Multi-Processor Blocks

Luiz Adriano de Mello Ferreira

Pontifícia Universidade Católica de Minas Gerais, Instituto de Informática, Brazil

Abstract

Programming for multicore CPUs has been a challenge for many on recent times. And even more challenging is programming for multiple architectures with a varied number of processors, each of those with different characteristics. This paper briefly presents the Multi-processor Blocks programming paradigm that was designed in an effort to deal with those problems allowing asynchronous processing of information.

Keywords: multiple processor programming, programming paradigms

Authors' contact:
luizdemello@ymail.com

1. Introduction

Programming for multicore CPUs has been a challenge for many developers on recent times. And even more challenging is programming for multiple architectures with a varied and arbitrary number of processors.

Aiming to ease some of those problems, especially the lack of support for programmers in high level programming languages, the author of this paper designed the Multi-processor Blocks (MpB) paradigm. It was originally inspired by an Artificial Intelligence technique called Artificial Neural Networks (ANN) and the reader may find it interesting to be familiar with it, though it is not necessary. The major similarity between MpB and ANN is the fact that there is no predetermined number of information processing units (Processors and Neurons, respectively). This allows the paradigm to be flexible, being able to be adapted for use with both generic/nonspecific architectures (e.g. software for PCs), and very specific/regular architectures (e.g. IA for a specific model of robot).

The MpB paradigm was originally designed to be incorporated in high level programming languages and that is the focus of this paper. Even so nothing bars compilers and/or languages to incorporate a similar logic in the background, or even hardware.

2. Related Work

The ANN form one of the bases of this work and any work explaining them is therefore related to this paper's subject in a way or another. Information on ANN can be found in multiple sources, like the book Artificial Neural Networks: An Introduction to ANN Theory and Practice [Braspenning P.J. et al, 1995] or David Kriegel's [2007] manuscript called A Brief Introduction to Neural Networks, available online. Also, National Instruments' website provides a series of articles called Multicore Programming Fundamentals that touches many of the considerations made while creating this paradigm.

3. Presenting the Paradigm

The most basic concept of the MpB paradigm is to structure a set of instructions (a Block) in three Layers, like on ANN. The structure of these Blocks is what allows easy and efficient distribution of instructions among various processors.

Basically, a Block is composed by several Tasks that will be executed individually and asynchronously, each being processed by only one processor, or core (from now 'cores' are considered individual processors). The execution of the Tasks is the middle layer. The first layer holds a Distribution Function (DF) and an Execution Order Criteria (EOC) that are, together, responsible to group Tasks and regulate in what order those Tasks are going to be send to execution. This first layer is dynamic: the programmer only defines parameters so that the tasks can be distributed in an adaptative way according to the actual architecture during execution time. Lastly there is an additional Task called Output Function (OF) that is responsible to combine the results of each Task in the middle layer. It is on this layer that all the Tasks get synchronized (or rather, their results

The scope of a Block is similar to that of a function, which is to solve a unique problem; it does not need to encompass the whole program. It is also possible to have nested Blocks depending on how each language implements MpB.

3.1 First Layer: Distribution Function and Execution Order Criteria

Both the DF and EOC constitute the first layer of an MpB. This Layer roughly equals to the inputs layer of ANN. The 'inputs' would be the Tasks that comprises the Block, in other words, the second layer.

The responsibilities of both components of this Layer are intertwined and focus on the same objective:

to define the order in which the Tasks within a block are to be performed.

The DF arranges all the Tasks in Groups based on the Attributes and Sub-Attributes of each Task, and/or other characteristics that the programmer, language or compiler defines. It may be implicit and defined only automatically by the language and/or compiler, rather than explicitly declared by the programmer. By allowing customized arrangement of the Groups, it becomes possible to create better arrangements that consider the particular characteristics of the Tasks that the Block will perform, while automatic arrangement will only consider generic characteristics.

The EOC is what actually defines in what order the Tasks are to be performed. It uses the Groups, the order in which each Task appears within the Groups, and the characteristics of the processors as its main factors. Attributes and Sub-Attributes may also affect, but as a secondary factors. Observe that the DF and the EOC combined will tune the order in which the Tasks are performed considering both factors known at compilation time (the Tasks) and what may be unknown (each target platform architecture).

3.2 Second Layer: The Tasks

This layer is responsible for the majority of the work of an MpB most of times. It is also the one most similar to its counterpart in the ANN, which is the hidden layer. The number of Tasks may be known beforehand, but the number of processors (the 'neurons') that the target platforms possess will not be necessarily known.

Each Task may have both Attributes and Sub-Attributes, or just either of them. Attributes are meant to be the main determining factor used by the DF to group the Tasks, while Sub-Attributes provide additional information to complement the main attributes or that are conditional (blocks the execution while a specific state has not yet been reached. Conditional Sub-Attributes are called Requirements). Apart from the fact that Attributes cannot be conditional like Sub-Attributes can, they are essentially the same thing, the difference lies on their priorities.

The most important aspect of this layer is the fact that we know how many Tasks there are and it is also known that each Task will be executed exclusively by one processor thoroughly, but we do not know how many processors there are. Ideally the Tasks are designed to be run in parallel and asynchronously (as long as their Requirements are provided), regardless if there are more or less Tasks than processors. The only half-exception are Tasks that have the completion of other Tasks as Requirements. These obviously cannot run together with any of their Requirements and those that have them as a Requirement (parts of the same chain of Tasks that not necessarily will be executed by the same processor but will necessarily be executed one after the other). However they still have to be able to run in parallel with any other Task not in the same chain.

But what are Tasks? They are smaller groups of instructions that are better performed together and that together solve a smaller piece of a larger problem; that larger problems is the scope of the Block. Normally, all Tasks do not store persistent information in memory; rather they return any information whose goal is to remain in memory after the end of the Task and/or Block. The values, or sets of values, returned by each Task will be later on received and processed by the Output Function. A Task may still store persistent information in memory if it is not part of the return value, preferably if it is information that will not be accessed by any other Task (the DF does not count in this case) within the same Block. This is important to keep the ability of the Tasks to be performed asynchronously.

3.3 Third Layer: The Output Function

Roughly equivalent of the output layer of the ANN, the Output Function (OF) is responsible for the output of the whole block. This responsibility, however, should not be mixed with a return value. That simply means that the OF uses the partially processed results of each Task and stores it in memory to be used by other instructions after the Block is completed; a return value is entirely optional and may not even be available.

The main point of the OF is to perform a last set of instructions before the Block ends. These instructions should be done in this function either because it is impossible or inviable to spread them through multiple processors. The OF has parameters and they are not defined arbitrarily. They appear in the order and have the same type of the return value of each Task. The OF then should process those values and then store any information that needs to be persisted after the end of the Block. It may optionally return a value, but a language is not required to allow Blocks to be able to return any value.

3.4 Additional Information

There are two points that were not addressed in the previous sections, and are beyond the scope of this paper, but are important to consider. The first is automatically generated Tasks and the second is Tasks with no return value.

The major application of Tasks generated automatically would be within repetition cycles, such as the common For and While structures. Several factors would impact on the decision of a language to implement this. Chief among them is the access and use of data structures. Are they read-only or will be written? If they will be written, will the indexing structure of the data change or just the content in the already indexed positions of memory? Those and many other questions must be considered on any possible implementation, but it is possible nonetheless.

At the Second Layer it was said that Tasks normally always return one or more values. There is one exception, that is when all the Tasks have no return value, what would cause the OF to be optional. It could be used, for example, to walk through an array and perform some sort of operation only with the data on that cell and change its internal values. Each Task would operate in one or more positions, but there would be no need to later on return a value, much less combining any returned data as there would be none. The OF could still be executed if there was any operation we would like to perform, like setting a flag that this array has already been processed and is ready for use by some external agent like a thread that constantly scans the object and its flag property.

4. Example

For this example we will be assuming that Distribution Function and Execution Order Criteria are predefined, so we will not have to write it down for our specific Block. The Distribution Function groups Tasks by the first Attribute, which will be a string representing the name of the group to assign that Task (basically it is a manual distribution of Groups). We will also use a function-like attribute to define Tasks as Requirements for another Task. The Execution Order Criteria is simple. It picks the first Task not processed and with no Requirements blocking it from the first Group, then it picks the first of the next Group and so on, going back to the first Group after the last Group. We will also assume that all processors are roughly equal, so there is not much to customize about them. These assumptions are done in order to simplify the example, making it easier to focus on how the paradigm works overall.

First, let us take a look at the following pseudo code written in a traditional fashion:

```
function F_Traditional(reference a, reference b,
reference c)
    {
    1     a = (1+1) * (2+2) * (3+3) * (4+4) * (5+5) *
          (DataBaseValue1 + DataBaseValue1) *
          (DataBaseValue2 ** DataBaseValue3)

    2     b = a + random(0, 100)

    3     c = b – date.day_of_year

    4     Print(c)
    }
```

It is a simple code indeed. Now let us see how it would be structured on a Multi-processor Blocks language:

```
function F_MpBlocks(reference a, reference b, reference
c)

    Multi-processor_Instruction_Block
    Task One: "Group_1"
        return 1+1

    Task Two: "Group_1"
        return 2+2

    Task Three: "Group _1"
        return 3+3

    Task Four: "Group _1"
        return 4+4

    Task Five: "Group _2"
        return 5+5

    Task Six: "Group _4"
        Temp = Database.Retrieve(DatabaseValue1)
        return temp+temp

    Task Seven: "Group _3", Requires(Six)
        //This temp is local to the block and independent
        //from the one in Six
        Temp = Database.Retrieve(DatabaseValue2)
        return Power(Temp,
Database.Retrieve(DatabaseValue3))

    Task Eight: "Group _2"
        return random(0,100)

    Task Nine: "Group _4"
        return Date.day_of_year

    Output Function(int one, int two, int three, int four,
int six, int seven, int eight, int nine)
        a = one * two * three * four * five * six * seven
        b = a + eight
        c = b - nine
        print(c)
```

After the DF was executed, the Groups would be: Group_1, with Tasks One through Four; Group_2, with Tasks Five and Eight; Group_4 with tasks Six and Nine; and lastly Group_3 with Task Seven. Observe that the order of the Tasks in the Groups is the same as they were declared because that is how our DF works and the same is valid for the order of Groups. Let us assume that each Group takes the exact same amount of time to be processed and that each task within the same group uses the same amount of time of their peers. If the process time is 1 second, then Group_1's Tasks are done each in 0.25s, Group_2's in 0.5s, Group_4's in 0.5s, and Group_3's in 1s. We will also assume that the CPU has two regular processors (like a processor with two cores) and no other.

At first, Tasks One (first of Group_1) and Five (first of Group_2) will be selected for Processor 1 and Processor 2 respectively. At the end of the following 0.25s task One will be completed, and task Six (first of Group_4) will be selected. Notice that it selects Group 4 because that group appeared before Group_3 in our code. At total time 0.5s Task Five will be done, but we have a problem. Task Seven requires Six to be finished. We defined it like that because both access the database; we did not want concurrence since we do not know how this access is handled internally. In that case, task Two becomes the next in line, however Seven goes to an especial list of top priority list (or an extra group that has special characteristics) of tasks to be performed before any others if their requirements are complete. Then it proceeds by the same logic. The following table will describe in detail how the order will be:

	Task Entering Processor		**Task Using or Leaving Processor**	
Time (s)	1	2	2	2
0	One	Five	None	None
0.25	Six	None	One	Five
0.5	None	Two	Six	Five
0.75	Seven	Eight	Six	Two
1.25	None	Nine	Seven	Eight
1.75	Three	Four	Seven	Nine
2.00	None	None	Three	Four

If we assume that each isolated attribution and plus/minus operations takes 0.01s, and that printing and multiplying takes 0.1s, we would have the whole function processed in 2.75 at the end (time to process the Tasks plus the time to process each individual instruction in the Output Function).

If we executed the sequence coded in the traditional way we would see that: Line 1 is com posed by Tasks One through Seven, 6 Multiplications and one Attribution, taking a total of 3.61s to complete; Line 2 is Task Two, one Addition and one Attribution, taking 0.52s to complete; Line 3 is Task Nine, one Subtraction and one Attribution, with total time of 0.52s; lastly Line 4 is one Print, so it takes 0.1s. We could assume that a processor/compiler would organize the instructions by lines of code, given that each line but the fourth has an attribution and all of them use a variable defined in the previous line. By no means is it the only possibility, but it is one very likely to happen in a real scenario. Being that the case, we would have the whole Line 1 happening in one Processor. In the other we would partially process Line 2 and 3 (finishing 'Tasks' Two and Nine of each, respectively, while waiting to execute the Minus/Plus operations and the attributions). As their times are smaller, those two parts would be done before Line 1 is finished and the second processor would be idle. When the first processor finishes Line 1, we would then sum and attribute the result to finish line 2, and only then subtract and attribute to finish Line 3, since it cannot happen without Line 2 being complete. In the end we would perform Line 4. The total time would be 4.75s and in this case it would be sensibly slower. Observe that although the execution times were arbitrarily defined, they are consistent with real scenarios in regard to their proportion.

4.1 Considerations

In this example the Tasks were manually divided into groups that have the same processing time because each Task's processing time was previously known.

Observe that, instead of manually defining these groups, it was possible to use a compiler that has time estimative for each of its binary instructions and that uses the information to automatically estimate what assortment of Tasks would generate Groups that are more likely to have the same execution time or the closest possible. It could also be possible to add Sub-Attributes to determine the type of the processor where each Task would have a better performance (Generic/CPU, Video/GPU, etc.), and if it can be run by other types of processor if none of that type is free when it is the Task's turn, or if the Task can only be performed exclusively by that type of processor. That addition is actually recommended for generic and nonuniform architectures, and may be interesting on some specialized architectures.

Conclusion

This paradigm is an effort toward an easier and clearer way of programming for architectures with a variable number of processors. Its strongest characteristic is to allow programs to be written for any number and type of processors in the same fashion, leaving programmers able to focus in write code and letting the language and operational system worry about low level and platform specific aspects. It has potential to improve the way we program computers today, and to work as basis for new researches in this area. But potential alone does not achieve results, it is necessary to conduce more research and tests with real programming languages to transform that potential in reality.

Acknowledgments

The author would like to thank his friend Fortuna for the tips she gave; without her this paper would probably never exist.

References

Braspenning P.J., Thuijsman F. and A.J.M.M., 1995. Artificial Neural Networks: An Introduction to ANN Theory and Practice. *Springer.*

Kriesel, D., 2007. *A Brief Introduction to Neural Networks* [online]. Available from: http://www.dkriesel.com/en/science/neural_networks [Accessed 17 April 2010].

National Instruments, 2008. *Multicore Programming Fundamentals* [online] National Instruments. Available from: http://zone.ni.com/devzone/cda/tut/p/id/6422 [Accessed 12 April 2010].