Journal of Software Engineering and Applications, **2012, 5, 1023-1030**
http://dx.doi.org/10.4236/jsea.2012.512119 Published Online December 2012 (http://www.SciRP.org/journal/jsea)

A Distributed Virtual Machine for Microsoft .NET*

Morten N. Larsen, Brian Vinter

The Niels Bohr Institute, University of Copenhagen, Copenhagen, Denmark.
Email: momi@nbi.ku.dk, vinter@nbi.ku.dk

Received October 2nd, 2012; revised November 4th, 2012; accepted November 15th, 2012

ABSTRACT

Today, an ever increasing number of natural scientists use computers for data analysis, modeling, simulation and visualization of complex problems. However, in the last decade the computer architecture has changed significantly, making it increasingly difficult to fully utilize the power of the processor, unless the scientist is a trained programmer. The reasons for this shift include the change from single-core to multi-core processors, as well as the decreasing price of hardware, which allows researchers to build cluster computers made from commodity hardware. Therefore, scientists must not only be able to handle multi-core processors, but also the problems associated with writing distributed memory programs and handle communication between hundreds of multi-core machines. Fortunately, there are a number of systems to help the scientist e.g. Message Parsing Interface (MPI) [1] for handling communication, DistNumPy [2] for handling data distribution and Communicating Sequential Processes (CSP) [3] for handling concurrency related problems. Having said that, it must be emphasized that all of these methods require that the scientists learn a new method and then rewrite their programs, which mean more work for the scientist. A solution that does not require much work for the scientists is automatic parallelization. However, research dating back three decades has yet to find fully automated parallelization as a feasible solution for programs in general, but some classes of programs can be automatically parallelized to an extent. This paper describes an external library which provides a Parallel. For loop construct, allowing the body of a loop to be run in Parallel across multiple networked machines, *i.e*. on distributed memory architectures. The individual machines themselves may be shared memory nodes of course. The idea is inspired by Microsoft's Parallel Library that supplies multiple Parallel constructs. However, unlike Microsoft's Library our library supports distributed memory architectures. Preliminary tests have shown that simple problems may be distributed easily and achieve good scalability. Unfortunately, the tests show that the scalability is limited by the number of accesses made to shared variables. Thus the applicability of the library is not general but limited to a subset of applications with only limited communication needs.

Keywords: Microsoft .NET; Parallelization; Distribution; Data Parallelism

1. Introduction

During the last decade the usage of high performance computing has increased beyond classic areas for scientific computing, the type of problems that are solved by high performance computing has widen, but most importantly the user group has changed from programming specialist to a more mixed group of scientists from fields like chemistry, physics, environmental sciences, engineering etc. These two factors have meant that the tools for aiding the users in handling hardware are more important today than ever before. As a natural consequence, there is an increase in the solutions that can help the users. Solutions ranging from automatic parallelization to tools like Message Parsing Interface (MPI) and Communicating Sequential Processes (CSP). Nevertheless, many of the tools available have very little usage in practice and/or do not provide enough scalability compared to the manually written code. However, the greatest problem is that many of the tools have a very steep learning curve, and thus, presents problems for many non-computer specialists, who may be able to write a sequential program, but do not have knowledge of locks, raise conditions, deadlocks and memory layout.

In an attempt circumvent this problem Microsoft has in recent years improved .NET with tools to help users write Parallel code. The functionality resides mainly in the Microsoft Parallel Library [4] and consists of a set of tools; however, this paper focus exclusively on one, namely the Parallel.For construct. The construct as the name reveals, is the Parallel version of the normal For-loop. The usage is very simple and the users should in theory just replace the For-loops with the Parallel.For loop and the code will then be executed across all available

*The Innovation Consortium supported this research with grant 09-052139.

cores in the machine. Importantly though; in the current version of the tool the parallelization does not go beyond a single shared memory machine.

To improve Microsoft's idea by enabling distribution beyond a single machine, we have examined Microsoft .NET and the Microsoft Parallel system and will in this paper describe a solution for adding an external module to the system. The focus has been on making minimal changes to the code compared to the original code with a Parallel.For loop. Furthermore, the use of Microsoft.Parallel has been replaced by our implementation named DistVES (Distributed Virtual Execution System) as described in this work. From the beginning it was clear that the proposed solution would not work for every type of .NET program especially not programs with many interrupts, GUI programs, programs that have a lot of disk usage, etc. Therefore, the target programs have been limited to scientific application e.g. data analysis, modeling, simulation and visualization. Furthermore, simple algorithms which should yield good speed-up have been chosen for testing the initial version.

The rest of the paper is structured as follows: Section 2 gives a short introduction to Microsoft's Common Intermediate Language, which is the level at which DistVES transforms the original code. Section 3 gives a description of the design including consistency, client/server and code generation. In Section 4 the results of running a number of benchmarks are discussed. Future work is described in Section 5 and finally Section 6 gives a summary of our findings.

Related Work

DistVES is as mentioned above, closely related to Microsoft Parallel Library with the main difference that DistVES supports multiple machines. This clearly changes the intrinsic properties of the two systems, but for the users the two systems seem similar. Another closely related system is OpenMP [5] which needs to be incorporated in the compiler of a given programming language and many C/C++ and Fortran based programing languages are supported including .NETs Visual C++. Originally, OpenMP only supported shared-memory multiprocessor platforms, but IBM has orked on a version that supports a cluster [6]. Yet another way to help the programmer is to have support for distributed shared memory on the .NET objects. However, due to problems with scalability and usability, these types of systems have never proved a good solution [7]. Common for the three methods is that they only result in good scalability when the implemented algorithms are very simple and straightforward to parallelize.

A lot of research over the last decades has been dedicated to auto-parallelization. The general position is that it only works for very simplified algorithms and therefore alternative solutions must be found. Instead of auto-parallelization systems, some systems focus on making the communication between machines easier. Systems like the MPI provide functionality to distribute and run tasks on a large set of computers and gather the results of the computations. Likewise systems of the CSP type provide mechanisms of communication between different machines. The goal of CSP is to help the programmer writing correct code e.g. free of live-locks, dead-locks, and race conditions.

Ultimately, before most scientists can fully utilize large Parallel machines, it might be that a whole new approach for making hardware and new Parallel programming languages must be defined [8].

2. The Common Intermediate Language (CIL)

Before describing the design of DistVES, we will give a short introduction to the Common Intermediate Language (CIL) as the language is not commonly known. CIL is the backbone of the .NET framework and is a stack-based; platform neutral and type safe object oriented assembly language designed for .NET. The purpose of CIL is to allow multiple source-languages e.g. C#, VB.NET, and F# to be compiled into the same non-platform specific assembly language. The .NET runtime can then at runtime compile the CIL assembly to a machine specific machine code. This firstly allows for cross platform usage and secondly that programs written in e.g. C# can call methods from libraries written in languages like F# or VB.NET. **Figure 1** gives an overview of the pipeline from source language to machine code.

3. Design

The design of DistVES consists of three components; a distribution model, a client/server model, and a code generator. These all play a role in turning a .NET program with a Parallel. For construct into a distributed program that can be executed on a cluster computer. The shared fields play a key role in the system, as they should be identified in the original .NET program and made into distributed variables. Thereby making them available to all the clients in the system. Furthermore, the coherence model should ensure that the clients always see the current version of a shared variable.

3.1. Distribution including Server/Client

We start by giving an overview of the model and then go through the details about data coherence and code generation.

For simplicity of implementation a central server model without client-to-client communication has been

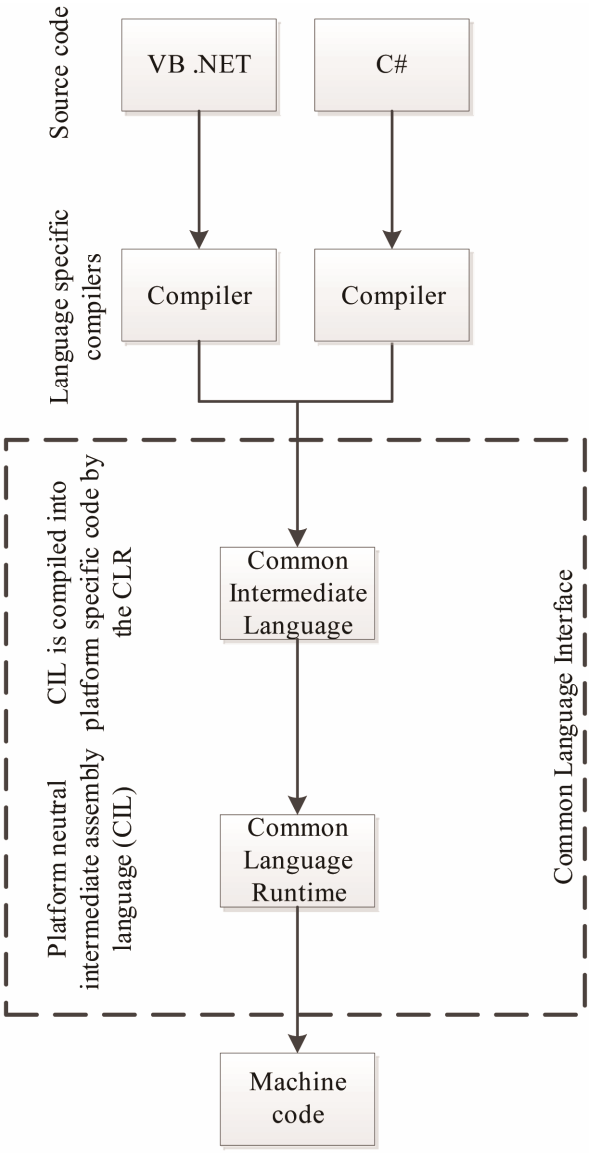

Figure 1. Overview of the CIL pipeline.

chosen for the initial version. This naturally sets an upper limit of scalability, but it will be able to show if our idea has potential. Each node in the system runs a thread which is dedicated for server communication. Again for simplicity, a single machine running multiple workers still runs one communication thread per worker, even though the workers could share a single communication thread.

Sending messages over a network requires that the objects are serialized before being sent and then deserialized at the receiving end. .NET supports automatic serialization of a class when marked with the *Serializable* attribute. Many of the built-in types in .NET are marked with this attribute, but when a programmer makes a new type it is not by default serializable. Therefore, DistVES only allow the use of the primitive types e.g. int, double, float, char and single/multi-dimension arrays of primitive types which all are serializable.

When distributing a .NET program with a Parallel construct the compiler generates an action delegate (subclass to the caller class) which contains the code from inside the Parallel construct. This is unfortunately not clear from the source code and means that some local variables can be promoted to a field in the delegate (see **Table 1**). Furthermore, the delegate will hold a reference to the caller class. During a normal run this reference is somewhere in the local memory and may be accessed from multiple threads, but when the program is being distributed, this reference can point to a memory location on another machine. As we cannot make a deep copy, because the class may possible not be *Serializable*, every client must create a local copy that mirrors the original. At the same time a given field must have the same unique identifier in all local copies of a given class. Through this process, DistVES can ensure that updates made to one field will be distributed to all clients. In practice, this is done by having all clients register all fields using the class ID and the field name with the server when executing the constructor of a given class. The server will then return the fields unique ID, which will be used for the rest of the execution.

3.2. Data Consistency

Maintaining multiple copies of the same object on different machines requires a system to ensure data consistency, so that all machines see the same version of the data like on a conventional shared memory machine. However, having systems with latency and transfer time means that we cannot guarantee at any given point that all machines have the exact same version of an object. Nevertheless, we can guarantee that all machines at some point will get the most recent version of the object. This is called sequentially consistency [9]. More relaxed consistency models exist [10], but in order to utilize them information about access patterns is required. As the CIL assembly does not contain information about access patterns, the programmers need to annotate the source code to use a more released system. However, making the programmers annotate the code is in conflict with the goal of making it easier for the programmer to utilize distributed computers. An implementation of sequential consistency could be the MESI protocol [11,12], which is known from hardware cache implementations. The MESI protocol relies on an object in cache at a given time having one of four states Modified, Exclusive, Shared or Invalid. The state of an object can change over time depending on either local or remote (other caches) making changes to the object. As seen in **Figure 2** the state of an object changes whenever an action is made to the object.

Table 1. Source code example, followed by the assembly view of the compiled code (the CIL instructions are omitted).

```
class Test
{
    // The following variable is a private field in the Test class
    int globalValue = 0;

    public void Run()
    {
        // The following local variable in the class Test is "promoted"
        // to a field in the delegate because it is accessed within the delegate
        int value = 0;

        /*
         * Code inside the Paralle.For loop is compiled to a subclass of
         * Test (a delegate). If the body of the Parallel.For did not touch
         * local variables in the Run method, the body of Paralle.For would
         * be compiled to a method in the Test class
         */
        Parallel.For(0, 10, i =>
            {
                int count = i;
                value = count;
                globalValue = count;
            }
        );
        // Next line gives compiler error, because the variable count is out of scope
        // value = count;

        // This should print "Value is 9 and 9"
        Console.WriteLine("Value is {0} and {1}", value, globalValue);
    }
}
```

```
___[MOD] …CodeExample.exe
    |         M A N I F E S T
    |___[NAMESPACE] CodeExample
    |    |___[CLASS] CodeExample.Test
    |    |    |        .class private auto ansi beforefieldinit
    |    |    |___[CLASS] <>c__DisplayClass1
    |    |    |    |        .class nested private auto ansi sealed beforefieldinit
    |    |    |    |        .custom instance void [mscorlib] System.Runtime…
    |    |    |    |___[FIELD] <>4__this : public class CodeExample.Test
    |    |    |    |___[FIELD] value : public int32
    |    |    |    |___[METHOD] .ctor : void()
    |    |    |    |        <Run>b__0 : void(int32)
    |    |    |
    |    |    |___[FIELD] globalValue : private int32
    |    |    |___[METHOD] .ctor : void()
    |    |    |___[METHOD] Run : void()
```

Non-active cache actions are operations made by a remote cache, whereas active cache actions are operations done by the local cache. Snoop and Update actions are in the MESI model broadcasted to all other nodes in the system. Snoop broadcasts always include a type which is either "Write" or "Read" depending on how the shared object is accessed by the cache.

The next question is how to integrate the MESI protocol to an object in .NET. The most obvious way is to encapsulate all objects (those from shared fields) into a custom DistVES object which contains both the original object and the control code to acquire the functionality of the MESI protocol.

The first task in designing the custom object is to define the methods that are required to have a correctly working MESI protocol. Firstly, it should be possible to access (write/read) the original object inside the custom object. These methods are called from the user code but are blocking if the custom object's MESI state requires that the server must be contacted e.g. for an updated version of the data. Secondly, the MESI protocol requires that it is possible to "remotely" snoop the object along with the possibility to "remotely" update the object. These two methods are called from the communication thread and if the update method is called it releases the blocking user code. This is typically done in a situation

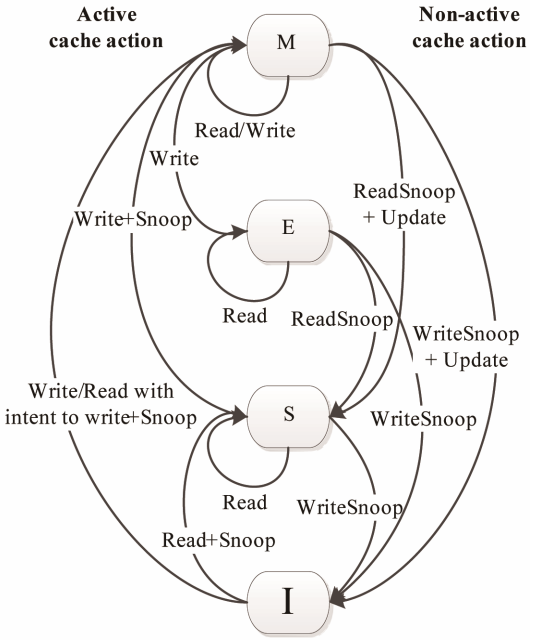

Figure 2. State transactions in the MESI protocol.

where the user code accesses an object with MESI state "invalid". Then the server is asked for an updated version of the data and the user code is blocking while waiting for a response.

The response is handled by the communication thread and will update the data before requesting the blocking user code to continue work. Furthermore, the communication thread should handle snoop request, which mainly involves changing the MESI state of objects and/or sending an updated version of data to the server.

Now that the custom object can handle the MESI protocol, the next step is to define how the object should integrate the different types that a field can have. The shared fields in the user code can be divided into two types; value-type and reference-type. Value-type fields have the value encapsulated into the field, whereas reference-type contains a reference to an object. This yields two different implementations of the custom object as the MESI states should follow the data and not the field. Therefore, if the field is a value-type then the field itself should be a custom object. In contrast, if the field is a reference-type then the referenced object should be a custom object. **Figure 3** illustrate a field with a value-type where the type of the field has changed from "int" to the custom object named "MESIValueField<int>". The MESIValueField contains all control code to correctly handle the MESI protocol.

Figure 4 shows the case of a shared field with a reference-type to an object of type "MyObj" which again contains a shared field of value-type "int". The type of the shared field is now changed from "MyObj" to "MESIReferenceField<MESIReference<MyObj>>". MESI-

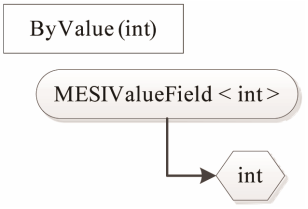

Figure 3. A shared "ByValue" field encapsulated in a MESI object.

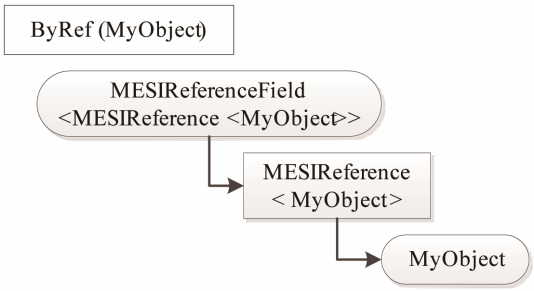

Figure 4. A shared "ByReference" field incapsulated in a MESI object.

ReferenceField does not contain the MESI protocol; however, it contains functionality to notify others if the field is assigned a new object (reference). The MESI protocol is implemented in the custom object named MESIReference which contains a reference to the actual "MyObj" object.

It should be noted that a special case arise with reference-type fields if the referenced object is of the type Array. The difference is that the real data of the arrays are the elements of the arrays and these are accessed through the CIL instructions Ldelem/Stelem. Therefore, we need a special case to handle arrays which we define as "MESIValueArray" (see **Figure 5**). This object has a MESI state for each of the elements in the array, but furthermore has support for defining a block size. Thus, enabling the control code to handle blocks of elements in order to minimize the overhead when accessing a large part of an array iterative.

3.3. Code Generator

The third component of DistVES is the code generator, which has the responsibility for transforming the original code into a distributed version of the same code. To do this, the code generator must first make a complete tree-based structure of the code to ensure efficient rewriting. The tree contains information on relations between classes and instructions, e.g., the Add instruction takes two arguments, which means that the Add instruction must have two incoming instructions. If the result of the addition is afterwards stored in a variable the Add instruction has an outgoing instruction, which is the store instruction. Furthermore, it is necessary to identify instructions that

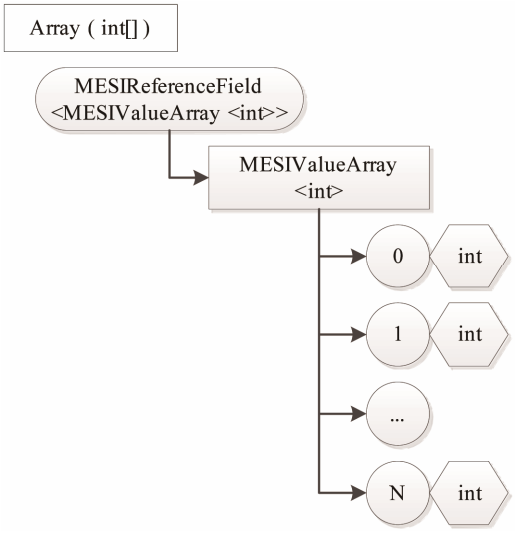

Figure 5. A shared Array field incapsulated in a MESI object.

call another method.

During the actual code generation all Parallel.For loops are identified and the name of the delegate class, which is the body of the loop, is noted. The next step is to modify these classes. As we know that the fields are the only type that can be shared between threads, the fields are a good starting point in order to keep the modifications of instructions to a minimum. As there only are to CIL instructions to access a field namely the Ldfld and Stfld instructions, the code generator looks for these two instructions and when finding them, a recursive modification using the incoming and outgoing instructions starts.

4. Benchmarks

In order to test the performance of the implementation three algorithms were implemented using the Parallel.For constructs. The tests were executed on four machines each with an Intel i7-860 processor at 2.8 GHz and 8 GB of RAM. The machines were connected using a Gigabit network through a single switch. The experiments were performed with 1 - 16 workers (1 - 4 workers per machine) and repeated five times to get consistent measurements. The tests labeled *Microsoft* Parallel.For and DistVES Parallel.For was executed on a single machines. Tests labeled *DistVES Network* (3) indicates that one machine ran the server and main client, and the three others machines ran the workers (3, 6, 9 and 12 workers in total). Finally the tests labeled *DistVES Network* (3 + 1) means that all machines ran the same amount of workers giving a total of 4, 8, 12 and 16 workers. In addition one of the machines ran the server and the main client.

The three test applications were written in C# and the source code was not changed between running with the Microsoft .NET Parallel.For and the DistVES Parallel.For other than a switch indicating which Parallel.For method to use.

- Black-Scholes: The algorithm gives the price of European style options and is frequently used in the financial world.
- Ising: A Monte Carlo simulation of the ising model which is a mathematical model for simulating magnetism in statistical mechanics.
- Prototein: Simplification of protein folding with only 2 dimensions and folding in angles of 90 degrees.

Discussion

The Black-Scholes is an embarrassingly Parallel problem; it has very little input data and generates only a single double value as output. Therefore, a good speed-up is expected from both Parallel methods. As seen in **Figure 6** linear speed up is achieved using one to six workers. Afterwards, the two tests using multiple machines still show an increase in speed-up, but with a much lower gradient which flattens as the number of workers increases. The single machine tests show that both methods scales well when running; however, both methods generate some overhead, which result in s scaling that is not perfect. The result is acceptable as the code is much easier to write, than the code required to make perfect scaling. In the network tests, we have a good scaling when using one or two cores per machine; however, using more than three cores result in decreasing utilization. The primary problem is the time span between the main client creating tasks and the initial work being distributed to the workers. The time span was measured to around 25% of the total running time, when using 16 workers (4 per machine). A secondary problem is the time required in

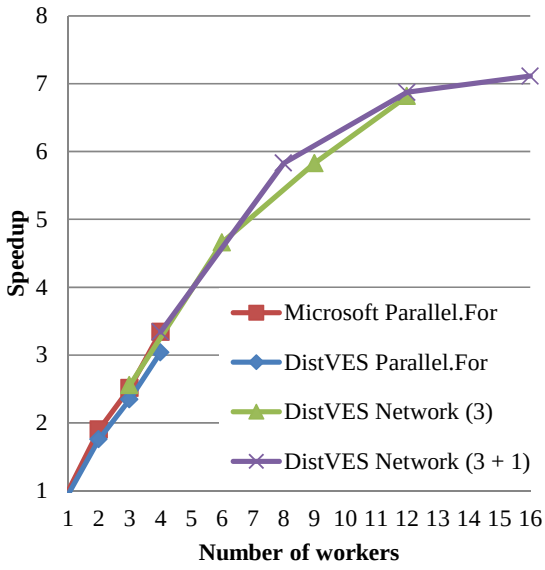

Figure 6. Black-Scholes.

the server to handle messages from clients. The time span is too high between a client sending a request and receiving the responds.

The Ising simulation is Monte Carlo based and thereby embarrassingly parallel as well. On the other hand the Ising simulation contains a barrier to synchronize the calculation of each round. The cost of the synchronization would increase and become the dominating factor if we ran the simulation on a fixed problem size and just increased the number of workers. Therefore, we ran this test increasing the problems size when the number of workers increases. The size of the array for a single worker is 3500 × 3500 elements, for two workers 7000 × 3500, for three workers 10,500 × 3500 and so forth. It was not possible to make a run using a total of 16 workers due to memory restraints. When using DistVES these arrays must be transferred even though one of the tests is executed on a single machine, on the contrary Microsoft Parallel use shared memory, and therefore access the memory directly. Therefore, we expect that Microsoft Parallel will scale better than DistVES. A decrease in utilization should furthermore be expected when using the network. As we see in the Gustafsson graph in **Figure 7**, DistVES is actually outperformed with 20%, which is a bit high. Nevertheless, none of the four methods are close to the optimal horizontal line. The two network tests show a linear decrease in utilization and when using 12 workers the utilization becomes less than 50%. The main problem is the overload of the central server and the barriers, but also the high number of accesses to the elements in the array. Each element access has a higher cost in DistVES because of all the bookkeeping required to guarantee consistency among other things.

The final benchmark is the simplified protein folding, which again should yield very good speed-up. The initial step in the program is that the main client creates tasks each containing a partly folded prototein. The tasks are then distributed to the clients, which locally keeps a copy of the fully folded prototein with the highest score. When all prototeins are folded the main client collects the best scores from the workers and finally finds the overall best prototein structure.

As with the Ising simulation, the Prototein folding have an input of some size; however, it is not as large as the Ising simulation. Furthermore, the Prototein folding does not require any barriers to synchronize calculations. Therefore, the expectations are a linear scaling where Microsoft Parallel will have a better gradient e.g. closer to optimal scaling. As seen in **Figure 8** all methods show good scaling, again; however, the two network tests show a decrease when using more than two workers per machine.

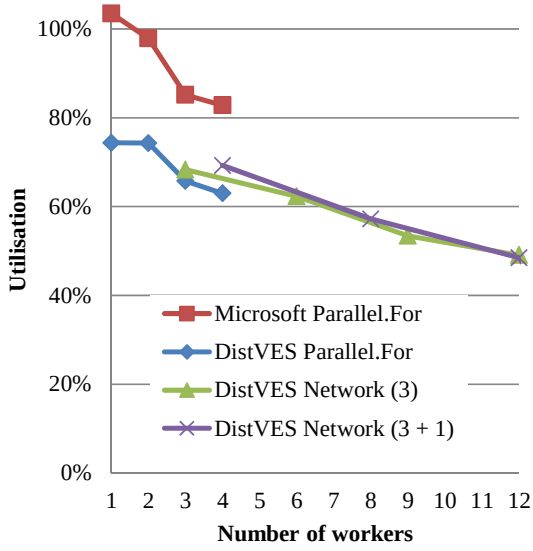

Figure 7. Ising simulation.

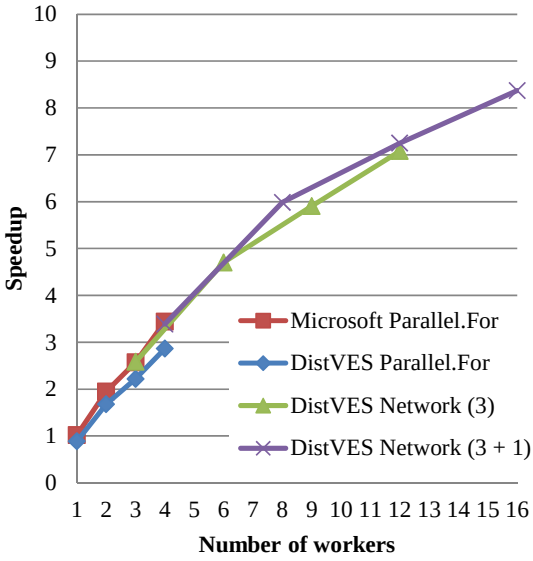

Figure 8. Prototein folding.

Furthermore, the two single machine tests show that DistVES scales as well as the method provided by Microsoft; however, DistVES is a bit slower due to more overhead.

The reason for utilization of only 50% when employing a total of 16 workers is the same as seen in the previous tests, namely the cost of the element access.

5. Future Work

As seen in the previous section the major problem with the current version of DistVES is the high cost of accesses to shared variables. Furthermore, the consistency model is strict and to gain a better performance a more released consistency model is required. There are a couple of ways to achieved this; either by code annotation,

by a more intelligent code generation, or by modifying the virtual machine of .NET (VES/CLR).

It is clear, that code annotation would make it easier to implement a more release consistency model like entry consistency. However, this shifts attention away from making it easier for the programmer to write code, which is essential. Unfortunately, it seems that achieving a good performance is not possible without code annotation.

The second option is to make the code generator more intelligent. A solution could be to categorize shared fields into read-only (write-once) and read-write. Thereby, the control code for the MESI protocol could be skipped by read-only fields and they would work like a normal field, making the performance better.

The best solution is properly to integrate a system like DistVES directly into the VES, but the open-sourced Mono project is the only choice for implementation as the Microsoft implementation of .NET is closed-sourced. The gain is that the user's code does not need to be changed at all and the accesses to shared variable will be the same running with or without a distributed Parallel.For. There will; however, be added some overhead when using the distributed version, but it will hopefully be less than in DistVES. The main concern is that the incredible effort it will require to modify the execution engine of Mono will not be justified by the gained speed-up.

6. Conclusion

Improving ease-of-use for scientists with limited programming knowledge to utilize the available hardware on multi-core and cluster computers is very important. Therefore, much effort has been put into making tools that assist the scientists; however, many tools are not widely used and/or will not give the wanted scalability. In this paper we have presented our view on such a system using .NET and a Parallel.For construct, which allows the users to easily convert their existing scientific programs into programs that utilize a distributed computer setup. Microsoft has already made support for using the Parallel.For construct on a single multi-core machine, but the system described in this paper extent that idea to utilize multiple machines. The implemented test cases show that for some simple scientific problems DistVES scales as well as Microsoft's solution; however, in some cases it does not. Altogether it is should be clear that a Parallel programmer's implementation of the tested problems at any time will scale better than the versions using DistVES or Microsofts Parallel Library; however, the two systems can be used by scientists that are not experts in Parallel programming and are having a simple scientific application that they want to parallelize. Therefore DistVES cannot be used to parallelize all types of programs, but for a subset e.g. simple scientific application, it will do fine. To improve DistVES a number of ideas, ranging from code annotation to rewriting the VES implementation in Mono in order to support distribution have been proposed as future work.

REFERENCES

[1] A. Geist, *et al*., "MPI-2: Extending the Message-Passing Interface," *Euro-Par*'96 *Parallel Processing*, Springer, Berlin/Heidelberg, 1996, pp. 128-135.

[2] M. R. B. Kristensen and B. Vinter, "Numerical Python for Scalable Architectures," *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model* (*PGAS*'10), New York, 12-15 October 2010, pp. 15:1-15:9. doi:10.1145/2020373.2020388

[3] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of ACM*, Vol. 21, No. 8, 1978, pp. 666-677. doi:10.1145/359576.359585

[4] Microsoft, "Parallel Programming in the .NET Framework," 2012. http://msdn.microsoft.com/en-us/library/dd460693

[5] OpenMP, "OpenMP," 2012. http://www.openmp.org

[6] J. P. Hoeflinger, "Extending OpenMP to Clusters," 2012. http://www.hearne.co.uk/attachments/OpenMP.pdf

[7] T. Seidmann, "Distributed Shared Memory Using the .NET Framework," 3*rd IEEE/ACM International Symposium on Cluster Computing and the Grid*, Tokyo, 12-15 May 2003, pp. 457-462. doi:10.1109/CCGRID.2003.1199401

[8] K. Asanovic, "The Landscape of Parallel Computing Research: A View from Berkeley," University of California, Berkeley, 2006.

[9] L. Lamport, "How to Make a Multiprocessor That Correctly Executes Multiprocess Programs," *IEEE Transactions on Computers*, Vol. C-28, No. 9, 1979, pp. 690-691. doi:10.1109/TC.1979.1675439

[10] S. V. Adve and H. D. Mark, "Weak Ordering—A New Definition," *Proceedings of the* 17*th Annual International Symposium on Computer Architecture* (*ISCA*'90), Seattle, 28-31 May 1990, pp. 2-14.

[11] M. S. Papamarcos and J. H. Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories," *Procedings of the* 11*th Anual Internation Symposium on Computer Architecture*, Ann Arbor Michigan, 5-7 June 1984, pp. 348-354.

[12] J. P. Hoeflinger, "Extending OpenMP to Clusters," 2012. http://www.hearne.co.uk/attachments/OpenMP.pdf

