.NET Task Parallelization in the Cloud

Runtime Support for Seamless Distribution of Shared Memory Parallel Tasks

Luc Bläser

HSR University of Applied Sciences Rapperswil
Institute for Software
lblaeser@hsr.ch

Abstract

The native parallelization support in predominant programming languages focuses on local cores, while mostly neglecting the huge parallelization potential of distributed execution. We have therefore developed a .NET runtime system extension that automatically distributes ordinary shared memory parallel tasks into the cloud, to execute them on a large number of cores, e.g. on a cluster. The runtime mechanism takes care of transmitting the necessary task code and data to the service, as well as of propagating task results and side-effect changes back to the client memory. For programs with long-running tasks or a high amount of tasks, the system is able to soon compensate the network transmission overheads and thereafter scale with the number of tasks up to the available server-side cores.

Categories and Subject Descriptors Software notations and tools [*Compilers*]: Runtime environments

Keywords Task parallelism; cloud; distributed shared memory system; runtime system design; .NET; cluster

1. Introduction

Today's mainstream programming languages principally target only on local multi cores with their institutionalized parallelization features, as it fits well to their underlying shared memory model. However, moving from local to distributed parallelization, imposes the substantial extra burden for the developers to leave their conventional programming model and close the gap to distribution on their own, e.g. by redesigning the programs for specific distribution frameworks (e.g. service, remoting, or grid computing architectures) or by engaging a different more distribution-friendly programming paradigm (e.g. descriptive dataflow models).

It is therefore of no surprise that parallelization in daily programming practice largely concentrates on utilizing local processors, while mostly neglecting distributed parallelization - unless there is a strong performance urge justifying the efforts of a corresponding dedicated solution. Another strong distribution obstacle is the fact that developers and users do normally not have remote processor power at hands, although many clusters and distributed systems would have free capacities – but unfortunately without an easy-to-use "parallelization-as-a-service" interface.

There has been intensive research done in this area, principally going into three main directions: (1) new/different programming models inherently suited for distribution, such as Actors/MPI [1, 11], or dataflow/query models [8, 22], (2) distributed task/thread frameworks, e.g. in the grid computing area [12, 17, 20], and (3), distributed shared memory systems [2, 10, 14, 16, 24]. While the first direction certainly takes the more radical and sustainable approach of tackling the distribution impedance already at its roots, it usually compels programmers to apply the different paradigm on top of or aside their ordinary imperative shared memory programming language. The second direction, represented by the many existing grid computing or distributed thread/task frameworks, typically leads to visible seams in the program design: it necessitates explicit task and data offloading, marking data serializable, wrapping data in specific sub-classes etc., which is far away from closing the semantic gap between the problem-space and the machine-space parallelism. The third direction, distributed shared memory systems, transparently enables distribution of normal shared memory programs across machines. Although, it is generally heavyweight for this purpose, operating on the whole program rather than selectively on the distributed parallel algorithmic parts. A more detailed survey of related works is given in Section 5.

Our research goal is to significantly ease distributed parallelization by providing a new runtime system that allows scaling up parallel programs *seamlessly* on massive processor power in the cloud, i.e. with the same programming model as for local parallelization and without requiring any explicit code and data transmissions. For this purpose, we propose an enhanced thread pool mechanism that transparently integrates remote multi-processing: Although described like conventional local parallel tasks operating on shared memory, the runtime system automatically distributes the tasks to a web service into the cloud. Behind the service, the tasks are to be executed on a large number of cores before their results and their potential effected memory changes are finally sent back to the client runtime. The web service is of our design and can abstract an arbitrary parallel processing infrastructure behind the interface, e.g. a high-performance computer cluster. We have realized this system for the .NET framework, to demonstrate the concept by the example of a popular shared memory programming platform.

This paper makes the following contributions:

- We have developed distributed task parallelization as a model to seamlessly utilize remote multi-processor power in mainstream programming, such as in .NET.
- We have implemented a new runtime system for distributed task parallelization in .NET, and have experimentally validated its practicability and benefit.

CONF 'yy, Month d–d, 20yy, City, ST, Country.

http://dx.doi.org/10.1145/nnnnnnn.nnnnnnn

```
var distribution = new Distribution(ServiceURL, AccessCode);
var taskList = new List<DistributedTask<long>>();
foreach (var number in inputs) {
  var task = DistributedTask.New(
    () => Factorize(number)
  );
  taskList.Add(task);
}
distribution.Start(taskList);
foreach (var task in taskList) {
  Console.WriteLine(task.Result);
}
```

```
long Factorize(long number) {
 for (long k = 2; k * k <= number; k++) {
   if (number % k == 0) { return k; }
 }
 return number;
}
```

Figure 1. Distributed parallel tasks factorizing numbers.

The remainder of this paper is structured as follows: Section 2 describes the programming model of distributed parallel tasks. Section 3 explains the design and implementation of the runtime system. Section 4 presents performance and scalability results. Section 5 compares distributed task parallelization to related work. Section 6 finally draws a conclusion of this work.

2. Programming Model

Cloud task parallelization encourages programmers to implement and start distributed parallel tasks that can be dispatched and executed on remote processors.

2.1 Distributed Tasks

In our system, which is based on .NET, distributed tasks can be programmed like conventional local thread pool tasks offered by the .NET task parallel library [15]. A distributed task is implemented as an ordinary .NET delegate[1] or lambda[2]. In principle, working with distributed tasks remains analogous to using local parallel tasks, i.e. they can be instantiated, started, and joined. Certain restrictions apply for distributed tasks: Inner synchronization and calls to IO are for example forbidden. A detailed explanation of restrictions is provided in subsequent sections.

Figure 1 shows a code example for factorizing a set of numbers in parallel, each number being factorized as a separate distributed task. No extra compilation step is involved here; adding a reference to the library of our cloud task parallelization is sufficient for the runtime mechanism. The code sample looks very similar to the local task parallelization, as depicted in Figure 2. The URL and access authorization code need to be specified in advance for the remote task parallelization service, before a set of tasks can be started. Accessing task results blocks as long as the corresponding task is not terminated, where task faults are propagated as exceptions. We deliberately did not unify the local and distributed task class because we would like to encourage explicit combined starts of multiple distributed tasks for reducing network roundtrips, whereas the existing local task class promotes starting one-by-one.

For increased convenience, distributed tasks can also be applied in the form of data parallelism (Figure 3), by using parallel invocations or parallel loops. As for a parallel loop, each loop step starts a distributed task that executes the body and is joined again at the loop end.

[1] A .NET delegate is a reference to a method and an associated object.

[2] A .NET lambda is an anonymous delegate in the form of an inline statement or expression with access (closure) to variables of its lexical scope.

```
var taskList = new List<Task<long>>();
foreach (var number in inputs) {
  var task = Task.Factory.StartNew(
     () => Factorize(number)
  );
  taskList.Add(task);
}
foreach (var task in taskList) {
  Console.WriteLine(task.Result);
}
```

Figure 2. Analogous solution with local parallel tasks.

```
distribution.ParallelFor(0, inputs.Length, (i) => {
  outputs[i] = Factorize(inputs[i]);
});
```

Figure 3. Distributed data parallelism.

Alike local tasks, distributed tasks are allowed to also perform side-effect changes on disjoint memory locations in shared memory, as also illustrated in Figure 3. The modifications in the array `outputs` of the example become automatically visible after task completion. For this purpose, the runtime system collects side-effect changes of tasks at the server side and propagates them back to the client-side memory. The system detects certain data races, as described in the next section.

2.2 Task Isolation

Distributed tasks are required to be independent of all other active tasks and threads, i.e. read-only accesses on shared variables and arbitrary accesses on non-shared variables are allowed. The granularity of variable accesses is per field or array element. Notably, this does not constitute a strong limitation because for local task code, synchronization in task execution is usually also avoided for highest possible performance. This applies for both synchronization primitives and memory model atomicity/visibility. The demanded task isolation eases the distribution significantly, since it excludes information flow between active distributed tasks, as well as, between active distributed tasks and the remaining program code. Our system detects certain violations of task isolation, namely when tasks employ synchronization, or when write-write conflicts happen due to data races. Read-write conflicts are not detected though: The reading task will not see the change of another concurrent task. In contrast, data races in local concurrency are not detected at all, meaning that our system provides somewhat more runtime guards.

2.3 Security Concerns

The runtime system prevents distributed tasks from directly or indirectly executing IO operations, system calls, reflection or unsafe/unmanaged .NET code. IO and system calls are not allowed because we do not delegate these calls back to the client, such that they would otherwise become effective on the remote machines. If reflection and unmanaged code would not be forbidden, programmers could accidentally or intentionally inspect or modify arbitrary program state or corrupt memory safety at the remote side.

3. Runtime System

The system for distributed task parallelization consists of three components: a client runtime library, the cloud processing web service, and a server runtime library.

3.1 Processing Roundtrip

The processing of distributed tasks involves the following steps, as illustrated in Figure 4: (1) The potentially executed program

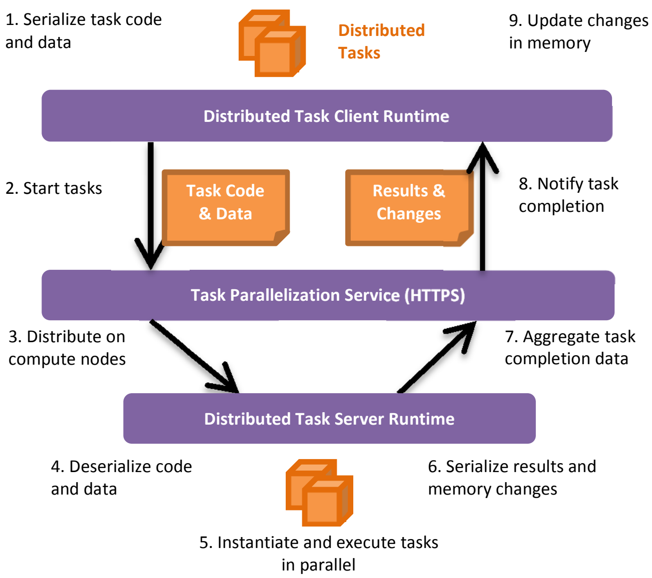

Figure 4. Distributed task runtime system.

code and accessible data of the invoked tasks are collected and serialized by the client side library at runtime. (2) The serialized code and data are shipped to our web service which represents the cloud processor resources. (3) The service distributes the tasks on server-side compute nodes, currently by launching a HPC cluster job consisting of a HPC task per input .NET task, i.e. by using the default task scheduling of the cluster. (4) The code and data are deserialized and instantiated by the server runtime library on each server compute node. (5) The remote tasks are executed on the compute nodes. (6) When terminated, the results and modified data of tasks are collected and serialized by the server runtime library. (7) The serialized data of task completion is sent back to the initiating client over the web service. (8) The updates are finally made effective in local memory of the client.

3.2 Task Serialization

When tasks are started for distribution, the client runtime component serializes the necessary code and data in two phases by way of reflection.

In a first phase, a conservative context-insensitive code analysis determines all reachable program code. Starting from the task delegate, the transitive closure of potentially directly or indirectly invoked methods is calculated. Additionally, it records all potentially used classes and accessed fields within the reachable methods. The code of each visited method is examined to only contain supported instructions and calls according to the security constraints (the system triggers a runtime exception if the code cannot be distributed). The set of reachable methods and their intermediate language code is eventually serialized.

In a second phase, all potentially accessed task data is collected. For this purpose, the system generates a partial heap snapshot, being the graph of objects that are reachable via references from the task delegate, by only considering the references occurring in potentially accessed fields according to the preceding code analysis. For the collected objects, only the state of accessible fields needs to be serialized. Besides the object instances, the snapshot also includes static fields and constants that can be used by tasks. Because of the required task isolation, the runtime serialization delivers a consistent state without need of synchronization, i.e. the system never blocks other running threads. Due to the conservative analysis, the snapshot may, however, include data that is not effectively accessed by the tasks and therefore also not required to be isolated: the state of this data may be inconsistent though but it is also never accessed by the distributed tasks.

3.3 Task Results

The server runtime component returns all necessary information of completed tasks, such as the task delegate result value, modifications on transmitted objects and static data (updates of fields and array elements), as well as, all reachable new objects that have been created by the remote task execution. The client in turn performs the in-place updates on arrival of the task completion information, i.e. modifications are applied to the corresponding objects and static fields of its input snapshot. We perform change detection by comparing the field and array element state before and after task execution. With this approach, the client runtime also detects certain data races, namely illegal write-write conflicts across distributed tasks.

3.4 Service Design

Task code (program metadata and intermediate language code) and data (object graph and static fields) are encoded in an own binary format to reduce client-to-service traffic as much as possible. The service functionality basically comprises two operations, one for starting a set of tasks and another for awaiting the termination of a set of tasks. To reduce network roundtrips, multiple tasks can be sent in one bunch, where the task instances can also share the same task code. To support secured network transmission, a HTTPS service binding can be used.

3.5 Limitations

Our system supports the essential language feature set for implementing algorithmic tasks (arrays, variables, control statements, elementary types, methods, objects). However, the present version also has some implementation restrictions: nested task starts are not supported, as this would involve rebalancing of distributed tasks across compute nodes in the backend. Moreover, some specific language features such as type polymorphism (inheritance, interfaces, delegates, exception catching), struct-types, ref/out parameters are currently not yet implemented within tasks - although there is no conceptual reason against it. The system detects unsupported features in distributed tasks and reports such by an exception.

4. Experimental Results

Distributed task parallelization is intended for running computing-intensive tasks and/or a large amount of tasks, offering a high potential of parallelization.

4.1 Measurement Setup

For an experimental evaluation of our current system version, a set of synthetic parallel problems have been implemented on the basis of distributed tasks and eventually run in an environment with a MS HPC computer cluster behind the cloud service. The cluster comprises 32 nodes with 12 Intel Xeon cores, 2.6 GHz each (of which we were allowed to use 100 cores for our experimental study). The client and web service each run on an Intel 2 Core, 2.9 GHz machine, with 100Mbit/sec bandwidth and 1ms network delay between client, service and the cluster. All measurements have been performed by using compiler-optimized 64-bit .NET program assemblies. For all runtime results, the minimum of three repeated runtimes is considered, to reduce negative influences of temporary network speed fluctuations.

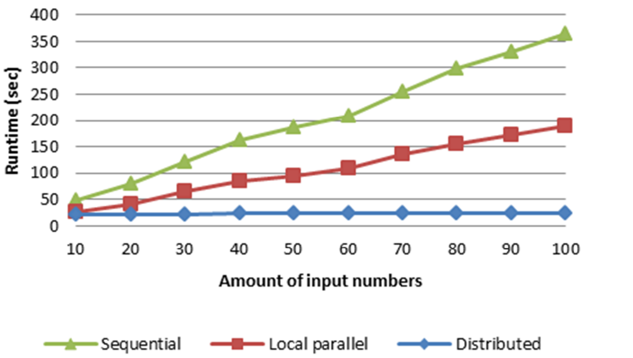

Figure 5. Performance scaling by number of tasks.

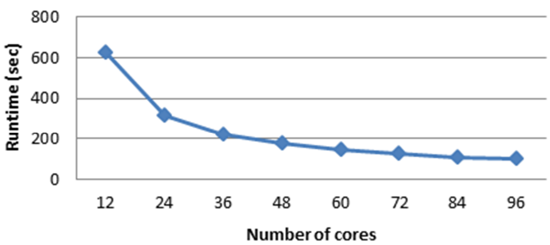

Figure 6. Performance scaling by number of cores.

4.2 Performance Scalability

To study the performance scalability, we measure the runtime for a set of independent computation tasks. To start with a first scenario, we compute the factorization of a set of sample numbers, each composed of two larger prime factors. Each number is factorized independently in a parallel task. The comparison involves three processing approaches: (1) with distributed tasks, (2) with local tasks, and (3) sequential execution. Figure 5 shows the runtime in seconds depending on the amount of input numbers, which is equal to the number of parallel tasks. As expected, the parallel speedup of distributed tasks scales linearly with the number of tasks – in this scenario, each task runs on a separate instance of the 100 available cores. Naturally, local parallelization only offers a speedup of 2 on the two core client machine. Of course, the speedup also depends on the number of free cores available in the cluster. Figure 6 shows the necessary runtime in seconds for factorizing 100 numbers, by limiting the number of available cores in the cluster.

4.3 Cost Breakdown

The runtimes for distributed task parallelization involve different performance cost factors, which vary from problem to problem: (1) the effective task execution time on the server side, (2) the network transfer time from the client over the service to the cluster, (3) the cluster dispatching costs, and (4) the accumulated effective overheads of our runtime mechanism, that is task serialization, deserialization, and change/result propagation. Table 1 depicts the breakdown of runtime costs in seconds for the factorization of 10 numbers, i.e. for 10 tasks. In this scenario, task execution represents the most significant part: this is the time where tasks are executed in parallel on the cluster. Network transfer constitutes the second most substantial portion. The remaining cost factors, including our own runtime mechanism, are relatively small.

4.4 Performance Comparisons

For a more general performance comparison, we evaluate the runtimes for different problem cases: (1) Mandelbrot fractal computation for a specified image size, as a representative of a parallel problem with a relatively high data amount compared to the task computation time. (2) Knight tours computation on a chess board of a specified size, as a representative for relatively long-running task computations. (3) Primes scanner counting primes in a specified number range, as a representative of relatively short-running tasks. Table 2 shows the runtimes in seconds, rounded to two significant figures, for the specified instances of these problems. We again compare distributed task parallelization (100 cores), local task parallelization (2 cores) and sequential execution. Once more, significant performance improvements can be achieved with the runtime support of distributed tasks.

Runtime costs (seconds)	Factorization (10 numbers)
Node execution	**18.4**
Network transfer	1.9
Cluster dispatching	0.3
Task serialization	0.3

Table 1. Runtime costs breakdown.

Runtime (seconds)	Mandelbrot (10000 x 1000)	Primes Scanner (range 10^7)	Knight Tours (6 x 6 board)
Distributed	**8.0**	**4.7**	**120**
Local parallel	20	9.2	1100
Sequential	37	19	2200

Table 2. Performance comparison of parallel problems.

4.5 Result Discussion

As expected, the examples confirm that the runtime system is able to reach a high parallel speedup by the large amount of remote cores. However, the gain of parallelization needs to compensate the involved overheads, which are primarily the network transmission time, depending on the size of task serialization, the data bandwidth and network delay. Distributed task parallelization is therefore generally beneficial if a large amount of tasks is executed, tasks are running sufficiently long, or tasks entail relatively low data transfer.

5. Related Work

5.1 Distributed Data Parallelism

Microsoft DryadLINQ [22], built on the distributed runtime engine Dryad [13], has close relation to our system: It permits automatic distributed processing of .NET LINQ [19] queries on clusters. While this model is oriented on descriptive programming of distributed dataflows in terms of queries, our system promotes more imperative task or data parallelization. Therefore, our system also allows distributed tasks to perform side-effects or changes that are propagated to the client, while the only backflow in DryadLINQ evaluations are the query results. Side effects of delegates inside the queries of DryadLINQ are ignored.

MapReduce [8] and in particular, also the Hadoop MapReduce implementation[3], are popular dataflow programming models for high-scale distributed parallelization. The integration in a client program is, however, less seamless than in our model: Data is to be explicitly passed to the map and reduce functions from files or serializeable key-value sets. This is different to the shared memory

[3] `http://hadoop.apache.org`

illusion of our model, where the data of distributed tasks is automatically transmitted. MapReduce does also not directly incorporate a cloud approach where clients can easily offload their dataflows to a service. Though, such architecture can be designed around. Similarly, FlumeJava [5] and Cascading[4] realize distributed data parallelism in Java on top of Hadoop. In both systems, custom parallel processing operations cannot use shared memory.

Other grid computing systems such as Pegasus[5] and Swift [23] also facilitate DAG-like task workflow distribution on cloud computing resources but again with explicitly programmed data and task transmission. CIEL [20] supports powerful parallel task workflows with dynamic task spawning implemented in a specific language (Skywriting). Tasks can trigger batch commands, or invoke Java/.NET code in a less transparent way than our system: by denoting the class name and passing arguments and results.

5.2 Distributed Task Parallelism

Existing distributed thread/task programming frameworks, such as JPPF[6], Hadoop, ProActive Parallel Suite[7] [12], the already mentioned CIEL Skywriting [20], Alchemi [17], Manjrasoft Aneka .NET Tasks[8], and many others, make distribution significantly more visible than in our system: heap data from the client program is not automatically shared across distributed processes but must be passed as explicitly serializeable objects within task parameters and results, or has to be managed in specific grid heaps or distributed data collections. However, the focus of our system is on enabling mostly seamless and convenient task parallelization on remote processor resources.

5.3 Message Passing Models

The Actor model [1, 11] facilitates inherent distribution of active instances (actors) across machines, because actors only interact via explicit message communication and do not share memory. This can even be applied on top of mainstream programming languages, by frameworks or libraries such as Akka[9], MPI [10], ProActive parallel suite [3], and many more. If applied within a conventional shared memory language, this indispensably provokes a semantic gap, since programmers need to think in a different paradigm than the native language and have to stick to particular conventions. For example, actor communication must not be bypassed by ordinary references.

5.4 Distributed Shared Memory

Various systems have realized virtual shared memory on distributed computers, be it at the operating system level [10, 16] or at the runtime system of a programming language [2, 14, 24]. While this can establish automatic distribution of an entire program, our system employs distribution only selectively for task parallelization. Moreover, we provide the distribution as a service for use by a possibly open group of clients.

5.5 Modular Distribution

Transparent distribution can be enabled for modular systems, by configuring a flexible deployment of the modules across machines and letting the runtime system transmit the inter-module calls. The R-OSGi [21] middleware distributes the modules of a Java program (based on the OSGi framework) across systems, by automatically replacing method calls by remote invocations. While it allows flexible and mostly seamless distribution, it is not designed for enabling massive remote parallelization, e.g. on a cluster.

5.6 Offloading in the Cloud

Dynamic code offloading to the cloud gains increasing popularity in the research area of mobile computing [6, 7, 9]. MAUI [7] is a .NET-implementation, in which distributable methods need to be explicitly marked by an attribute. CloneCloud [6] goes further and employs static analysis and dynamic profiling for code partitioning instead of explicit information. The primary motivation of this research area is to reduce execution and power on mobile devices and not necessarily to increase parallel speedups. Stack-on-demand execution [18] realizes transparent task distribution in Java by partial thread migration across machines. The offloading is very fine-granular, i.e. procedure activation frames are transferred as needed and objects are fetched from target side on demand. The design somewhat differs to our goal of massive parallelization: We dispatch an entire set of tasks in one roundtrip, requiring the runtime system to collect all necessary code and data in advance, without suspending the program. Moreover, our system allows distributed tasks to modify disjoint fields and array elements even on the same object, and also detects distributed write-write data races.

5.7 Consistency Models

The work on cloud types with revision diagrams [4] proposes a more relaxed model of dealing with shared mutual state in concurrent and distributed systems on the basis of eventual consistency. This is no option in our system, as it ought to fulfill the standard .NET programming model, where unsynchronized read-write, write-read, and write-write accesses are low-level data races, i.e. programming errors. Introducing new types with a different consistency model would have sacrificed the transparent move from local to distributed parallel programming. While data races are not detected in .NET as well as in other mainstream programming languages (resulting in undefined behavior), our system goes beyond this by reporting at least write-write data races.

6. Discussions and Conclusions

The presented runtime system enables seamless distributed task parallelization with the illusion of shared memory. While the programming model remains principally identical to working with local parallel tasks, the runtime engine automatically dispatches tasks over a service onto remote processor resources in the cloud. In contrast to other less seamless systems, this liberates developers from any distribution-specific programming artefacts, such as developing explicit remote code, realizing explicit communication, implementing any serialization, or wrapping/marking/attributing code or data for distribution-awareness.

Of course, distributed task parallelization is not appropriate for all classes of parallel problems. It is rather designed for computing-intensive tasks or a large amount of tasks, where it can achieve very high speedups. Thereby, the total task execution time has to be significantly larger than the network-dependent transmission time of task data between the client and the service.

We see a high potential if programmers can use "parallelization-as-a-service" in a way that is as simple and convenient as our task parallelization in the cloud.

Certainly, there is room for various improvements that we would like to address in future: (1) The runtime system could be enhanced to support more features, especially nested task starts, task chaining, task canceling, as well as, remote monitoring and debugging. (2) It could be investigated on alleviating task isolation by permit-

[4] `http://cascading.org`

[5] `http://pegasus.isi.edu/`

[6] `http://jppf.org`

[7] `http://proactive.activeeon.com`

[8] `http://www.manjrasoft.com`

[9] `http://akka.io`

[10] `http://www.mpi-forum.org`

ting well-defined synchronizations across tasks. (3) Instead of conservative reachability analysis, one could study other techniques such as lazy data transmission requested by the server at task execution time. (4) It would be interesting to offer a public parallelization service where users can directly consume and perhaps also offer multi-processor power on demand.

7. Availability

The runtime system with program samples is available on our project website:

`http://concurrency.ch/Research/TaskParallelism`

Acknowledgment

I gratefully appreciate the feedback from Peter Sommerlad, Svend Knudsen and Thomas Corbat, helping to improve this paper. I also express my thanks to Henrik Nordborg from the Microsoft Innovation Center for Technical Computing for offering access on the Microsoft HPC Cluster at the HSR during the experimental evaluations.

References

[1] G. Agha. *Actors: a model of concurrent computation in distributed systems*. MIT Press, Cambridge, MA, USA, 1986. ISBN 0-262-01092-5.

[2] Y. Aridor, M. Factor, and A. Teperman. cJVM: a single system image of a JVM on a cluster. In *Parallel Processing, 1999. Proceedings. 1999 International Conference on*, pages 4–11. IEEE, 1999.

[3] F. Baude, D. Caromel, and M. Morel. From distributed objects to hierarchical grid components. In *International Symposium on Distributed Objects and Applications (DOA), Catania, Sicily, Italy, 3-7 November*, Springer Verlag, 2003. Lecture Notes in Computer Science, LNCS.

[4] S. Burckhardt, M. Fähndrich, D. Leijen, and B. P. Wood. Cloud types for eventual consistency. In *Proceedings of the 26th European Conference on Object-Oriented Programming*, ECOOP'12, pages 283–307, Berlin, Heidelberg, 2012. Springer-Verlag. ISBN 978-3-642-31056-0.

[5] C. Chambers, A. Raniwala, F. Perry, S. Adams, R. R. Henry, R. Bradshaw, and N. Weizenbaum. FlumeJava: easy, efficient data-parallel pipelines. In *ACM Sigplan Notices*, volume 45, pages 363–375. ACM, 2010.

[6] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti. Clonecloud: elastic execution between mobile device and cloud. In *Proceedings of the sixth conference on Computer systems*, pages 301–314. ACM, 2011.

[7] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl. MAUI: making smartphones last longer with code offload. In *Proceedings of the 8th international conference on Mobile systems, applications, and services*, pages 49–62. ACM, 2010.

[8] J. Dean and S. Ghemawat. MapReduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[9] M. S. Gordon, D. A. Jamshidi, S. Mahlke, Z. M. Mao, and X. Chen. COMET: code offload by migrating execution transparently. In *Proceedings of the 10th USENIX conference on Operating Systems Design and Implementation*, pages 93–106. USENIX Association, 2012.

[10] A. Gupta, E. Ababneh, R. Han, and E. Keller. Towards elastic operating systems. In *Proceedings of the 14th USENIX conference on Hot Topics in Operating Systems*, pages 16–16. USENIX Association, 2013.

[11] C. Hewitt, P. Bishop, and R. Steiger. A universal modular ACTOR formalism for artificial intelligence. In *Proceedings of the 3rd international joint conference on Artificial intelligence*, IJCAI'73, pages 235–245, San Francisco, CA, USA, 1973. Morgan Kaufmann Publishers Inc.

[12] F. Huet, D. Caromel, and H. E. Bal. A High Performance Java Middleware with a Real Application. In *Proceedings of the Supercomputing conference*, Pittsburgh, Pensylvania, USA, Nov. 2004.

[13] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. *ACM SIGOPS Operating Systems Review*, 41(3):59–72, 2007.

[14] E. Jul, H. Levy, N. Hutchinson, and A. Black. Fine-grained mobility in the Emerald system. *ACM Transactions on Computer Systems (TOCS)*, 6(1):109–133, 1988.

[15] D. Leijen, W. Schulte, and S. Burckhardt. The design of a task parallel library. In *Acm Sigplan Notices*, volume 44, pages 227–242. ACM, 2009.

[16] K. Li and P. Hudak. Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems (TOCS)*, 7(4):321–359, 1989.

[17] A. Luther, R. Buyya, R. Ranjan, and S. Venugopal. Alchemi: A. net-based enterprise grid computing system. In *International Conference on Internet Computing*, pages 269–278, 2005.

[18] R. K. Ma, K. T. Lam, C.-L. Wang, and C. Zhang. A stack-on-demand execution model for elastic computing. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 208–217. IEEE, 2010.

[19] E. Meijer, B. Beckman, and G. Bierman. Linq: reconciling object, relations and xml in the. net framework. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 706–706. ACM, 2006.

[20] D. G. Murray, M. Schwarzkopf, C. Smowton, S. Smith, A. Madhavapeddy, and S. Hand. CIEL: A universal execution engine for distributed data-flow computing. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, pages 9–9, Berkeley, CA, USA, 2011. USENIX Association.

[21] J. Rellermeyer, G. Alonso, and T. Roscoe. R-OSGi: Distributed applications through software modularization. In R. Cerqueira and R. Campbell, editors, *Middleware 2007*, volume 4834 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2007. ISBN 978-3-540-76777-0.

[22] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for general-purpose distributed data-parallel computing using a high-level language. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, pages 1–14, 2008.

[23] Y. Zhao, M. Hategan, B. Clifford, I. Foster, G. Von Laszewski, V. Nefedova, I. Raicu, T. Stef-Praun, and M. Wilde. Swift: Fast, reliable, loosely coupled parallel computation. In *Services, 2007 IEEE Congress on*, pages 199–206. IEEE, 2007.

[24] W. Zhu, C.-L. Wang, and F. C. Lau. Jessica2: A distributed java virtual machine with transparent thread migration support. In *Cluster Computing, 2002. Proceedings. 2002 IEEE International Conference on*, pages 381–388. IEEE, 2002.