Parallel Ada - A Requirement for Ada 2020

A. Burns
Department of Computer Science,
University of York, UK

Abstract

Much of the focus with multi-core hardware has been on the mapping of tasks to cores. It is important that languages can support both static and dynamic forms of this mapping. However, as the number of cores increase and platforms become more heterogeneous it becomes necessary to identify and support parallel execution within tasks. Various forms of 'parallel' statement have been discussed in the literature. Here we argue for the need for simple changes to the language that can go a long way towards exploiting fine grain parallelism.

1 Introduction

During a Panel on language support for multi-core platforms at the Ada Europe conference in 2011 an informal classification [3] was made between platforms with

- one core, or
- a few cores, or
- many cores.

The point was made that most languages were defined for single CPUs. A 'few cores' was defined to be a number less than the natural number of tasks found in embedded software. For such platforms the 'task' is the most straightforward means of exploiting the platform's parallelism. Tasks are allocated (either statically or dynamically) to cores, and at run-time there is usually more executable tasks than cores and hence an adequate exploitation of the hardware is being made.

With 'many cores' however there are more cores than tasks (by definition) and hence the parallelism can only be exploited if parallelism within tasks is identified and utilised. Unfortunately, the considerable effort that has been exerted in trying to automatically extract parallelism from sequential code has, in large part, failed. It therefore seems that programmers must directly address the issue, and programming languages must support adequately fine grain parallelism. This paper considers the form of support this might take for Ada as this languages moves towards its 2020 definition.

2 Fine Grain Concurrency

Few programming languages support fine grain concurrency. Most, if they support concurrency at all, do so by a task, thread or process notion. A program consists of a relatively small number of relatively large/long sequential tasks. There are however some examples of sub-task concurrency. For example, OCCAM [2, 6, 10] defined parallel composition to be equivalent (syntactically) to sequential composition:

```
PAR
  A := 1
  B := 2

SEQ
  A := 1
  B := A + 2
```

Only when there is an explicit need to constrain the code is the `SEQ` statement used. Of course `PAR` does not mean parallel execution, it just indicates that parallel execution is allowed. A better label would have been `CON` for concurrency.

Whereas extracting parallel execution from sequential code is problematic, extracting parallel execution from fine grain concurrency is relatively straightforward. Both the compiler and the run-time support software have a role. The compiler can convert too fine grain code to sensibly sized parallel 'chunks'. And the run-time can dynamically decide on how to partition code. With the above example, the assignments to `A` and `B` really should be sequential and the compiler can accomplish this. But in

```
PAR
  F1
  F2
```

where `F1` and `F2` were arbitrary function calls then the compiler would retain in the executable code the potential for parallel execution and the run-time would indeed execute the functions in parallel if there were cores available. Of course if `F1` and `F2` turn out to be executed in sequence then there should be no (or very little) extra overhead for the programmer using `PAR` rather than `SEQ`. For `PAR` and `SEQ` to be equivalent then `F1` and `F2` cannot share any program entities.

For programming languages with pure functions (i.e. no side effects) then the parallel execution of `F1` and `F2` is easily guaranteed to be correct. It is perhaps an issue for Ada 2020 whether pure functions should be supported more explicitly.

3 Requirements

Exploiting highly parallel hardware will require run-time entities that are simpler (more light-weight) than processes or even threads. A number of terms are used in the OS literature for such entities, including, micro-threads and pico-threads. Typical constraints/requirements for these entities are:

- They do not block.
- They can potentially start execution as soon as they are created.
- They have no internal state (they are more event like).
- They terminate (and cease to exist) as soon as they complete the execution of their code.
- They are not named and are not therefore directly addressed in the program.

Clearly Ada tasks are much more heavy-weight. Support for parallelism must come at the statement level (from within tasks). As any particular task will only execute in a single (Ada) partition, the following is focussed on partitions with many-cores.

For statement-level parallelism there are really only two main requirements, and these were well supported in OCCAM. It must be possible to state that a block of code statements can be executed in parallel (**`parbegin/parend`** perhaps). Also it must be possible to say that each iteration of a loop can execute in parallel (perhaps indicated by **`parloop/end parloop`**).

Both of these constructs have the form: create parallelism, execute immediately, wait for completion of all parallel components, continue. So really a simple fork and join construct but with a higher branching factor.

It might be possible to get new key words into Ada 2020, or if not then perhaps a syntactical form using the new aspect feature could be utilised. In the code examples used below the simple addition of '#' to either **`loop`** or **`begin`** is used to identify potential parallelism. This use of a simple single character is used to show that little does actually have to change to existing code.

There have of course been a number of papers that have advocated the introduction of some sort of 'parallel' key words into Ada [1, 4, 5, 7–9]. The motivation for this paper is to bring the issue forward for discussion at IRTAW16, and to illustrate that in fact the requirements on the language are not too excessive.

The above has focused on control parallelism, there is also the requirement to support data parallelism. Concurrent parts of a program might be assessing different parts of a large data structure (for example an array), but if the underling hardware serialises access to the memory via either the bus or the memory itself, then much of the potential advantage available from the parallel processing hardware will be lost due to non-parallel memory. This issue must be addressed by any proposed change to the language.

4 Examples

The above additions to Ada are quite simple. Are they sufficient to write programs that can exploit highly parallel hardware? The answer to this is probably yes, but for Ada the question should be: are they sufficient to write correct programs that can easily exploit highly parallel hardware?

As mentioned above pure functions would help structure parallel code. Otherwise the programmer must ensure that parallel entities cannot interfere with each other (e.g. use shared variable). As our micro-threads cannot block then they cannot make use of protected objects (POs). POs are for tasks (which would still exist), for parallel constructs it is necessary to use different programming patterns. Here we give one such example which might help to define what Ada 2020 needs to support (or need not support).

Lets start with code that is clearly easy to parallelise – a loop that adds one to each element of an array. So rather than

```
for I in 1..100_000 loop  -- i.e. a large N
  Store(I) := Store(I) + 1;
end loop;
```

the programmer could put

```
for I in 1..100_000 loop#  the # notation implies a parallel loop
  Store(I) := Store(I) + 1;
end loop#;
```

A compiler might be able to put this into sequential chunks of say 100 iterations; if not the programmer may need to:

```
for I in 0..999 loop#
  for J in I*100+1 .. (I+1)*100 loop
    Store(J) := Store(J) + 1;
  end loop;
end loop#;
```

Ideally the code would not need to be explicit about the size of chunk, rather this information could be given directly to the compiler.

A more general problem is to sum the elements in this array:

```
Sum := 0;
for I in 1..100_000 loop
  Sum := Sum + Store(I);
end loop;
```

Clearly it would not be sensible to simple replace the **loop** by a **loop**#. But again the programmer could make the level of parallelism explicit by summing up parts of the array in parallel to obtain subtotals that are then added together:

```
Sum := 0;
for I in 0..999 loop#
  Summ(I) := 0;
  for J in I*100+1..(I+1)*100 loop
    Summ(I) := Summ(I) + Store(J);
  end loop;
end loop#;
for I in 0..999 loop
  Sum := Sum + Summ(I);
end loop;
```

An alternative means of achieving this parallelism is to use recursion rather than an explicit loop. Now let `Summer` be a function, with two parameters:

```
function Summer(A, Z : integer) return integer is
    temp, temp2 : integer := 0;
begin
  if Z-A <= 10 then  -- unit of sequential execution
    for I in A..Z loop
      temp := temp + Store(I);
    end loop;
    return temp;
  else
    begin#
      temp := Summer(A,A+(Z-A)/2);
      temp2 := Summer(A+(Z-A)/2+1,Z);
    end#;
    return temp + temp2;
  end if;
end Summer;

-- with the usage:
Summer(1,100_000);
```

Here the programmer has decided that the minimum chuck of code is 10 summations. At run-time a dynamic decision can be made as to how much parallel execution is exploited. At each **begin**#, if there is an available core, then the code will genuinely run in parallel. For example, if there are 4 cores then the first call of Summer(1,100_000) will result in parallel calls to Summer(1,50_000) and Summer(50_001,100_000). The first of these may result in parallel calls to Summer(1,25_000) and Summer(25_001,50_000); and the second may result in the parallel execution of Summer(50_001,75_000) and Summer(75_001,100_000). The subsequent recursive calls will result in sequential execution as there are no further available cores to exploit.

5 Conclusions

The need to write programs that can easily and efficiently exploit highly parallel hardware is an increasingly significant one. For Ada this means that the programmer should be able to designate code that can be executed in parallel . And this code will need to be at a granularity well below that of the task. Sufficient implementation freedom must be available to allow the compiler and the run-time support system to jointly target the capabilities of the hardware - whether this is a multicore SMP or heterogeneous hardware directly fabricated to meet the specific needs of the program.

Although an important challenge, significant support for parallelism can be obtained, in a language like Ada, by small changes to the language. In particular, including the ability to designate blocks of code to be executed in parallel and loops to have all iterations executed in parallel is a straightforward but nevertheless important facility.

Functional programming languages have been shown to more easily exploit parallel hardware because they contain pure functions (functions without side effects). Ada 2020 should therefore look at ways of designate such functions.

References

[1] H. Ali and M. Pinho. A parallel programming model for Ada. In *SIGAda*, pages 19–26, 2011.

[2] A. Burns. *Programming in occam 2*. Addison Wesley, 1988.

[3] A. Burns. Programming languages for real-time applications executing on parallel hardware. In Alexander Romanovsky and Tullio Vardanega, editors, *Reliable Software Technologies - Ada-Europe 2011*, volume 6652 of *Lecture Notes in Computer Science*, pages 193–195. Springer Berlin Heidelberg, 2011.

[4] R. Chun, R. Lichota, B. Perry, and N. Sabha. Synthesis of parallel ada code from a knowledge base of rules. In *Parallel and Distributed Processing, 1991. Proceedings of the Third IEEE Symposium on*, pages 600–607, 1991.

[5] M. Hind and E. Schonberg. Efficient loop-level parallelism in Ada. In *Tri-Ada*, pages 166–179, 1991.

[6] INMOS Limited. *Occam2 Reference Manual*. Prentice-Hall, London, 1988.

[7] H.G. Mayer and S. Jahnichen. The data-parallel Ada run-time system, simulation and empirical results. In *Proceedings of Seventh International Parallel Processing Symposium*, pages 621–627, 1993.

[8] E.K. Park, P.B. Anderson, and H.D. Dardy. An ada interface for massively parallel systems. In *Computer Software and Applications Conference, 1990. COMPSAC 90. Proceedings., Fourteenth Annual International*, pages 430 –435, 1990.

[9] J. Rakesh. Parallel ada: issues in programming and implementation. In *Proceedings of the fourth international workshop on Real-time Ada issues*, IRTAW '90, pages 126–132. ACM, 1990.

[10] A.W. Roscoe and C.A.R. Hoare. *The Laws of Occam Programming*. Oxford University Programming Research Group, PRG-53, 1986.