A Parallel Programming Model for Ada

Hazem Ali & Luís Miguel Pinho

ABSTRACT

Over the last three decades, computer architects have been able to achieve an increase in performance for single processors by, e.g., increasing clock speed, introducing cache memories and using instruction level parallelism. However, because of power consumption and heat dissipation constraints, this trend is going to cease. In recent times, hardware engineers have instead moved to new chip architectures with multiple processor cores on a single chip. With multi-core processors, applications can complete more total work than with one core alone.

To take advantage of multi-core processors, parallel programming models are proposed as promising solutions for more effectively using multi-core processors. This paper discusses some of the existent models and frameworks for parallel programming, leading to outline a draft parallel programming model for Ada.

Keywords

Ada, Many-core systems, Parallel programming, Lightweight threads model

1. INTRODUCTION

Multi-core architectures, integrating several processors on a single chip, are quickly becoming widespread, even in small embedded systems. This cheaply available computational power makes parallel programming more than ever a concern for software developers, since the sequential programming model does not scale well for such multi-core systems [1].

The current trend to use multi-core platforms will thus not provide improvements on the performance of software, and may even impact its reliability, if programming environments are not also evolved to account for the new paradigm of naturally parallel hardware [2]. It is recognized that new (or old) parallel programming models are needed to take advantage of (large) parallel platforms, that data structures, algorithms and code generation tools must be made aware of the underlying architecture changes, and that programming should be independent of the number of processors, to shield from likely hardware evolution. The problem is exacerbated for platforms with larger number of cores (usually noted as many-core).

It is not a surprise that many research projects and commercial frameworks have been either proposing new or re-using old models, specifically targeting the potentially large-scale parallelism found in multi-cores. Frameworks such as Cilk [3], Intel's Threading Building Blocks [4], Java Fork/Join [5], OpenMP [6], Microsoft's Task Parallel Library [7], StackThreads/MP [8] or Paraffin [9] provide a model where the programmer divides the application in numerous potentially parallel computing units[1], which are then dynamically assigned to worker threads in the cores by the frameworks' runtime, considering the actual load in the system. To deal with the load balancing of these parallel units in the worker threads (and thus in the cores), the work-stealing algorithm [10] is currently one of the most widely-used, although it may not perform better in all cases [11].

Work-stealing has the advantage of reducing task contention, due to the support for double-ended queues, with LIFO behaviour when worker-threads process their own-generated units, and FIFO behaviour when threads steal from other threads queues. Another advantage is that as soon as one unit migrates (is stolen) to a new core, all units generated by it are placed in this core queue, thus decreasing the need for further stealing. Finally, as threads execute units in LIFO order, they maximize the probability of data still being in the cache. Contrarily, stealing is performed in the other end of the queue, targeting older units, minimizing the probability of the data being in the cache (of the old wrong core).

This parallel programming model based on potentially parallel computation units also provides higher-abstraction advantages. For instance, the programmer can focus on writing functional code, only explicitly specifying potentially parallel operations, leaving to the underlying framework the dynamic mapping of units to threads. This separation of concerns leads to more reliable code, and more optimized runtimes. Also, it improves programmer productivity.

---

[1] These units may be called lightweight threads, tasks, pJobs, depending on the context and framework. For consistency and simplicity in this paper (and to not clash with the Ada notion of task) we use potentially parallel computation units.

Also, the composability of several different components, all using this model is easily performed, as these components only create the units, being all of them scheduled by the underlying runtime.

Obviously, this approach introduces some overhead, both on data structures needed to manage the computation units and their mapping to cores, but also on the migration of units (and eventual impact in caches). However, it is important to note that stealing only takes place if a particular core is idle. Therefore, the overhead is not significant, particularly as the number of cores increases larger than the number of concurrent activities (threads) of an application.

Considering the above, it is important to assess the use of this model in Ada. Although multi-core programming support will be available in the forthcoming revision of the language (Ada 2012) [12], the programming model of Ada will still be based on the definition of heavier task units, as it is targeted to environments where the number of cores is far less than the number of application tasks.

The structure of the paper is as follows. The next section reviews some previous attempts to define parallelism in Ada. Section 3 then provides an overview of several current parallel programming models using potentially parallel computation units. Afterwards, Section 4 provides and discusses the proposed Ada model. Finally, some conclusions are presented.

2. PARALLEL PROGRAMMING AND ADA

It is important to note that parallel programming approaches in Ada were considered several years ago (e.g. [13,14,15]) [2]. The work in [13] introduces a `parallel` keyword, for `for loops`, allowing a specific compiler to optimize loop iterations, targeted to a multiprocessor platform. The work in [14] is similar, as it also targets the optimization of parallel loops; furthermore, the authors state that Ada tasks are not the appropriate unit of parallelization thus proposing a concept of minitasks which can be optimized by compilers and runtimes aware of this model**.** It is interesting to note that [14] already puts forward some of the ideas that are currently being (again) discussed concerning the use of tasks/threads for manycores, in particular the excessive context and initiation overhead which is required to manage tasks in parallel machines. The solution proposed is in line with the current move to provide more efficient parallel units.

Contrarily, in [15] the author proposed a model for integrating parallel dataflow programming with the Ada tasking model, proposing two extension keywords to standard Ada: `parallel` and `single`. The `parallel` keyword is used for declaring explicitly that a set (block) of statements or a `for loop` will be executed in parallel. It transforms these into a sequence of task declarations with a separate task representing each statement or iteration respectively. On the other hand, the `single` keyword is used for declaring single-assignment types (also known as immutables) for exchanging data and synchronization between parallel blocks.

After that, the research evolved in [16] by defining a new programming language called Declarative Ada where parallelism is implicit. Declarative Ada is a programming language based on a Pascal-like subset of Ada. The difference between Declarative Ada and the previously proposed extension is that all variables are considered as single-assignment. This allows implicit parallel execution of programs with synchronization through run-time dataflow.

Although in Declarative Ada all statements can be executed in parallel, we believe that it will not be up to the expectations from the point of view of performance. This comes from the fact that all parallel executions are mapped to Ada tasks, thus creating higher overhead during execution that will eliminate any gained speedup from parallel execution [14] Moreover, this very fine grain parallelism, which comes from the fact that each statement is a parallel block, may lead to a mass synchronization overhead between different parallel blocks. This means degraded overall performance.

The previous discussion demonstrate the growing need for constructs or methods that define true parallelism in Ada away from the well known task model, which should be used for concurrency. As Robert Harper states: *"The first thing to understand is parallelism has nothing to do with concurrency"* [17]. Parallelism is concerned with efficiency of programs operating in parallel platforms, and where the output results are deterministic.

Concurrency, on the other hand, refers to the nondeterministic execution of applications where expected and unexpected events must me managed. Such situations are not concerned with efficiency and performance as much as getting the system to operate correctly and under control. Concurrent systems can be implemented in parallel platforms, but can also be in sequential ones.

The task model of Ada is undoubtedly suitable for concurrent systems, where each task maps to an application concurrent activity, that can abort, suspend or resume its execution according to the system requirements. Orthogonal to this model, parallel constructs proposed for Ada should adopt a lightweight model, i.e., lightweight computation units used as building blocks, mapped to a pool of worker (system) tasks/threads with a special purpose scheduling discipline.

This is the model of a more recent approach to provide parallel programming support in Ada [9], with support to potential parallel computation units, with work-sharing, work-seeking and work-stealing functionality through an external library. Our proposal, although in the same context, is different, proposing that Ada revisits its parallel programming model, intending to explore a language based approach that hides from the programmer the concrete mapping of the application into the parallel platform whilst allowing him/her to define the potentially parallel blocks.

3. PARALLEL PROGRAMMING MODELS

In order to propose a parallel programming model for Ada, it is important to analyze currently used approaches for the design of parallel programs. In the brief analysis in this paper, the common example of the Fibonacci function will be used to present the most relevant features of each approach. Note that although the iterative version of the Fibonacci function is more efficient than the recursive version, the recursive version may be a better solution in parallel platforms, when a large percentage of the time processors are idle. Furthermore, it is a good example of a simply parallelizable function.

[2] There is indeed a general trend to look back to the past in all areas of computers as parallel platforms become widespread.

The first example we present is how to develop the function by using Ada tasks as the unit of parallelization. Two approaches are shown. The first approach (Listing 1) creates one task per function execution, in order to potentially (and naively) try to maximize parallelism. It is clear that the overhead of task creation will impair the advantages of the parallelization.

```
task type Fib (N: Natural) is
   entry Result(R: out Natural);
end Fib;
task body Fib is
   Res, N1, N2: Natural;
   Fib_Acc_N1, Fib_Acc_N2 : access Fib;
begin
   if Value < 2 then
      Res := Value;
   else
      Fib_Acc_N1 := new Fib(N - 1);
      Fib_Acc_N2 := new Fib(N - 2);
      Fib_Acc_N1.Result(N1);
      Fib_Acc_N2.Result(N2);
      Res := N1 + N2;
   end if;
   accept Result(R: out Natural) do
      R := Res;
   end Result;
end Fib;
```

Listing 1 – Fibonacci Example #1

The second (naïve) example (Listing 2) uses the number of cores information to divide the problem, thus creating one task per core. The program gets more complex as it is necessary to consider the relation between the number of cores and the size of the problem, and it is necessary to keep track of the available tasks. If a task is not available, then the calculation will be done with a sequential version of the algorithm.

Note that the size of the problem will not be the same on each core, thus there will be idle cores while other will be overloaded. If the programmer attempts to do a load balancing solution, he/she will end up with redeveloping a complete work-sharing or work-stealing algorithm.

```
function Seq_Fib (N : in Natural)
      return Natural is
begin
   if Value < 2 then
      return N;
   return Seq_Fib (N - 1) +
          Seq_Fib (N - 2);
end Seq_Fib;
```

```
task type Fib is
   entry Value (N: in Natural);
   entry Result(R: out Natural);
end Fib;

protected Task_Pool is
   procedure Try_Get(T: out access Fib;
                     Val: Natural);
private
   Workers: array (1..CPU_Count) of Fib;
end Task_Pool;

task body Fib is
  Val, N1, N2: Natural;
  Fib_Acc_N1, Fib_Acc_N2 : access Fib;
begin
   loop
      accept Value (N: in Natural) do
         Val := N;
      end Value;
      if Value < 2 then
          Res := Value;
      else
         -- try parallel of both branches
         Task_Pool.Try_Get(Fib_Acc_N1,
                           Val - 1);
         Task_Pool.Try_Get(Fib_Acc_N2,
                           Val - 2);
         if Fib_Acc_N1 /= null then
            Fib_Acc_N1.Result(N1);
         else
            N1 := Seq_Fib (Val - 1);
         end if;
         if Fib_Acc_N2 /= null then
            Fib_Acc_N2.Result(N2);
         else
            N2 := Seq_Fib (Val - 2);
         end if;
      end if;
      accept Result(R: out Natural) do
         R := N1 + N2;
      end Result;
```

```
   end loop;
end Fib;
protected Task_Pool is
   procedure Try_Get(T: out access Fib;
                     Val: Natural) is
   begin
      for I in 1..CPU_Count loop
         select
            Workers(I).Value(Val);
            T := Workers(I)'access;
            exit;
         else
            null;
         end select;
         T := null;
      end loop;
   end Try_Get;
end Task_Pool;
```

Listing 2 – Fibonacci Example #2

The next sub sections present three different frameworks, all based on creating potentially parallel computation units, following the technique described in the Introduction. These frameworks are different, as they follow different approaches: a library based one; a pre-processor based one, and a language based one.

3.1 Library-based approaches

The example of a library-based approach is provided using the Intel's Threading Building Blocks [4], which is a library, implemented using the C++ Standard Template Library, and that provides the required classes to design and manage the parallel computation units, called tasks.

```
class FibTask: public task {
public:
   const long n; long* const sum;
   FibTask( long n_, long* sum_)
       : n(n_), sum(sum_) {}

   task* execute() {
      if(n < 2) {
         *sum = n;
      } else {
         long x, y;
         FibTask& a =
            *new(allocate_child())
             FibTask(n-1,&x);
         FibTask& b =
            *new(allocate_child())
             FibTask(n-2,&y);
         set_ref_count(3);
         spawn( b );
         spawn_and_wait_for_all( a );
         *sum = x+y;
      }
      return NULL;
   }
}
```

Listing 3 – TBB Example

This code in Listing 3 uses an object of the class `FibTask`, which inherits from the special class task, to do the actual work. It starts by creating two new task objects to compute `n-1` and `n-2`, and then spawns these tasks (the last one is a spawn and wait which will cause the main task to wait for the two children). Note that these tasks are not similar to Ada tasks, but lightweight computation units which will be executed by runtime worker threads.

The work of [9] proposes a similar approach for Ada, where a library of generics is proposed to allow for common parallel patterns in Ada programs.

3.2 Pre-processer based approaches

The next example presents the same function using the OpenMP [6] specification, a pre-processor based approach. OpenMP is a specification produced by an industry consortium, based on directives that allow the pre-processor or the compiler to automatically inject the code required to execute the program on top of the parallel runtime.

```
int fib(int n)
{
   if (n < 2) return n;
   int x, y;

   #pragma omp task shared(x)
   x = fib(n - 1);

   #pragma omp task shared(y)
   y = fib(n - 2);

   #pragma omp taskwait
   return x+y;
}
```

Listing 4 – OpenMP Example

The example in Listing 4 provides two of these directives. The first, which is used before both recursive calls to the `fib` function

notes the pre-processor that the following block of code can be executed in parallel. Thus, `fib(n - 1)` and `fib(n - 2)` can be executed by two parallel threads. The final directive causes the main task to wait for the end of the tasks that it has created. The `shared(x)` information informs the compiler that variable `x` will be accessed by different threads, therefore exclusion algorithms should be used.

The work in [9] also provides a brief proposal how Ada could have a similar approach, by using a set of `pragmas`, with the compiler converting the code to use the generic libraries, thus hiding more complex programming.

3.3 Language based approaches

The final example presents the language based approach of Cilk Plus [18], an evolution of Cilk [3] / Cilk++ [19], that provides a very simple and small set of linguistic extensions to C++ to support parallel applications, on top of libraries and runtimes providing work-stealing capabilities. Because it is a small extension, parallelizing existent code is a very easy task.

```
int fib(int n)
{
   if (n < 2) return n;
   int x, y;

   x = cilk_spawn fib(n-1);
   y = fib(n-2);

   cilk_sync;

   return x+y;
}
```

Listing 5 – Cilk Example

The `cilk_spawn` keyword in Listing 5 performs the same functionality of the `omp task` directive in the previous sub section, noting the compiler that `fib(n - 1)` can execute in parallel. `cilk_sync` is equivalent to `omp taskwait`.

4. A PROPOSAL FOR ADA

In this proposal for Ada, the goal is to maintain the structure of Ada programs, but allowing the programmer to specify code which is potentially parallel, which then the runtime can dynamically during runtime either execute sequentially, or parallelize. The followed approach is a language-based one, as the authors consider it to be more appropriate to the Ada philosophy of supporting concurrency directly at the language level.

In this preliminary work, three constructs are proposed:

- Parallelizable blocks
- Parallelizable functions
- Parallelizable for loops

The proposal introduces two new keywords: `parallel`, which specifies potentially parallel operations, and `future`, which specifies values which are calculated asynchronously.

4.1 Parallelizable blocks

Ada's block construct is a natural candidate for declaring potentially parallel code, as it encloses a sequence of statements in a single statement that can be placed anywhere in an Ada program. Furthermore, the variables in the declarative part can be created in the actual core of execution, similar to private variables in OpenMP, allowing for a better utilization of the local caches. Listing 6 provides the structure of the parallel block, which is identified with the `parallel` keyword.

```
-- not legal Ada
declare
   Local_var : ...;
   Local_copy : ... := Global_var;
parallel begin
   -- ...
end parallel;
```

Listing 6 – Proposal for a parallel block

Nevertheless, it is important that these "parallel blocks" do not update global variables. This can be allowed, but programmers must understand that the behaviour and performance may be the same as variables being assessed by different tasks in different threads (and different processors), so protected objects should be used. Read only variables may be copied by the programmer in the declarative part, or may be implicitly copied by the compiler.

Since these parallel blocks execute asynchronously with the following code, it is important to determine how the results of the block are used, and it must be possible for the main program to wait for them to be available. A potential solution is to use futures [20], variables which are a placeholder for a future result. Synchronization is only required when the value is actually used. Obviously, for a future to be used, its scope must be enclosing of the parallel block (Listing 7).

```
 -- not legal Ada
Future_V: future ...;
begin
   -- ...
   declare
      Local_var : ...;
   parallel begin
      -- ...
      -- code that computes Future_V
   end parallel;

   -- asynchronous execution
   Do_Something_Else;
```

```
X := Future_V; -- The result of the
               -- computation is required
               -- Program will wait for
               -- end of parallel block
```

Listing 7 –parallel block example

4.2 Parallelizable functions

A second construct presented in this paper is a simple way for the programmer to specify that a function can execute in parallel. Therefore, when a call to the function is performed, the underlying runtime can decide to parallelize the call, if there are enough available cores.

For instance, the example in Listing 8 maps the parallel Fibonacci function, as presented in the previous section, with the parallel function construct.

```
-- not legal Ada
parallel function Fibonacci (
                      Value : in Natural)
      return future Natural is
   n1, n2: Natural;
begin
   if Value < 2 then
      return Value;
   n1 := Fibonacci (Value – 1);
   n2 := Fibonacci (Value – 2);
   return n1 + n2;
end parallel Fibonacci;
```

Listing 8 – Proposal for a parallel function

Note the use of the `parallel` keyword in the function signature [3], and the specification of the return value as a `future`, as the function may be executed asynchronously, thus the calling code must wait for its completion when the return value is required (Listing 9).

```
 -- not legal Ada
Fib_Res: future Natural;
begin
Fib_Res := Fibonacci (10);
... -- code in parallel to the
    -- fibonacci function call
X := Fib_Res; -- The result of the
              -- computation is required
              -- Program will wait for
              -- end of parallel funtion
```

Listing 7 –parallel block example

4.3 Parallelizable for loops

The last construct that can be paralyzed is the `for loop`. Note that in this case, it is not mandatory that each iteration of the loop is executed in parallel. Efficient runtimes may partition the data set into blocks, and assign each block to a potentially parallel unit (an approach similar to TPL's `Parallel.For` [7]). Listing 10 provides an example of incrementing all elements of an array.

```
-- not legal Ada
for I in Buffer'Range    parallel loop
   Buffer(I) := Buffer(I) + 1;
end parallel loop;
```

Listing 10 – Proposal for a parallel loop

A more complex approach is required if the for loop is performing an aggregation (loop iterations are not independent). For example, Listing 11 provides and example of a sequential `for loop`, performing the sum of an array. Obviously, this cannot be parallelized with the same approach as in Listing 9.

```
Sum := 0;
for I in Buffer'Range loop
   Sum := Sum + Buffer(I);
end loop;
```

Listing 11 – Sequential Sum loop

The parallelization of this type of loops is only advantageous if there is the capability to do partial sums for array blocks and then performing an aggregate sum in the end. For that, the change to the structure of the `for loop` needs to be more complex.

Note that it is not possible to delegate to the programmer the definition of blocks and partial arrays, as it may be the underlying runtime that determines during the execution the number and size of blocks. But it is necessary for the program to be able to reason in terms of block ranges and partial results. A solution (Listing 12) could be to create specific attributes to arrays, which could be used to know the range of the current block (`Partial_First .. Partial_Last`), and to allow variables to have a `Partial` attribute referring to a local copy in each parallel block. After the end of the `parallel for`, these local copies could be available in an array, accessible by the `Partial_Array` attribute.

```
-- not legal Ada
Sum := 0;
for I in Buffer'Range parallel loop
   for J in Buffer'Partial_First ..
            Buffer'Partial_Last loop
      Sum'Partial := Sum'Partial +
                     Buffer(J);
```

[3] Previously we had considered the `parallel` keyword to be placed after `is`, just before the declaration block, or before `begin`, as in parallel blocks. However, it was later decided to place it in the beginning to be able to more easily define potential parallel functions also at declaration. Aspects were also considered, but using `function X with Parallel` does not convey the correct idea.

```
   end loop;
end parallel loop;


for I in Sum'Partial_Array loop
     Sum := Sum + Buffer'Partial(I);
end loop;
```

Listing 12 – Proposal for a parallel loop with aggregation

Note that the code after the `parallel for loop` waits for all parallel iterations to terminate, before being able to execute.

4.4 Discussion

The work presented in this paper is still preliminary as there are several issues which need to be considered. For instance, the interaction of parallel computations and the exception model of Ada is complex, as parallel computation may be performed in worker threads, thus in a context which is not of the enclosing task. Also exiting from blocks or loops in recursive parallel computations must take into account the potential need to abort other computations being executed in other threads.

Nevertheless, the three proposed constructs for parallelism mentioned in the previous sections (parallelizable blocks, functions and for-loops), can be the basis for creating a parallel programming model in Ada. In addition, the future keyword can play an important role in synchronization, acting as a join function for different parallel computations that need to meet at some point of execution.

Another area of importance to the design of parallel Ada programs is the data-sharing model. It is interesting to note that asynchronous message passing between parallel code is more and more considered to be an option for highly parallel programs, instead of data-sharing. Another area which needs to be considered is the use of non-blocking data structures or software transactions instead of lock-based data sharing.

Finally, the incorporation in the Ada runtime model of the support to the parallelizable computational units is also of paramount importance. It is thus clear that the definition of the semantics of this model is indeed a challenging (but potentially parallel) task, considering the interaction with all Ada features.

Nevertheless, the provided examples are sufficient to outline how the model could be implemented within the Ada language model (the code still looks Ada), and it is a starting point to foster a discussion on this issue. It is the opinion of the authors that the Ada community must start considering that for the foreseen future platforms (tens or hundreds of cores), the available task model may not scale. The area of programming models for parallel computing is (once again) with immense activity, and Ada should define its model. Both the work presented in this paper, and the generics/pragma implementation of [9] are two directions that can be considered.

5. CONCLUSION

The current trend to increase processing power by manufacturing chips including multiple processor cores has popularised the ability to execute concurrent software in parallel. This tendency for even larger number of processor cores will further impact the way systems are developed, as software performance must rely on efficient techniques to design and execute concurrent software in parallel.

This paper discusses some existent approaches to parallel programming using the lightweight thread model, where the programmer specifies a set of potentially parallel computation units, which are then dynamically mapped by the runtime to a set of worker threads, and proposed a draft of how the Ada language could be augmented to support such model.

6. ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their valuable comments to improve the paper.

This work was supported by the VIPCORE project, ref. FCOMP-01-0124-FEDER-015006, funded by FEDER funds through COMPETE (POFC - Operational Programme 'Thematic Factors of Competitiveness') and by National Funds (PT) through FCT - Portuguese Foundation for Science and Technology, and the ARTISTDESIGN – Network of Excellence on Embedded Systems Design, grant ref. ICT-FP7-214373.

7. REFERENCES

[1] H. Sutter and J. Larus, "Software and the concurrency revolution," Queue, vol. 3, pp. 54–62, September 2005.

[2] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The landscape of parallel computing research: A view from berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, Dec 2006.

[3] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the cilk-5 multithreaded language. SIGPLAN Not., 33:212-223, May 1998

[4] Intel. Thread Building Blocks, http://threadingbuildingblocks.org/. Last access September 2011.

[5] D. Lea. A java fork/join framework. In Proceedings of the ACM 2000 conference on Java Grande, JAVA '00, pages 36-43, New York, NY, USA, 2000. ACM.

[6] A. Marowka. Parallel computing on any desktop. Commun. ACM, 50:74-78, September 2007.

[7] Microsoft. Task parallel library, http://msdn.microsoft.com/en-us/library/dd460717.aspx. Last access September 2011.

[8] K. Taura, K. Tabata, and A. Yonezawa. Stackthreads/mp: integrating futures into calling standards. ACM SIGPLAN Notices, 34(8):60–71, 1999.

[9] B. Moore, "Parallelism generics for Ada 2005 and beyond", SIGAda'10 Proceedings of the ACM SIGAda annual conference, October 2010.

[10] R. D. Blumofe and C. E. Leiserson. Scheduling multithreaded computations by work stealing. J. ACM, 46:720-748, September 1999.

[11] Moore, B., "A comparison of work-sharing, work-seeking, and work-stealing parallelism strategies using Paraffin with Ada 2005", Ada User Journal, Vol 32, N. 1, March 2011.

[12] A. Burns and A. J. Wellings, “Dispatching Domains for Multiprocessor Platforms and their Representation in Ada,” 15th International Conference on Reliable Software Technologies - Ada-Europe 2010, Valencia, Spain, June 14-18, 2010.

[13] H. G. Mayer, S. Jahnichen, “The data-parallel Ada run-time system, simulation and empirical results”, Proceedings of Seventh International Parallel Processing Symposium, Aprl 1993, Newport, CA , USA , pp. 621 – 627

[14] M. Hind , E. Schonberg, “Efficient Loop-Level Parallelism in Ada”, TriAda 91, October 1991

[15] J. Thornley, “Integrating parallel dataflow programming with the Ada tasking model”. In *Proceedings of the conference on TRI-Ada '94* (TRI-Ada '94), Charles B. Engle, Jr. (Ed.). ACM, New York, NY, USA, 417-428, 1994. DOI=10.1145/197694.197742 http://doi.acm.org/10.1145/197694.197742

[16] J. Thornley, “Declarative Ada: parallel dataflow programming in a familiar context". In *Proceedings of the 1995 ACM 23rd annual conference on Computer science* (CSC '95), C. Jinshong Hwang and Betty W. Hwang (Eds.). ACM, New York, NY, USA, 73-80, 1995. DOI=10.1145/259526.259540 http://doi.acm.org/10.1145/259526.259540

[17] R. Harper, “Parallelism is not concurrency”, Ropert Harper Blog, http://existentialtype.wordpress.com/ 2011/03/17/parallelism-is-not-concurrency/, Last access September 2011.

[18] Intel, Cilk Plus, http://software.intel.com/en-us/articles/intel-cilk-plus/, Last access September 2011

[19] C. Leiserson, “The Cilk++ concurrency platform”, Proceedings of the 46th Annual Design Automation Conference , ACM New York, USA, 2009.

[20] H. Baker, C. Hewitt, "The Incremental Garbage Collection of Processes". Proceedings of the Symposium on Artificial Intelligence Programming Languages, SIGPLAN Notices 12, August 1977.