

Transparent Offloading of Computationally Demanding Operations in Microsoft .NET

Morten Larsen¹ & Brian Vinter¹

¹ University of Copenhagen, Copenhagen, Denmark

Correspondence: Brian Vinter, University of Copenhagen, Copenhagen, Denmark. E-mail: vinter@nbi.ku.dk

Received: April 16, 2013 Accepted: May 7, 2013 Online Published: May 20, 2013

doi:10.5539/nct.v2n1p52

URL: <http://dx.doi.org/10.5539/nct.v2n1p52>

Abstract

For many years, the group of preferred programming languages for writing algorithms meant for large clusters contains among others C/C++ and FORTRAN. However, normally one does not consider the Microsoft .NET programming languages as a part of this group. The reason for this is that only few tools exist that can help programmers simplify the process of writing parallel .NET code besides the official tools from Microsoft i.e. Task Parallel Library (TPL) (Microsoft, n.d.) and HPC Pack. (Microsoft, n.d.) Furthermore, most of the official tools only supports a Microsoft Windows or Microsoft Azure platform and not a mixture of non-virtualized platforms like a Linux machine with Mono (Mono, n.d.) or the decommissioned DotGNU (GNU, n.d.). In addition, some of the most useful tools for writing parallel .NET code does not support multiple machines and as a result, programmers seldom choose .NET as the framework for writing parallel programs. Therefore, this paper presents a .NET tool, which will use well-known parallel tools as inspiration and allow programmers to call a number of methods that can send a job consisting of a user-defined method (code) along with sets of parameters and shared data to a central machine. The central machine will then modify the code and afterwards distributes the work to the connected machines each running one or more workers. By implementing three simple benchmarks, initial tests shows that the benchmarks can achieve linear scaling on a small cluster consisting of Windows machines, and by presenting future design ideas, it is believed that it will be possible to extent the linear scaling to a larger mix-platform cluster consisting of both internal resources (workstations/servers) and external cloud resources.

Keywords: Microsoft .NET, cloud, high performance computing, parallel programming

1. Introduction

Simplifying the process of writing algorithms, which have the ability to execute in parallel on large clusters, has been the goal of much research over the last couple of decades. Today, popular tools like MPI (Geist et al., 1996) and OpenMP (OpenMP, n.d.) are widely used by programmers wishing to implement algorithms that they can distribute among many machines. With the blossoming of cloud computing this trend has increased further as it is now possible for people, without resources to buy a large supercomputer, to use cloud instances to make a cheap cluster computer, which they only have to pay for when they actually do computations. The programming languages supported by many of these tools are the traditional ones such as C/C++, FORTRAN and for many years, these programming languages have been the preferred languages among scientists both in terms of speed but especially due to the great amount of well-tested and highly optimized libraries like Basic Linear Algebra Subroutine (BLAS) (BLAS, n.d.) and Linear Algebra PACKage (LAPACK). (LAPACK, n.d.) However, Microsoft has during the last couple of years introduced a number of tools like the Task Parallel Library (TPL) and the HPC Pack for .NET in order to simplify the process of writing code, which can run in parallel on the .NET platform and the Microsoft Azure Cloud system. The platform supported by the HPC Pack can consists of cloud instances (Azure), servers and workstations running some sort of Windows, whereas the TPL library currently only supports a single machine and programmers cannot use the tool for high performance computing. Thus, there is a gap between using the traditionally programming languages on a mixed platform and using the .NET programming languages on a platform of machines running some sort of Windows. Therefore, we propose a tool, which will allow running source code written using one of the .NET programming languages in parallel on a mixed platform. The goal of the tool is to provide programmers with a set of methods that allow them to run the computational intensive parts of a program in a Fork-Join (see Figure 1) style on multiple

machines.

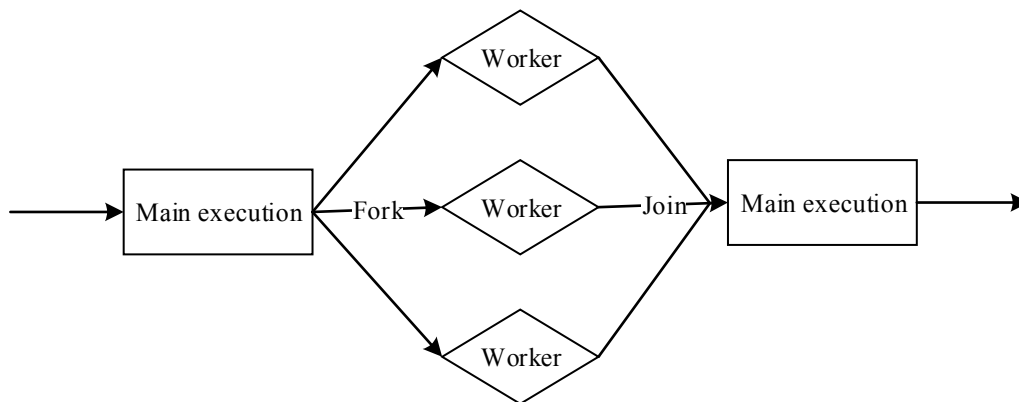


Figure 1. Illustrations of the Fork-Join principle

The rest of the paper is organised as follows: first, we give a description of the design followed by a number of benchmarks. Finally, the paper presents a section on proposed future work and a conclusion, which will summarize the findings.

1.1 Related Work

As mentioned above Microsoft is the main contributor to research with the .NET platform as the target. The TPL library exposes an API with a number of functions for writing both data and task parallel code. A number of loop constructs like `Parallel.For`, `Parallel.Foreach` makes it very simple for programmers to write data parallel code. Programmers can archive task parallelism by using the task construct and making arrays of tasks and then use a method to run the tasks in parallel. Both the data and task parallelism only support a single shared memory machine.

The HPC Pack from Microsoft contains, among other things, a MPI implementation for .NET. (Microsoft, n.d.) It allows running tasks on a cluster made of Windows machines including workstations, servers, and Azure cloud instances. Furthermore, it contains a service oriented architecture (SOA) scheduler, a scalable cluster management tools etc.

Other tools that support .NET exist, including a Distributed Shared Memory (DSM) system, which allows programmers to share objects among multiple shared memory machines using a software layer DSM model implemented using proxy objects (Seidmann, 2003). However, the success achievable in scaling this system to many machines is unknown, as the tool builds on top of the executing system (Common Language Runtime) of .NET.

2. Design

This section will describe the design of the proposed system including the definition of the API, which programmers can use to interact with the system. As described previously, the system is intended to support a mixed platform consisting of workstations, servers, and cloud instances all capable of running the intermediate assembler language of .NET named Common Intermediate Language (CIL) either using Mono or the .NET framework for Windows. The tool use the Fork-Join principle by which the programmer can run some code locally, then fork the execution in order to run computational intensive code on a number of machines. When the computational heavy part finishes, the system joins the results on the client side, the code can continue execution locally, and many tools like OpenMP use this well-known principle.

2.1 The Basic Idea

The TPL tool simplifies the process of writing algorithms that can fully utilize a multi-core machine, as the programmer can just use a `Parallel.For` or define a number of tasks that can execute in parallel. However, even though it is simple for the programmer to use, the tool does not encourage the programmer to consider how the program share data between tasks and therefore the risk of writing code containing race conditions is high. In contrast, the advantages of this approach is that it makes the scheduling easier for the TPL as there is no relation between tasks and that the TPL does not need to include a model for ensuring consistency of any shared data. In addition, fine-grained consistency on top of the .NET execution system (VES) is very expensive when the

programmer does not provide extra information regarding how the program accesses shared data. In addition, a previous attempt (Larsen & Vinter, 2012) to implement a consistency model in .NET turned out to be difficult, as one can only derive little information from the CIL code.

Therefore, another approach is needed and preferable one where programmers are strongly encouraged to consider how they access data and at the same time implicitly provide this information to the tool. A well-known paradigm, which does this, is Communicating Sequential Processes (CSP) defined by C.A.R. Hoare (1978) in the late seventies. CSP defines a program as a number of small processes connected to each other using directed channels. Shared data between two processes exists only in the sense that one processes can send data to another using the channel between them. This way of sharing data is very different from the shared memory approach, but it forces the programmer to consider how the program access data. In addition, it is easier for the programmer to eliminate race conditions and dead locks when each process has a size (tiny) which makes it possible to verify that no errors exist. Moreover, CSP contains logic which the enables programmers to formally prove algorithms written using CSP; however, as this will not be used in this tool it will not be further discussed (see (Hoare, 1978) for more information). Inspecting the practical part of CSP, a process can be thought of as a black box taking some input and based on this input returns one or more outputs. The black box access the input and the output by reading/writing to channels and therefore the data in the channels can be thought of as shared between the channels end-points. CSP defines multiple channel-types like One-2-One, Any-2-One, Any-2-Any (Vinter, Friberg, & Bjørndalen, 2009) where *any* means that multiple processes can be attached. If multiple processes connects to one end of a channel, the system defines numerous ways (round robin, random, prioritize) of deciding which process is granted access to the channel. Accessing a channel is in the basic case a blocking operation that the system will first execute when another process access the other end of the channel, meaning that data is first transferred when both writer and reader are ready to communicate. This makes it easy for the programmer to reason about a program as communication deals with both transferring data and acts as a synchronization point between the channels two end-points.

So how can this be transferred to .NET without adopting the whole CSP scheme of processes, channels etc., as we do not wish to force the programmer to write CSP programs, but just want to encourage the programmer to consider how she can access shared data. One possible solution is to disallow the programmer to run methods in parallel that access shared data and then use the input parameters and the return values to adopt the reading and writing of channels. This approach does not fully adopt the CSP way of thinking, as a .NET method can only have one return type. Additionally, the changes made to any reference-typed input parameter within the body of the method will also be available outside the method. Finally, the synchronization point defined when accessing CSP channels is non-existing in this approach. We can solve the first issue in many ways e.g. by making a special return type that could encapsulate multiple values like a tuple. The second problem is harder to solve, but we can do it by disallowing the programmer to change the input parameters unless the method also returns the parameters as the output of the method. However, controlling that the programmer does not violate this constrain is difficult as it requires a total recursive traversal of all accesses made to the parameters because a given parameter (an object) could be loaded and passed as an argument to another method that could then modify the parameter. The third problem of not having a well-defined synchronization point is in reality not a direct problem and if a hard synchronization point is needed it can be solved by dividing the method into two. Then the programmer can invoke the first method and use the result of the first method as input to the invocation of the second method like illustrated below:

```
result1 = Invoke(MethodOne, Paramaters);  
// Implicit synchronization point  
result2 = Invoke(MethedTwo, result1);
```

The problem of accessing shared data from multiple machines lies in how the system propagates changes made to a given pieces of data among all the machines having a copy of the given piece. This is especially difficult if the program uses a fine-grained access pattern i.e. accessing each element in an array. However, if the program does not make any changes to the shared data item, there is no need of special handling of these kind of accesses. As the intended use of this tool will lead to the system executing the same method multiple times with different input parameters, it seems sensible to allow read-only access of shared data. As with parameters, it is hard to detect if the programmer violates this constrain. However, as it is less intuitive that one cannot write to shared data (fields), we define that shared data items must be of the value-type or multi-dimensional arrays containing

value-typed elements. This constrain can easily be verified by ensuring that all fields is of the mentioned types and are only accessed using the “load field” CIL instructions named *ldfld* and *ldsfld*.

The last step is to consider the return type of a method, especially in the case where the programmer wishes to use it as an input parameter invoking another method. In this case, it is preferable that the data stays on the server-side of the application instead of transferring it back and forth between the client-side and the server-side. In combination with asynchronous invoked methods, this enables the client to disconnect from the server side and return later to check if the calculation has finished and eventually retrieve the result.

This finalizes the basic idea behind the tool and some of the initial design decisions and the next step is to define the component of the system, the workflow, and the API.

2.2 Design of Components, Workflow and API

So far we have mentioned two components; namely, a client and server-side component. The server-side will furthermore consists of two components; a monitor and a worker component (see Figure 2). The programmers will use the client to send jobs to the monitor, which will distribute the job (divided into tasks) to the connected machines each running one or more workers.

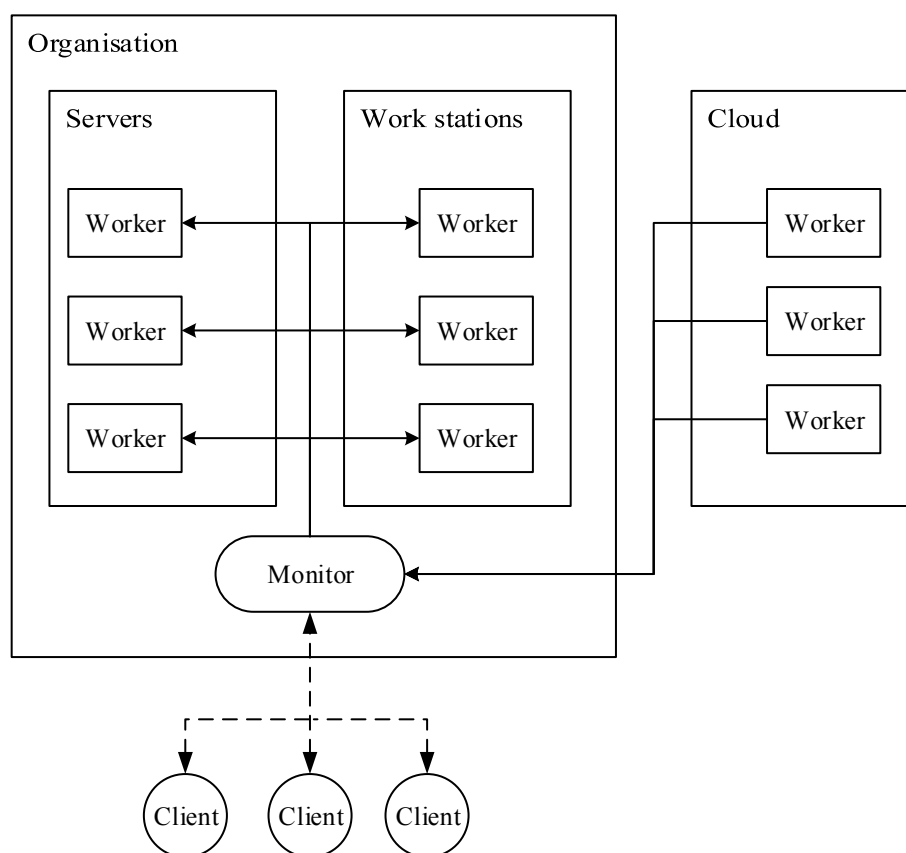


Figure 2. Overview of the components defining the tool

The monitor will be responsible for transforming a job consisting of a method, some parameters and possibly some shared fields into tasks, which the workers can execute. The first step is to control that the program does not violate the requirements mentioned above and determining if the program reads any shared data. If this is the case, the monitor must request the values (data) of the accessed fields from the client before it can proceed. After the monitor has received all the data, it will create a *ResultToken*, which the client can later use to access the result of the invocation.

Before the client invoke the system it will have created a class (instantiated object) to which the method to be invoked belongs. This class possibly contains many other methods and shared fields that are not accessed from the method, which the user wishes to invoke. However, as methods in .NET needs a target (instantiated object) unless they are static, all workers need the instantiated object mention above. Unfortunately, because the workers

does not share memory, the monitor must transfer the instantiated object over the network to all workers, which requires that the class is serializable. In order for a class to be serialized in the first place, it must have the *Serializable* attribute. This, along with the fact that the only code needed by the workers is the CIL instructions of the given method means that we chose another approach. This approach bases on defining a basic class, which the tool can modify at runtime to include the method along with the shared data (fields) accessed by the method. The constructor of this class named *Executor* takes as argument: a list of values, which the constructor uses to initiate the shared fields. The modified *Executor* class can now be distributed to all workers and they can instantiate the class which will make it possible to call the method.

As the parallelization of a method in the proposed system lies in the parameters meaning that the system will execute the same method multiple times using different parameters, we need a methodology to define these sets of parameters. We decide to define the parameters using a list of values for each parameter along with information about how the monitor should mix these lists together to make a single list of parameter-sets. The three ways of adding a new parameter are defined as “element wise”, “single element”, and “zip”. These are used to define how a “new” parameter is added to the already constructed list of parameter-sets. The values (a list) of each parameter is in turns added to the initial empty list of parameter sets. Examples of the resulting parameter-sets for the addition of the parameter consisting of the values (a, b, c) to the existing parameter list (1, 2, 3) are shown below:

Element wise: ([1,a],[1,b],[1,c],[2,a],[2,b],[2,c],[3,a],[3,b],[3,c])

Single element: ([1,(a,b,c)],[2,(a,b,c)],[3,(a,b,c)])

Zip: (requires same length): ([1,a],[2,b],[3,c])

As each method will use a single parameter-set as input and generate a single output (result), the outcome of using all parameter-sets is a single list of outputs. Each worker could send the result back to the monitor, which would later return the results to the client, but if the program later use the results as input for another method this is a waste of resources. An alternative approach is to let the workers keep the results locally and only send them to the monitor if requested by the program. This approach will minimize the amount of data transfers, but naturally increase the waiting time when the program requests the results. In addition, this approach requires that the monitor knows where the results are located. This problem will be touched upon below when discussing the scheduler.

The next step is to define the API. Firstly, the programmers need methods to invoke the tool. Such an invocation will return a *ResultToken*, which the programmers can use to fetch the results from the monitor:

```
class Client
{
    ResultToken = Invoke(ID, Method, List of selectors, Parameters, List of ResultTokens)
}

class ResultToken
{
    bool = IsDone()
    List = GetResult()
    Delete()
}
```

The implementation should use generics on both the *ResultToken* and definitely on the *Invoke* method to ensure that the types of the parameters and/or *ResultTokens* used as input in combination with the selectors matches the types of the input parameters of the method which the user wishes to invoke. Likewise, the output type of the method should match the type of the resulting *ResultToken*. This will help the programmer from making

mistakes.

The performance in a system like the proposed depends strongly on how well the monitor will distribute tasks to the connected workers. Many methods exist ranging from a simple centralized round-robin distribution to distributed and auto-balanced algorithms like work-stealing (Blumofe & Leiserson, 1999). As we established earlier, workers will not transfer results back to the monitor every time they finish a task, a simple round-robin distribution of tasks will naturally result in an increase of data transfers if the tasks being distributed use results from previously tasks as input. In such situations it is preferable if the monitor is capable of sending a given task to the worker that holds the results needed in order for the task to execute. For the monitor to be able to do this it needs two pieces of information; namely, which worker executes a given task and the size (in bytes) of any given result. One could argue that both pieces of information in most cases would be present at the monitor-side when it schedules the tasks because the monitor knows which worker will receive a given task and if the result of execution is one of the primitive types in .NET (int, float, double etc.), the monitor also knows the size of the result. However, if the return type is an object or an array, the size will be unknown at the time of scheduling. Therefore, and as the overhead of sending a single packet for each executed task is small, we decide that the worker would report which tasks they have solved including the size of the result. Now the monitor will be able to use the information to schedule the tasks based on where the input parameters to a method are located. Instead of doing this scheduling centrally, an alternative solution is to let the workers request tasks from the monitor and then let the workers inspect the input information. If most of the data is remotely located, the worker could forward the tasks to the worker having most of the data. This will partly distribute the scheduling, but at the risk of the scheduling becoming unbalanced. The "Future work" section of this paper will discuss a possible solution for solving this. As there will be a small overhead for the monitor to process each task-request from a worker, the workers should be able to request tasks asynchronously, so that the system will be able to hide most of the overhead of communication and scheduling.

3. Results

To test the implementation of the design, we implemented three simple benchmarks. Afterwards we executed the benchmarks on a small cluster consisting of four machines connected through a gigabit network. Each machine was equipped with a quad-core Intel i7 processor with HyperThreading running at 2.8 GHz and a minimum of 8 GB of memory at 1.333 MHz. We repeated the execution of each benchmark four times using one to sixteen workers. The workers were distributed among the machines in two ways, namely by round robin or by filling one machine (max four workers per machine) at a time. The three benchmarks chosen were a prototein folding, Black-Scholes and a K-nearest-neighbours (kNN).

3.1 Prototein

Prototein folding (Hayes, 1998) is a simplified two-dimensional version of the well-known Protein folding, using only two amino acids (H and P). Furthermore, the amino acids are limited to only having four neighbours in 90-degree angles to one another. Given a string of amino acids, the goal of the algorithm is to minimize the number of H amino acids in the string not having four neighbours. The algorithm is data parallel implemented using a bag-of-task approach and overall we expect the implementation to scale linearly.

Figure 3 illustrates the result of running the benchmark. In the test where the system distributed the workers by filling one machine at a time, we see that the scaling is linear and when using one and two workers the gradient is close to the optimal. As all four cores share the same memory bus, the bus cannot get data fast enough from the memory to the computational cores and we expect this to be the reason for the decrease of the gradient when using more than two workers per machine. When distributing workers using the round-robin model, the same is seen, but as workers are equally distributed between machines, the gradient is closer to the optimal until eight workers (two per machine), and then decreases. Thus, if one uses more than two workers per machine the additional workers (cores) will only contribute with approximately half of their potential. Overall, the result is satisfying with a 75% utilization when using four test machines. The workload on the machine running the central component is a bit higher, but nothing indicates that it cannot handle having even more connected machines.

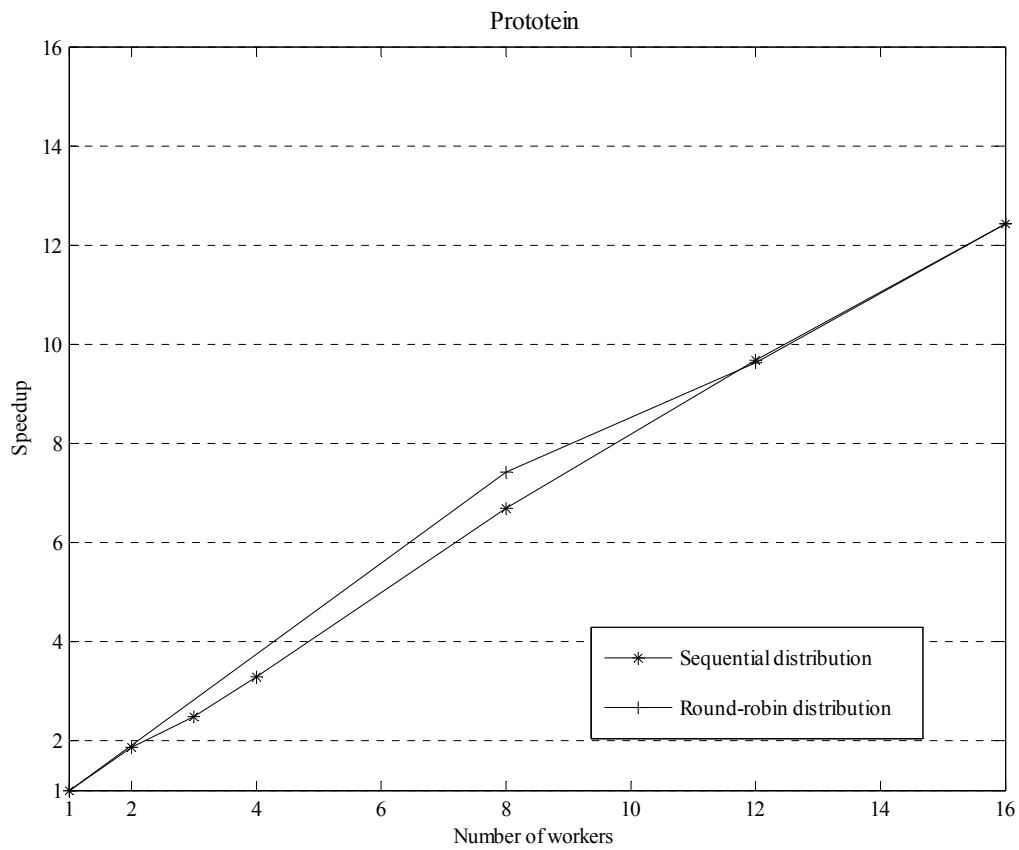


Figure 3. The result of running the Prototein benchmark

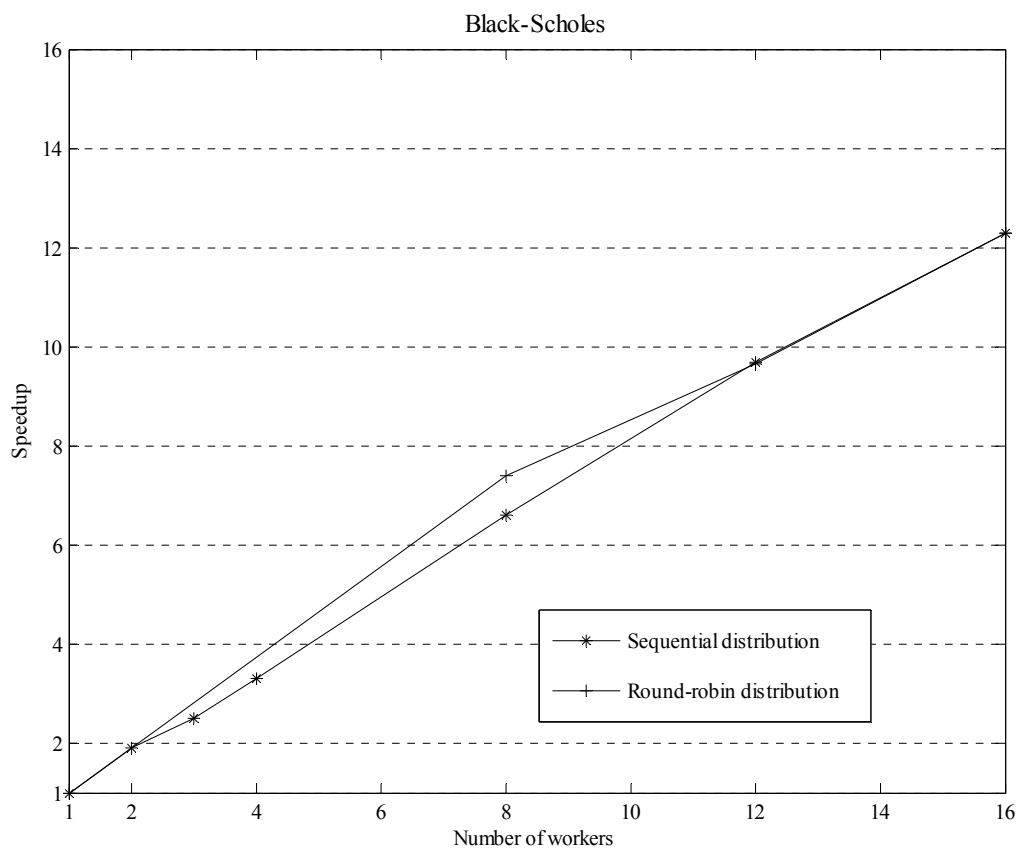


Figure 4. The result of running the Black-Scholes benchmark

3.2 Black-Scholes

The Black-Scholes is a Monte Carlo simulation used for pricing European-styled bonds and it is embarrassingly parallel and should scale close to linearly. The result of the execution is shown in Figure 4 and when comparing the result to the one given in Figure 4, one can see that the two graphs are almost identical, so again the problem scales linearly, but with a lower gradient when using more than two workers per machine due to the memory wall. Furthermore, when using eighth workers (round-robin distribution) the utilization is 92.5 %, meaning that tool generates some overhead mainly which overlapped execution cannot hide. An alternative reason for this overhead is that it is directly related to doing task distribution.

3.3 kNN

The final benchmark is the well-known k-nearest-neighbours, kNN problem. Given a number of datum in an N-dimensioned space, the goal of the algorithm is to find the k nearest (by distance) neighbours for all datum. The algorithm is very simple, but it requires a large amount of datum meaning an increase in the amount of transferred data. Furthermore, given a single particle the calculation needed in order to calculate the distances to all other datum is very small, and therefore each task should calculate the distances between multiple datum in order to increase the calculation/communication ratio. With a sensible task size, we expect the algorithm to scale linearly possibly with a lower gradient compared to the two previous benchmarks due to increased amount of data transferred.

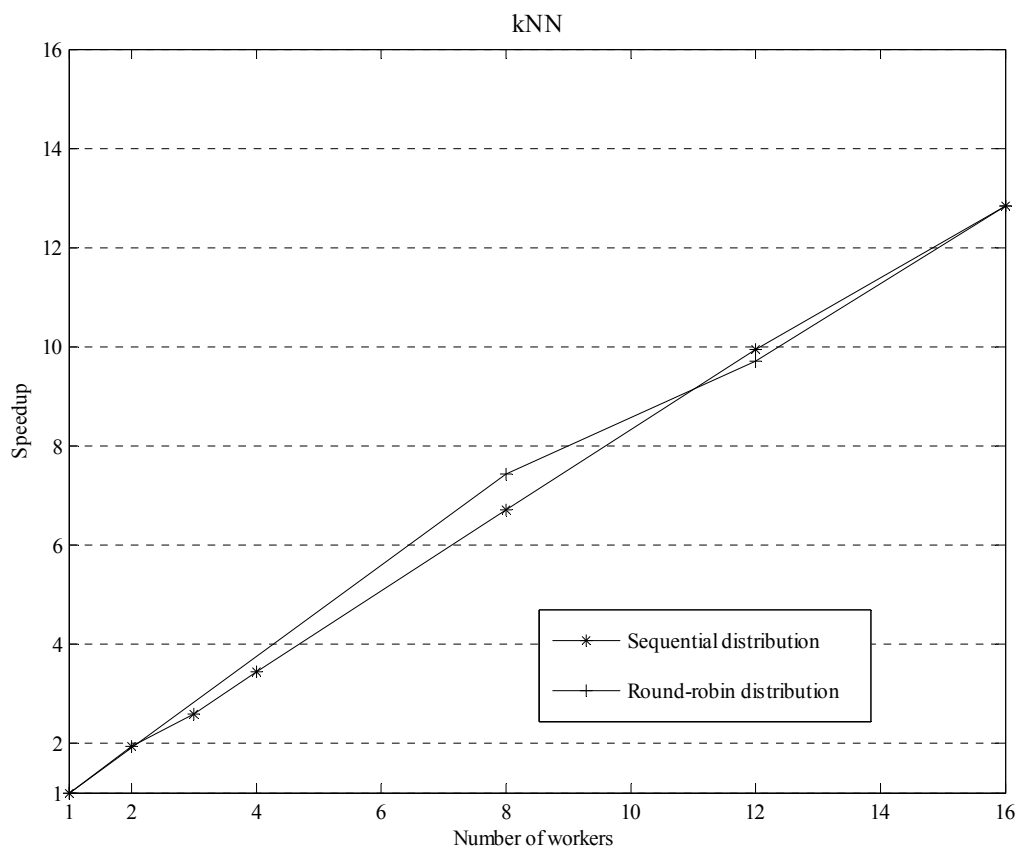


Figure 5. The result of running the kNN benchmark

Figure 5 shows, as the figures of the previously two benchmarks, that the problem once again become memory bound. However, the gradient is the same as the gradient seen in the graphs for the previous two benchmarks, which means that the system is able to hide some of the overhead related to data transfers because of the use of overlapped execution. This is very essential is the tool is used for algorithms where data transfers of a certain size are required.

4. Future Work

As the proposed scheduling method can lead to an unbalanced distribution, the system needs to use a more advanced method for distributing tasks. One possibility is to use work stealing where each worker has a double-ended queue with tasks prioritized based on the amount of data that is locally present. Alternatively, a more traditional approach could be used where the monitor does the initial distribution by prioritizing the tasks and then let the worker use a work-stealing implementation based on a normal double-ended queue.

Another issue concerns security when using cloud instances. With a platform consistency of both internal and external machines it is preferable that programmers can mark, certain data with a special attribute which specifies that the system must not transfer the data to the external cloud instances. Likewise, the programmer should also be able to specify if external resources can share jobs and/or return values. In order to do this, the monitor must be able to distinguish between workers running on the internal and external machines. We propose to solve this by including a management module, which also allows handling starting/shutdown cloud instances, servers, and workstations.

5. Conclusion

This paper presents a simple tool, which allows the parallel execution of code written using one of the Microsoft .NET programming languages on a mixed platform consisting of machines running Windows and/or Linux. The tool exposes a small API along with some requirements to the method, which the programmer wishes to execute in parallel. The tool analyses and modifies the code, before the monitor converts the provided parameters into a number of tasks and then distributes the tasks to the connected workers. Initial tests indicate that for the benchmarks linear scaling can be achieved, but all benchmarks show that if using more than two workers per machine the gradient of the scaling decreases as the memory wall is hit.

Acknowledgements

The authors would like to thank “The Innovation Consortium” for supporting this research with grant 09-052139.

References

- BLAS. (n.d.). *BLAS (Basic Linear Algebra Subprograms)*. Retrieved April 11, 2013, from <http://www.netlib.org/blas/>
- Blumofe, R. D., & Leiserson, C. E. (1999). Scheduling multithreaded computations by work stealing. *Journal of the ACM*, 46(5), 720-748. <http://dx.doi.org/10.1145/324133.324234>
- Geist, A., Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Saphir, W., ... Snir, M. (1996). MPI-2: Extending the message-passing interface. In *Euro-Par'96 Parallel Processing* (Vol. 1123, pp. 128-135). Berlin: Springer Berlin Heidelberg. http://dx.doi.org/10.1007/3-540-61626-8_16
- GNU. (n.d.). *DotGNU Project* (GNU). Retrieved April 11, 2013, from <http://www.gnu.org/software/dotgnu/>
- Hayes, B. (1998). Computing Science: Prototeins. *American Scientist*, 86(3), 216-221. Retrieved from <http://www.jstor.org/stable/27857018>
- Hoare, C. (1978). Communicating sequential processes. In *Commun. ACM* 21 (pp. 666-677). <http://dx.doi.org/10.1145/359576.359585>
- LAPACK. (n.d.). *LAPACK—Linear Algebra PACKage*. Retrieved April 11, 2013, from <http://www.netlib.org/lapack/>
- Larsen, M., & Vinter, B. (2012). A Distributed Virtual Machine for Microsoft .NET. *Journal of Software Engineering and Applications*, 12, 1023-1030. <http://dx.doi.org/10.4236/jsea.2012.512119>
- Microsoft. (n.d.). *Microsoft High Performance Computing for Developers*. (Microsoft.com) Retrieved April 11, 2013, from <http://msdn.microsoft.com/en-us/library/ff976568.aspx>
- Microsoft. (n.d.). *Microsoft MPI*. Retrieved April 11, 2013, from Microsoft MPI
- Microsoft. (n.d.). *Parallel Programming in the .NET Framework*. Retrieved April 11, 2013, from Microsoft.com: <http://msdn.microsoft.com/en-us/library/dd460693.aspx>
- Mono. (n.d.). *Mono project*. (Mono) Retrieved April 11, 2013, from <http://www.mono-project.com/Main Page>
- OpenMP. (n.d.). *OpenMP.org*. Retrieved April 11, 2013, from <http://openmp.org/wp/>
- Seidmann, T. (2003). Distributed shared memory using the .NET framework. In *CCGrid 2003. 3rd IEEE/ACM*

International Symposium on Cluster Computing and the Grid (pp. 457-462). IEEE.
<http://dx.doi.org/10.1109/CCGRID.2003.1199401>

Vinter, B., Friborg, R. M., & Bjørndalen, J. M. (2009). PyCSP Revisited. *Communicating Process Architectures*, 67, 263-276.