Paradigms and parallel constructions in modern computing

Ognian Nakov[1], Nadejda Angelova[2], Desislava Andreeva[3] and Haralambos Dokomes[4]

Abstract - **In the article is described the common usage of multi-core systems and modern parallel programming constructions. Programming parallel paradigms are analyzed in real examples with structured multithreading, parallelization and execution of concurrent tasks over a mululiple-cores structure. Improvements of performance are achieved.**

Keywords – **Parallelism, Multithreading, Data Flow, Multitasking.**

I. INTRODUCTION

Today, performance is improved by the addition of processors. So-called multicore systems are now ubiquitous. Of course, the multicore approach improves performance only when software can perform multiple activities at the same time. Functions that perform perfectly well using sequential techniques must be written to allow multiple processors to be used if they are to realize the performance gains promised by the multiprocessor machines [1].

For some time now, programmers have had to think about a programming challenge related to parallelism — concurrency.

The Microsoft .NET Framework provides the asynchronous programming model and notions of background workers to facilitate the common programming concern for parallelism — concurrency [2].

Parallel programming differs from concurrent programming in that you must take what is logically a single task—expressible using familiar sequential constructs supported by all major languages — and introduce opportunities for concurrent execution [2]. However, when concurrency opportunities are introduced with subtasks that share data objects, you have to worry about locking and races.

We'll describe and analyze some major approaches to parallel paradigms and practices and illustrate their use through abstractions that are under development. In particular, we'll illustrate both the C++ Parallel Pattern Library (PPL) [2,3]and the Parallel Extensions to .NET using C#.

[1]Ognian Nakov is with the Faculty of Computer System and Control at Technical University of Sofia, 8 Kl. Ohridski Blvd, Sofia 1000, Bulgaria, E-mail: nakov@tu-sofia.bg

[2]Nadejda Angelova is with the Faculty of Computer System and Control at Technical University of Sofia, 8 Kl. Ohridski Blvd, Sofia 1000, Bulgaria, E-mail: nade.angelova@gmail.com

[3]Desislava Andreeva is with the Faculty of Computer System and Control at Technical University of Sofia, 8 Kl. Ohridski Blvd, Sofia 1000, Bulgaria, E-mail: dandreeva@tu-sofia.bg

[4]Haralambos Dokomes is with the Faculty of Computer System and Control at Technical University of Sofia, 8 Kl. Ohridski Blvd, Sofia 1000, Bulgaria

II. STRUCTURED MULTITHREADING

Structured multithreading refers to providing parallel forms of key block-structured sequential statements. For example, a compound statement { A; B; } with sequential semantics where A is evaluated and then B is evaluated is made into a parallel statement by allowing A and B to be evaluated concurrently. The whole construct, however, does not complete, and control continues to the next construct until both subtasks have finished. This is an old concept and historically taught as a cobegin statement. It is sometimes referred to as "fork-join parallelism" to emphasize the structure. The same basic idea can be applied to loops where each iteration defines a task that may be evaluated concurrently with all the other iterations. Such a parallel loop completes when all iteration tasks complete.

A familiar example of the divide-and-conquer concept is the famous QuickSort algorithm. We'll illustrate a straightforward parallelization of this algorithm using C++ constructs. There are 2 new features involved. The first feature demonstrated is the new **C++ lambda syntax** that makes it extremely convenient to capture an expression or statement list as a function object. The new syntax:

[=] { ParQuickSort(data, mid);

creates a function object that, when it's invoked, will evaluate the code between the braces:

```
void ParQuickSort(T * data, int length, T* scratch)
{
    …
    int mid = ParPartition(data[0], data,length,..);
    parallel_invoke(
    [=] { ParQuickSort(data, mid); },
    [=] { ParQuickSort(data+mid, length-mid); });
}
```

The **leading [=]** marks the lambda and indicates that any variables in outer scopes referenced in the expression should be copied into the object and that references to those variables in the body of the lambda will refer to those copies.

The **parallel_invoke** is a template algorithm that, in this case, takes two such function objects and evaluates each one as a separate task so that those tasks may run concurrently. When both tasks complete and, in this case, both of the recursive sorts have been completed, the parallel_invoke returns and the sort is then complete.

A use of parallel loops might be code to perform a ray-tracing problem. Such a problem is trivially parallel over each

output pixel. This is expressed using the **Parallel.For** method from the Parallel Extensions to .NET. This code assumes that the various methods invoked in the body of a loop are safe for concurrent execution.

```
public void RenderParallel (Scene scene, Int32[ ] rgb)
{
    Parallel.For (0, screenHeight, y =>
    {
        Parallel.For  (0, screenWidth, x =>
        {
          … });
    }
};
```

Structured multithreading is ideal for working with parallelism where there is a natural, perhaps recursive, possibly irregular, data structure where the parallelism reflects that structure. The following example traverses a graph in a topological order and don't visit a node before having seen all predecessors. After visiting a node, we decrement the counts of successors (careful to make this a safe operation since multiple predecessor tasks may attempt it at once).

```
void topsort(Graph * g, void (*action)(Node*))
{
    g->forall_nodes([=] (Node *n)
    {
        n->count = n->num_predecessors();
        n->root = (n->count == 0);
    });
    g->forall_nodes([=] (Node *n)
    {
        if(n->root) visit(n, action);
    });
}
```

We have two phases: the first counts predecessors and identifies root nodes; the second starts a depth-first search from each root that decrements and ultimately visits successors:

```
// Assumes all predecessors have been visited.
void visit(Node *n, void (*action)(Node*))
{
    (*action)(n);
    parallel_for_each(n->successors.begin(),
    n->successors.end(),
    [=](Node *s)
    {
        if(atomic_decrement(s->count) == 0)
        // safely does "-- s->count"
        visit(s, action);
    });
}
```

The **parallel_for_each** method traverses the list of successors, applies a function object to each, and allows those operations to be done in parallel. Not shown is the assumed **atomic_decrement** function that uses some strategy for arbitrating concurrent accesses.

The structure of this algorithm guarantees that "action" is given exclusive access to its parameter so no additional locking is needed if action updates those fields. Further, there are guarantees that all predecessors have been updated and are not changing, and that no successor has been updated and will not change until this action completes.

III. Data Parallelism

Data parallelism refers to the application of some common operation over an aggregate of data either to produce a new data aggregate or to reduce the aggregate to a scalar value. The parallelism comes from doing the same logical operation to each element independent of the surrounding elements. There have been many languages with various levels of support for aggregate operations, but by far the most successful has been the one used with databases—SQL. LINQ provides direct support in both C# and Visual Basic for SQL-style operators, and the queries expressed with LINQ can be handed off to a data provider, such as ADO.NET, or can be evaluated against in-memory collections of objects or even XML documents.

Part of Parallel Extensions to .NET is an implementation of LINQ to Objects and LINQ to XML that includes parallel evaluation of the query. This implementation is called PLINQ and can be used to work conveniently with data aggregates.

The difference between LINQ and PLINQ is the **AsParallel** method on the data-collection points. One subtle point is the opposite - **Aggregate** operator. His third parameter is a delegate that provides a mechanism to combine results. With this method, the implementation is done in a parallel style by blocking the input into chunks, reducing each chunk in parallel, and then combining the partial results.

IV. Data Flow

A common technique for exploiting parallelism is through the use of pipelining. Using this model, a data item flows between various stages of the pipeline where it is examined and transformed before being passed on to the next stage. Data flow is the generalization of the idea where data values flow between nodes in a graph, and computation is triggered based on the availability of input data. Parallelism is exploited both by having distinct nodes executing concurrently and by having one node activated multiple times on different input data.

Parallel Extensions to .NET supports the ability to explicitly create individual tasks (of type Task, the underlying mechanism for implementing structured multitasking) and then to identify a second task that begins execution when the first completes. **The concept of a future** is used as a bridge between the worlds of imperative programming and data-flow programming. A future is the name of a value that will eventually be produced by a computation. This separation allows me to define what to do with a value before I know that value.

The **continueWith**() on a future is parameterized by a delegate that will be used to create a task that will be executed when the future value is available. The result of a call to continueWith is a new future that identifies the result of the delegate parameter.

As an example of this style, consider the parallelism within the Strassen optimized matrix multiplication algorithm.

One of these tasks might look like this:

var m1 = Future.StartNew(()
=> (A(1,1)+B(1,1))*(A(2,2)+B(2,2));

The first seven of the subtasks are independent, but the last four depend on the first seven as inputs. The basic data-flow graph would look like **Figure 1**. Here, the task labeled c11 depends on the results of tasks m2 and m3. I want that task to become eligible for execution when its inputs are available. In C# this can be expressed as:

var c11 = **Task.ContinueWhenAll**(delegate { ... }, m2,m3);

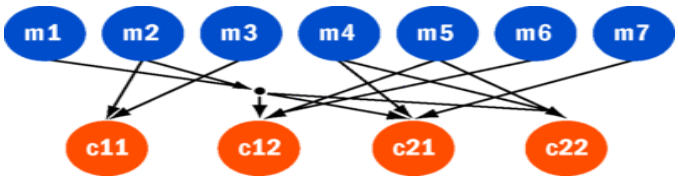

Figure 1

V. Streaming Parallelism

Beyond multiple cores, a second important feature of computer architecture is the **multiple layers of memory hierarchy**: registers, one or more levels of on-chip cache, DRAM memory, and, finally, demand paging to disk. Most programmers are blissfully unfamiliar with this aspect of system architecture because their programs are modest in size and fit well enough in the cache to which most references to memory are quickly returned. However, if a data value is not in the on-chip cache, it can take hundreds of cycles to fetch it from DRAM. The latency of providing this data makes the program appear to run slower since the processor spends a large fraction of the time waiting for data.

Some processor architectures support multiple logical processors per physical processing core. This is usually called (hardware) multithreading, and modest amounts of this have been used in mainstream processors (Intel has called this hyper-threading in some of its products). A motivation for multithreading is to tolerate the latency of memory access; when one logical hardware thread is waiting on memory, instructions can be issued from the other hardware threads.

As the number of processing cores grows, the number of requests it can make on a memory system increases and a different problem emerges, one of bandwidth limitations. A processor will be able to support only so many transfers per second to or from DRAM memory. When this limit is reached, there is no chance to get any gains through further use of parallelism; additional threads will just generate additional memory requests that will simply queue up behind earlier requests and wait to be serviced by memory controllers.

While there have been proposals for special purpose languages to allow streaming algorithms to be specified and their execution carefully planned, it is also possible to achieve this in many cases by careful scheduling. For instance, you can apply this technique to the QuickSort example. If the size of the data set you are sorting is so large that it does not fit in the cache, the straightforward work-stealing approach will tend to schedule the largest and coarsest subproblems onto different cores, which then work on independent data sets and lose the benefit of a shared on-chip cache.

If, however, you modify the algorithm to use only parallelism on data sets that fit in the cache, you gain the benefits of streaming. In this example, you still break large problems into smaller problems (and use parallelism in the partitioning step), but if both sub-problems won't fit in the cache at the same time, then we'll do them sequentially.

```
Use Parallelism on Smaller Data Sets
if(sizeof(*data)*length < cache_size)
{
      parallel_invoke(
      [=]{ ParQuickSort(data, mid, cache_size);
      [=]{ ParQuickSort(data+mid, length-mid, cache_size);});
}
else
{
      ParQuickSort(data,mid,cache_size);
      ParQuickSort(data+mid, length-mid, cache_size);
}
```

VI. Single – Program and Multiple

In the high-performance computing arena the kinds of problems are dominated by parallel loops over arrays of data where the bodies of the loops typically have a fairly simple code structure.

The earlier ray-tracing fragment is an example. The dominant parallelism model that emerged was called single-program, multiple-data, frequently abbreviated as SPMD. In this model, programmers think about the behavior of each processor (worker, thread) as a set of processors that logically participate in a single problem but share the work. Typically the work is separate iterations of a loop that work with different parts of arrays.

The notion of work sharing in an SPMD style is at the heart of the OpenMP of extensions to C, C++, and Fortran. The core concept here is a parallel region where a single thread of activity forks into a team of threads that then cooperatively execute shared loops. A barrier synchronization mechanism is used to coordinate this team so that the entire team moves as a group from one loop nest to the next to insure that data values are not read before they have been computed by teammates. At the end of the region, the team comes back together, and the single original thread continues on until the next parallel region.

VII. CONCURRENT DATA STRUCTURES

The previous discussion has focused almost entirely on control parallelism - how to identify and describe separate tasks that can be mapped down to the multiple cores that may be available. There is also a data side related to parallelism. If the effect of a task is to update a data structure, say insert a value into a Hashtable, that operation may be logically independent of concurrently executing tasks.

Simply putting a single lock around the whole data structure may create a bottleneck in the program where all of the tasks serialize, resulting in a loss of parallelism because too few data locations are concurrently in use.

It is thus important to create, in addition to the parallel control abstractions, new concurrent versions of common data structures - Hashtables, stacks, queues, various kinds of set representations. These versions have defined semantics for the supported methods that may be invoked concurrently, and they are engineered to avoid bottlenecks when accessed by multiple tasks.

As part of the PPL and Parallel Extensions to .NET, Microsoft provides suitable implementations of vectors, queues, and Hashtables that can be used as building blocks.

VIII. CONCLUSION

We propose an illustration of modern parallel paradigms introduced in C++ Parallel Pattern Library (PPL). The first feature analyzed is the new C++ lambda syntax, illustrated in an example of statement list in a function object.

A parallel solution is presented also for a pixel iteration task. Some suggestions are made concerning structured multithreading which is ideal for dealing with parallelism. A programming technique is proposed for exploiting parallelism through the use of pipelining. Different programming tools are included - multitasking, parameterization through delegates, coherence with multicore arhitecture and multithreading.

REFERENCES

[1] Jouppi N., The future evolution of High – Performance microprocessors, Stanford University, Stanford.edu/class/ee380/abstract/060927.html
[2] MSDN Xi 2008, Microsoft Corp.
[3] Bokar Sh. Thousand core chips: a technology perspective, video.dac.com/44th/papers/42_1.pdf