architecture. The main stumbling block 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.

SGI, Compaq, and PGI provide high level directives to specify data distribution and thread scheduling in OpenMP programs [13], [14], [15]. 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 syntax 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 [9] or TreadMarks [10].

One advantage that OpenMP has over UPC is that if an OpenMP program is designed to work when there is exactly one thread, it is then also a legal C program. The compiler directives have no effect on one thread and are ignored by the serial C compiler. A library of a few standard procedures is required, but is trivial to implement for a single thread. This is not the case, though, for UPC.

UPC is not a large extension to C but UPC programs cannot be maintained by C programmers unfamiliar with UPC. For example, a programmer has to be aware that adding a FOR loop into the body of a program could change the multithread behavior of that program. In contrast, adding a FOR loop, or making any other modifications outside a parallel region, to OpenMP-C is identical in effect to making the same change to the serial program providing no directives are involved.

IV. Conclusion

OpenMP-C and UPC are two contemporary based-languages programming models that are still in development. Both are based on ANSI C as a base-language. 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 crossroads. It was 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, OpenMP can be extended by HPF-like compiler directives for supporting data-distribution. A few vendors have already extended their compilers by such mechanisms [13], [14], [15]. This is the easy way. On the other hand, OpenMP can be integrated with MPI for creating a coherent hybrid model [23]. This is the hard way, but a much more promising one.

UPC is a very promising language-based programming model. It was designed a priori for NUMA architectures, introduces a small set of powerful constructs, adds advanced memory consistency options, and there are plans to add libraries for collective communications and parallel I/O.

OpenMP-C and UPC are another phase in the evolution process of the parallel programming models and languages. Unfortunately, they do not present a new paradigm that can be considered as a revolutionary change.

References

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

[2] T. El-Ghazawi, F. Cantonnet, P. Saha, R. Thakur, R. Ross, D. Bonachea, *UPC-IO: A Parallel I/O API for UPC.* V1.0 pre9, May 1 2003.

[3] E. Wiebel, D. Greenberg, S. Seidel, *UPC Collective Operations Specifi - cations.* pre4V1.0, April 2, 2003.

[4] K. Yelick, D. Bonachea, J. Duell and C. Wallace, *New UPC Memory Model Defi nition.* draft September 29, 2003.

[5] S. V. Adve and K. Gharachorloo, *Shared Memory Consistency Models: A Tutorial.* IEEE Computer, Vol. 29, No. 12, pp. 66-76, December 1996.

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

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

[8] A. Marowka, Z. Liu and B. Chapman, *OpenMP-Oriented Applications for Distributed Shared Memory Architectures.* Concurrency & Computation: Practice & Experience journal, Issue 16, Number 3, March, 2004.

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

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

[11] J. M. May, *Parallel I/O for High Performance Computing.* Morgan Kaufmann, October 2000.

[12] *MPI-2: Extensions to the Message-Passing Interface.* Message Passing Interface Forum, July 18, 1997.

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

[14] J. Bircsak, P. Craig, R. Crowell, Z. Cvetanovic, J.Harris, C, A. Nelson and C. D. Offner, *Extending OpenMP for NUMA machines.* Scientifi c Programming, Vol. 8, No. 3, 2000.

[15] J. Merlin, D. Miles, 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.

[16] D. B. Skillcorn and D. Talia, *Models and Languages for Parallel Computation.* ACM Computing Surveys, Vol. 30, No. 2, June 1998.

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

[18] D. E. Culler, A. Dusseau, S. C. Goldstein, A. Krishnamurthy, S. Lumetta, T. V. Eicken, and Y. Yelick, *Introduction to Split-C.* University of California, Berkeley, 1993.

[19] W. W. Carlson and J. M. Draper, *Distributed Data Access in AC.* Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP), Santa Barbara, CA, July 19-21, 1995, pp.39-47.

[20] B. Eugene and K. Warren, *Development and Evaluation of an Effi cient Parallel Programming Methodology, Spanning Uniprocessor, Symmetric Shared memory Multi-processor, and Distributed-memory Massively Parallel Architectures.* Poster SuperComputing'95, San Diego, CA., 1995.

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

[22] Z. Xu and K. Hwang, *Coherent Parallel Programming in C//.* Proceeding of International Conference on Advances in Parallel and Distributed Computing, IEEE Computer Society Press. pp. 116-122, March 1997.

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

[24] Ami Marowak. *Advanced Language-Based Parallel Programming Models.* Technical Report, Hebrew University of Jerusalem, January, 2004.