The MuSE System: A Flexible Combination of On-Stack Execution and Work-Stealing

Markus Leberecht

Institut für Informatik
Lehrstuhl für Rechnertechnik und Rechnerorganisation (LRR-TUM)
Technische Universität München, Germany
Markus.Leberecht@in.tum.de

Abstract. Executing subordinate activities by pushing return addresses on the stack is the most efficient working mode for sequential programs. It is supported by all current processors, yet in most cases is inappropriate for parallel execution of indepented threads of control. This paper describes an approach of dynamically switching between efficient on-stack execution of sequential threads and off-stack spawning of parallel activities. The presented method allows to incorporate work-stealing into the scheduler, letting the system profit from its near-to-optimal load-balancing properties.

Keywords: Multithreading, Work-Stealing, Scalable Coherent Interface, PC Cluster

1 Introduction

For sequential execution, the most efficient method of calling a subroutine is to push the return address onto the program stack and to perform a branch operation. On return to the superordinate function, the previously stored continuation, the return address, can simply be retrieved by fetching it from the location pointed to by the stack pointer. A following stack-pointer correction finishes the necessary actions. Using the stack for storing continuations only works as the program order for sequential programs can be mapped onto a sequential traversal of the tree of function invocations, the *call tree*. A parallel program execution, on the other hand, has to be mapped onto a parallel traversal of the call tree that is now often named *spawn tree*. A simple stack here does not suffice to hold the continuations of more than one nested execution as can be seen from Fig. 1.

As interconnects for networks or clusters of PCs and workstations exhibit shrinking roundtrip latencies well below the 20 μs mark for small messages and bandwidths up to hundreds of MBytes/s, fine-grained parallelism has become a definite possibility even on these types of architectures. In order to utilize most of the offered communication performance, runtime system services have to be

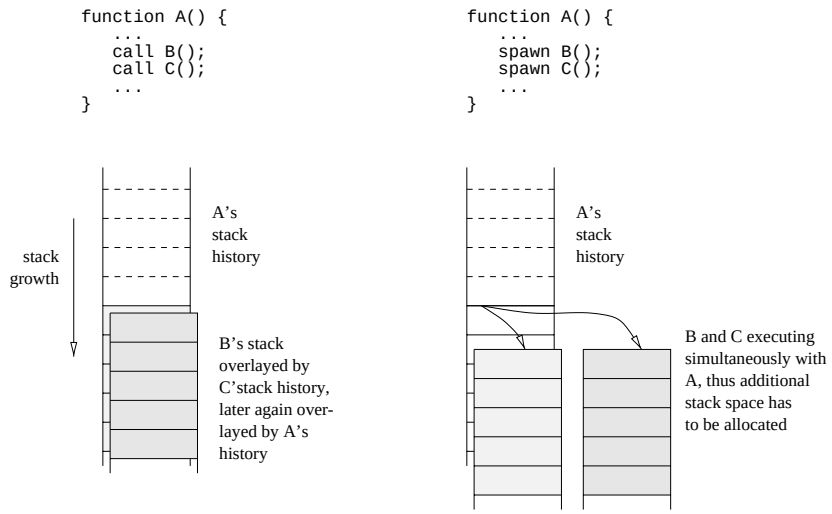

Fig. 1. Additional stack space is needed for parallel execution.

comparably efficient yet flexible. They e. g. need to adapt to the load situation and availability of nodes in a frequently changing cluster environment.

Spawning as a common runtime mechanism enables the migration of activities to underloaded nodes in a parallel or distributed system. Techniques of load balancing built on top of this feature are particularly important in the context of clusters of PCs with varying interactive background load.

The following text first presents the SMiLE cluster, a network of PCs connected by a contemporary low-latency network. Some communication performance figures are given in order to motivate the need for low-overhead runtime mechanisms, in particular with respect to work migration. The following section then describes the Multithreaded Scheduling Environment, a scheduler prototype with the ability to dynamically switch between on-stack and on-heap allocation of contexts. This property is utilized to implement a distributed work-stealing algorithm that achieves a significant improvement over those on runtime system using purely heap-based execution. A number of synthetic benchmarks are used to assess the performance of the proposed system. It can be concluded that speed-up still scales well in the given small test environment. A comparison with already existing systems and an identification of further improvements finish the text.

2 The MuSE System

2.1 The SMiLE Cluster of PCs

The name SMiLE is an acronym for *Shared Memory in a LAN-like Environment* and represents the basic outline of the project. PCs are clustered with the

Scalable Coherent Interface (SCI) interconnect. Being an open standard [11], SCI specifies the hardware and protocols allowing to connect up to 64 K nodes (processors, workstations, PCs, bus bridges, switches) in a high-speed network. A 64-Bit global address space across SCI nodes is defined as well as a set of read-, write-, and synchronization transactions enabling SCI-based systems to provide hardware-supported DSM with low-latency remote memory accesses. SCI nodes are interconnected via point-to-point links in ring-like arrangements or are attached to switches.

The lack of commercially available SCI interface hardware for PCs during the initiation period of the SMiLE project led to the development of our custom PCI-SCI hardware. Its primary goal is to serve as the basis of the SMiLE cluster by bridging the PCs I/O bus with the SCI virtual bus. Additionally, monitoring functions can be easily integrated into this extensible system. A good overview of the SMiLE project is given in [6]. The adapter is described in great detail in [1].

The current SMiLE configuration features four Pentium PCs running at 133 MHz with 32 MBytes of RAM and 2 GBytes of external storage coupled by four SMiLE PCI-SCI adapters in a ring configuration. The operating system is Linux 2.0.33.

Active Messages 2.0 [8] have been implemented on SMiLE as a communication layer particularly suited to be a building block for distributed runtime system communication. Its 0-byte message latency was measured to be $14\,\mu s$ while a maximum throughput of 25 MBytes/s can be reached with messages larger than 1 KByte.

2.2 The Multithreaded Scheduling Environment

The *Multithreaded Scheduling Environment (MuSE)* is a distributed runtime system specifically targeting the SMiLE platform. Programs running on MuSE are structured in a particular way in order to enable its own load balancing strategy.

Two basic methods exist in general for this purpose: while in *work-sharing* the load-generating nodes decide about the placement of new activities, *work-stealing* lets underloaded nodes request for new work. The *Cilk* project [2] highlighted that the technique of work-stealing is well suited for load balancing on distributed systems such as clusters of workstations. Its spawn-tree based scheduling ensures provably optimal on-line schedules with respect to time and space requirements. However, in order to permit the application of this technique, Cilk requires every subthread to be executed with a heap-allocated context. For program parts that are not actually executed in parallel due to runtime decisions or an excessive number of concurrent activities for the given number of processors, this means a potential source of inefficiency as unnecessary decomposition cost is paid.

MuSE in turn provides on-stack execution by default, trying to profit from its high sequential efficiency. In order to permit a Cilk-like work-stealing flexibility nonetheless, a different policy of spawning on-heap activities is required. The

following structure of the runtime system and its active entities provides the basis for this.

Organization of Active Entities. MuSE programs are compiled from the dataflow language SISAL. In intermediate steps, appropriate dataflow graphs are generated which are subsequently converted into MuSE-compliant C code. The front end of the *Optimizing SISAL Compiler (OSC)* [3] combined with a custom-made code generator and the SMiLE system's `gcc` are utilized to do this.

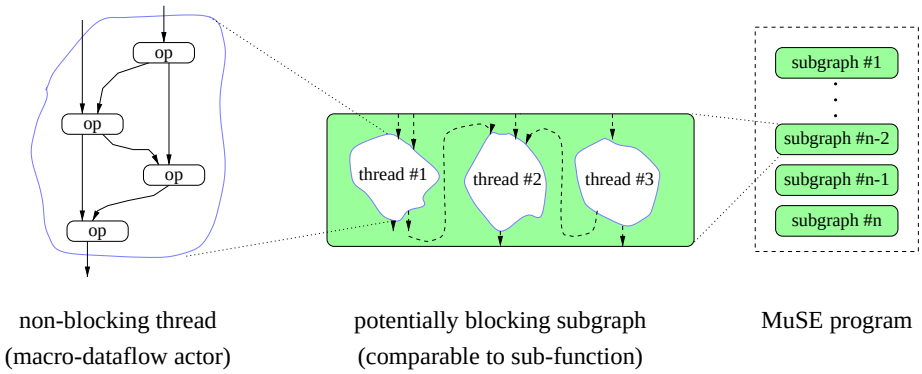

Fig. 2. MuSE program structure: non-blocking threads are combined into potentially blocking subgraphs a collection of which forms the MuSE program.

MuSE Graphs. A MuSE *graph* is the C code representation of IF1/2 dataflow graphs and follows their semantics. IF1/2 are intermediate dataflow representations put out by OSC and are described in [10]. The graph is represented by a regular C function with an appropriate declaration of parameters as well as return values. Input data is brought into the graphs via regular C function parameters, as is return data. Access to these within the graph is performed via a context reference, in this case pointing to the appropriate stack location. Thus, sequential execution of MuSE graphs is able to utilize the simple and efficient on-stack parameter passing of conventional sequential languages.

For parallel execution, i. e. on-heap allocation of graph contexts, the same mechanism is used. By allocating heap storage and changing the context reference to heap storage, spawning is prepared whenever necessary but transparently to the MuSE application code. The rationale behind the decision whether to either dynamically spawn or call a subgraph is explained below.

As spawning on a deeply nested level blocks at least one continuation within the caller until the callee has finished, heap context allocation will be performed for calling graphs on demand, resulting in lazily popping contexts off the stack.

In order to allow for blocking and to exploit intra-graph concurrency as well, graphs are subdivided into non-blocking activities called *threads*. Blocking by default has to occur at thread borders at which execution will later also resume. A graphical representation of graphs and their threads within a MuSE program is shown in Fig. 2.

MuSE Threads. A MuSE *thread* is a non-blocking sequence of C statements that is executed strictly, i. e. once all input data is present, threads can be run to completion without interruption. Threads are declared within a MuSE graph with the `THREAD_START(thread_number)` statement and finished with the following `THREAD_END` macro. These macros implement a simple guarded execution of the C statements forming the thread. The guard corresponds to a numerical synchronization counter responsible for implementing the dataflow firing rule.

MuSE threads are arranged by a simple as-soon-as-possible schedule within the graph as their underlying dataflow code is guaranteed by the compiler to be non-cyclic. Thus, whenever no blocking occurs, a MuSE graph executes like a regular C function.

Contexts. As already pointed out, in the case of blocking, MuSE graphs need to save their local data for later invocations. By default, graph parameters are located on the stack as are graph-local variables. Thus, this data has to be removed from the stack and put into a heap-allocated structure, the graph *context*. A context has to reflect the data also present on the stack during sequential execution and thus has to contain at least a pointer to the graph's C function, its input values as well as a container for the return value, some local variables, the threads' synchronization counters, and potentially the graph's continuation.

By choosing this form of implementing dataflow execution, MuSE effectively forms a two-level scheduling hierarchy. While the initial decision about which graph to execute is performed by the MuSE scheduler, the scheduling of threads within a graph is statically compiled in through the appropriate thread order. However, for nested sequential execution, the scheduler is only invoked once at the root of the call tree.

An advantage of this organization is the possibility to optimistically use calls instead of spawns to execute any MuSE graph. This means that only in those cases in which a spawn is occuring, the penalty for saving the context on the heap, memory allocation and copy operations, is paid.

MuSE Stack Handling. MuSE treats execution by default as being sequential first, thus attempting to use on-stack execution as long as it is possible. Only when the actual need arises, heap storage is requested and execution can proceed concurrently:

1. In the non-blocking case (a) of Fig. 3, execution can completely be stack-based.
2. In (b), some subgraph C blocks. The blocking can occur several levels deeper on the stack than where the sequential starting procedure or graph A is

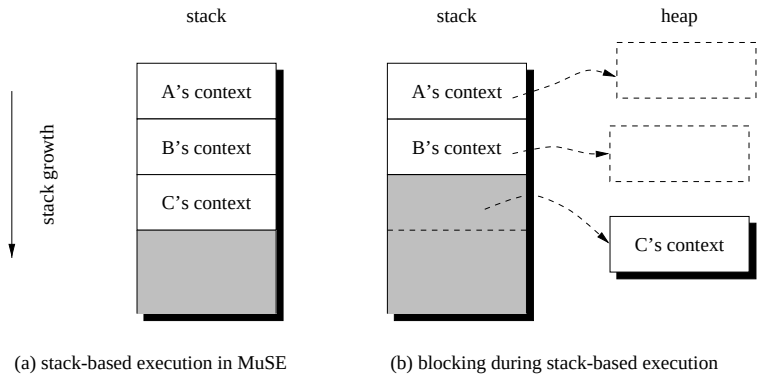

Fig. 3. Two cases during stack-based execution in MuSE: completion without blocking (a), or blocking at an arbitrary nesting level.

located. As C blocks, its data is transferred to a heap-allocated context. C returns to B, telling it through its return value that it has blocked. For B this means that at least one of its threads, C's continuation, will not be executed. This results in subsequent blocking and heap allocation operations until A is reached and all intermediate levels have blocked. During this, however, all remaining work that is executable in the spawn subtree below A has already been performed.

The described situation points out MuSE's advantage: being stack-based by default, this execution model offers a basically unlimited stack space to each new subgraph call. Even in the case of a deeply nested blocking graph, all remaining enabled work can proceed sequentially. Only those graphs that really ever block need context space from the heap.

Load Balancing and Parallelism Generation in MuSE. On the thread level of execution, the dataflow firing rule provides the basis for an effective self scheduling.

MuSE graphs are executed stricly, too. As a newly spawned context contains all input data and the graph's continuation, a context can basically be executed on any node. This property is used by the MuSE scheduler to distribute work across all participating nodes of a MuSE system.

MuSE Work-Stealing. While MuSE threads correspond roughly to Cilk threads, their grouping together into procedures or graphs has consequences regarding work-stealing. In Cilk, the grouping of threads into procedures does not exist for the work-stealing algorithm. The method is purely based on closures, i. e. migrating input data and continuations of ready threads.

In MuSE, threads do not have separate closures. Their parameters and synchronization counters are combined within the graph context. MuSE thus bases work-stealing on contexts instead of single thread closures. It is hoped that allocating a context once and performing a single migration per stolen graph, the slightly larger communication and memory management overhead can be amortized.

MuSE Queue Design. In Cilk, threads run to completion without being interrupted. The graphs in MuSE, however, can block. This means that a single ready queue does not suffice in MuSE. A graph that is currently not being executed can basically be in one of the three states `READY`, `BLOCKED`, and `UNBLOCKED` and is either placed in the *ready queue*, a *working queue*, or in a *notworking pool*.

The ready queue has to service potential thief nodes as well as the local scheduler in the same way as in Cilk. Therefore, a structure comparable to the LIFO/FIFO token queue of the ADAM machine [4] was chosen, its functionality, however, is equivalent to Cilk's *ready dequeue*.

Communication Communication across nodes in MuSE occurs in mainly three places:

- while passing return values,
- during compound data accesses, and
- during work-stealing of ready contexts.

Simple SCI transactions do not completely cover the functionality required to implement all three cases mentioned above. Thus, they have to be complemented by actions taking place on the local and the remote node. For instance, passing of dataflow values requires updating of synchronization counters, while work-stealing with its migration of complete contexts requires queue-handling capabilities connected to these communication operations.

In order to accomplish these tasks with the least overhead, MuSE bases all communication on the SMiLE Active Messages already mentioned in Sec 2.1. This is being done in accordance with other, similar projects. Cilk as well as the Concert system [9] all rely on comparable messaging layers. Active Messages have distinct advantages in this case over other communication techniques:

1. The definition of request/response pairs of handler functions enables the emulation of most communication paradigms through Active Messages. For MuSE, AMs allow to define a dataflow naming scheme of slots for dataflow tokens within contexts, and still operate on these in a memory-oriented way.
2. The handler concept of AMs allows to define sender-initiated actions at the receiver without the receiver side being actively involved apart from a regular polling operation.
3. Due to the zero-copy implementation on the SMiLE cluster, Active Messages actually offer the least overhead possible in extending SCI's memory oriented transactions to a complete messaging layer, as can be seen from its performance figures.

Polling Versus Interrupts. Technical implementations of SCI-generated remote interrupts unfortunetely exhibit high notification latencies, as was documented in [7] for the Solaris operating system and Dolphin's SBus-2 SCI adapter. The reason for this lies in context switch costs and signal-delivery times within the operating system. Thus, in order to avoid these, MuSE has to deal with repeatedly polling for incoming messages. In its current implementation, MuSE therefore checks the AM layer once during every scheduler invocation.

3 Experimental Evaluation

3.1 Basic Runtime System Performance

Table 1 summarizes the performance of several basic runtime services of MuSE on the SMiLE cluster that are explained below in more detail:

t_{AM_Poll} is the time required for the Active Messages poll when no incoming messages are pending. This effort is spent during every scheduler invocation.

t_{get_graph} represents the time typically spent for finding the a graph in the queues to be executed next.

$t_{exec,best}$ is the duration of an invocation of a graph by the scheduler whenever the context data is cached.

$t_{exec,worst}$ in contrast denotes the same time spent with all context data being non-cached.

t_{call} is the amount of time being spent for the on-stack calling mechanism.

t_{spawn} on the other hand represents the amount of time required for spawning a graph, i. e. allocating new context memory from the heap, initializing it, and placing it into the appropriate queue.

t_{return_data} summarizes the time of the operations necessary for passing return values on the same node whenever this cannot be done on stack.

t_{queue_mgmt} finally represents such actions as removing contexts from one queue and placing into another.

The most important result of Table 1 is the fact that the on-stack call overhead is smaller than the spawn overhead by more than an order of magnitude.

Unfortunately, the uncached execution of an empty graph points to a potential source of inefficiency of the current MuSE implementation: taking more than $20\,\mu s$ doing no actual application processing may be too high for significantly fine-grained parallel programs.

3.2 Load Balancing and Parallelism Generation.

Load balancing: `knary`(k, m, i) is a synthetic test program that dynamically builds a spawn tree. Three parameters define its behaviour, of which the spawn degree k describes the number of subgraphs being spawned in each graph, the tree height m specifies the maximum recursion level to which spawning occurs,

Function	Symbol	Time $[\mu s]$
empty poll operation	t_{AM_Poll}	0.9
obtain graph reference	t_{get_graph}	0.25
empty graph execution (cached)	$t_{exec,best}$	0.92
graph call overhead	t_{call}	0.26
graph spawn overhead	t_{spawn}	3.9
empty graph execution (non-cached)	$t_{exec,worst}$	21.9
local return-data passing	t_{return_data}	5.1
queue management	t_{queue_mgmt}	2.95

Table 1. Performance of various runtime system services.

and i is proportional to the number of idle operations performed in each graph before the spawn operation gets executed.

The first experiment consists of running `knary` on MuSE in a pure *on-heap* execution mode. The work-stealing mechanism performs load balancing among the participating nodes. This mode of operation is essentially equivalent to Cilk's. The case with $k = 4$ and $m = 10$ showed the most representative behaviour and is thus used in the remainder of this section.

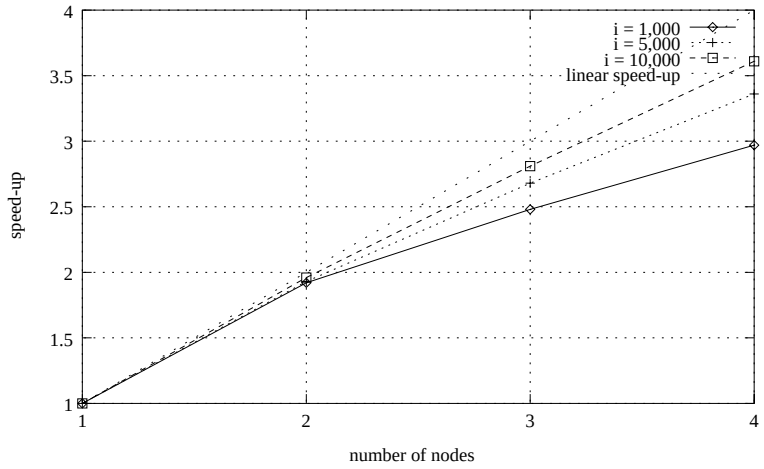

Fig. 4. Speed-up ratios of `knary`$(4, 10, 1000)$, `knary`$(4, 10, 5000)$, and `knary`$(4, 10, 10000)$ with sequential execution times of $20.6\,s$, $43.0\,s$, and $73.1\,s$ respectively.

Fig. 4 displays the gathered speed-up curves related to the respective single-node execution time. The result is as expected: work-stealing with per-default on-heap allocation of contexts achieves good speed-up ratios. MuSE thus conserves the positive Cilk-like properties. The drop compared to a linear speed-up can be

explained with the significant costs of work-stealing. The work-stealing latency is dominated by the time required for migrating a graph context from the victim to the steal node. Sending this 364 byte-sized chunk of memory is performed via a medium-length Active Message and takes approximately $t_{migrate} \simeq 120\,\mu s$ using the SMiLE AMs, representing a throughput of about 3.3 Mbytes/s for this message size. Only a few microseconds have to be accounted for the ready queue management on both sides and for the sending of the return data token. The gathered data supports this reasoning. As soon as the average execution time of a graph grows beyond the typical work-stealing latency — a value of $i = 5000$ represents an approximate graph runtime of $123\,\mu s$ — a work-steal becomes increasingly beneficial.

Load balancing with dynamic on-stack execution. It is expected that resorting to stack-based execution for mostly sequential parts of a program will help to speed-up a given application as the overhead for spawning in relation to the actual computation time shrinks. Switching these modes is based upon the following guidelines:

- MuSE uses by default the on-stack execution. Blocking is handled as described above and changes the execution mode for the blocked subgraphs to *on-heap*.
- On-stack execution does only involve the scheduler on the topmost level of the current spawn tree. Consequently, the polling operation within the main scheduler loop may be delayed. As this may hamper forward progress in the application and potentially starve other nodes due to not being serviced properly, the runtime system has to ensure that this does not happen. MuSE's strategy limits the total number of graph calls that are allowed to occur sequentially (`MAX_SEQUENTIAL`). Should no blocking operation occur during this time, the mode is nevertheless changed to on-heap execution in order to invoke the scheduler.

Again, `knary`$(4, 10, 1000)$ was used to derive reasonable numbers for both limits. Figure 5 displays the speed-up ratios for three different cases of `MAX_SEQUENTIAL`.

As absolute runtimes drop by a factor of 2.8 to 3.4, dynamic switching between the stack-based and the heap-based execution modes obviously offers fundamental improvements over the purely off-stack execution of the previous experiments.

In the single-node case MuSE takes care of executing the application purely on-stack, requiring only the initial graph plus its first level of four calls to be executed with a spawned context. Its overhead is easily amortized over the 349531 total subgraph calls in this program and thus represents the truly fastest single-node implementation. In contrast to this, the single-node runs of the previous experiments had to pay the unnecessary spawn overhead.

Increasing `MAX_SEQUENTIAL` from 10 to higher values also improves speed-up ratios, as can be seen from Fig. 5. `MAX_SEQUENTIAL` $= 100$ was chosen as a

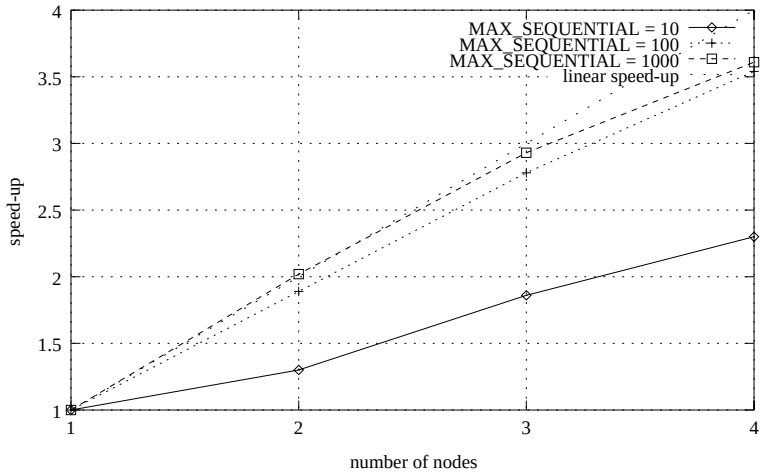

Fig. 5. Speed-up of `knary`$(4, 10, 1000)$ for different values of `MAX_SEQUENTIAL`. Sequential runtime was $t = 7.3\ s$.

compromise value since it realizes good speed-up while at the same time keeping the polling latency low. For instance, an average graph runtime of $2\,\mu s$ would yield a maximum of $200\,\mu s$ between two polls invoked by the scheduler. Higher values for `MAX_SEQUENTIAL` can still improve the speed-up somewhat, yet also increase the polling latency.

Load balancing under heterogeneous loads. In order to assess MuSE's capability of dealing with unevenly distributed background load while retaining its ability of switching execution modes dynamically, artificial load was systematically placed on each SMiLE node in the form of infinetely looping separate Linux processes. Assuming a normalized processing throughput of $Y_i = \frac{1}{p}$ for each node with p equally active processes on the node, the maximum ideal speed-up for n nodes would be

$$S_{max} = \sum_{i=1}^{n} Y_i. \tag{1}$$

Thus, for a single background load on one node and two otherwise unloaded nodes, MuSE execution can exploit a cumulated throughput of $\sum Y = \frac{1}{2}+1+1 = 2.5$ and therefore hope for a maximum speed-up of $S_{max} = 2.5$.

Again, `knary`$(4, 10, 1000)$ served as the test case. Table 2 summarizes the experiments with up to 4 background loads.

It is clearly visible that MuSE is able to balance the load even when the processing performance of the nodes is uneven. However, when the number of background processes increases, less and less of the offered throughput can be utilized. This can be attributed to the uncoupled Linux schedulers which are

nodes	wall-clock time [s]	speed-up	max. ideal speed-up	utilized max. performance
1 background load				
1	14.05	0.52	0.5	> 100 %
2	5.33	1.37	1.5	91.3 %
3	3.25	2.25	2.5	90.0 %
4	2.47	2.96	3.5	84.4 %
2 background loads				
2	7.70	0.95	1	95.0 %
3	4.26	1.71	2	85.7 %
4	3.42	2.13	3	71.2 %
3 background loads				
3	5.71	1.28	1.5	85.3 %
4	4.05	1.80	2.5	72.1 %
4 background loads				
4	4.84	1.51	2	75.4 %

Table 2. Absolute runtimes and speed-up values for `knary`$(4, 10, 1000)$ and $1 \ldots 4$ background loads.

bound to impede communication among the nodes more than necessary: while node A sends a request to node B, the latter is no longer being executed, forcing node A to wait for the next time slice of B.

Running more realistic SISAL applications unfortunately requires further tuning of the current prototypical and mostly C macro-based runtime system mechanisms. In particular, the prevailing implementation of MuSE's compound data handling suffers from significant inefficiencies that prevent actual application speed-up. This work therefore only covers synthetic benchmarks that nevertheless represent typical model cases.

4 Related Work and Conclusion

Load balancing actions usually introduce overhead due to two tasks: *decomposition cost* for setting up independent activities and *load migration cost* for transporting an activity to a different node. This distinction helps to differentiate the results of the seemingly similar execution models of MuSE and Cilk.

While in Cilk on-heap execution is the only and default execution mode, MuSE's default paradigm lets execution run on the stack and only switches back to on-heap execution when demanded. Or, to rephrase this differently, it means that decomposition cost for parallel execution in Cilk is implicitly hidden in the execution model and proportional to the number of Cilk threads being spawned — even for the sequential case in which this would not be necessary — while MuSE's costs for generating parallelism are made explicit by mode changes due to work-steal or fairness demands. This also means that Cilk's good speed-up

values are achieved partly due to a non-optimal sequential version while MuSE exhibits the fairer comparison: decomposition into concurrent active entities, the MuSE graphs with their contexts, only occurs when it is really necessary. In Cilk, only load migration cost has to be offset by the performance improvement while MuSE has to offset both costs yet still performs well.

Runtime systems suitable for fine-grained parallelism have to be efficient when supporting sequential as well as parallel programs. They thus have to overcome the difficulty of trading off efficient on-stack execution modes against more flexible but less lightweigt on-heap execution modes. Several methods for this have been presented so far:

- The *Lazy Threads* project [5] attempts to amortize heap allocation overhead by utilizing *stacklets*, independent and fixed-size chunks of heap space serving as stack for intermediate sequential execution. On stacklet overrun or a spawn operation, new stacklets are allocated, thus requiring static knowledge about when to spawn and when to call.

  In Lazy Threads, the decision about spawning and calling is non-reversible and has to be made at compile-time. MuSE in contrast offers a truly transparent model with the possibility to change execution modes at any time during the application's run.
- The *StackThreads* approach [12] defaults to sequential on-stack execution. In case of a spawn operation, a *reply box* in the calling thread serves as the point of synchronization for the spawned thread and its caller, blocking the caller until the spawned thread has completed.

  MuSE advances beyond StackThreads by not necessarily blocking a higher level activity as soon as a procedure has been spawned. Intra-graph parallelism through threads ensures that all ready execution can proceed on the stack while in StackThreads a single blocking operation always blocks an entire call tree.
- In the *Concert* system [9], differently compiled versions of the same code supporting calling as well as spawning are utilized dynamically.

  In contrast to Concert, MuSE's method requires only a single object code, avoiding the storage overhead of multiple versions.

MuSE thus offers distinct advantages over these described runtime systems. Unfortunately, its prototype implementation suffers from a number of inefficiencies. C macros that are used to implement certain runtime system functions and to enhance manual readability are not flexible enough for the presented purpose. They do not perform sufficiently well to allow for even more fine-grained parallelism and realistic dataflow applications. Although this is no actual surprise, they behaved interestingly well as a testbed for the presented mechanisms. Machine-level implementations of the runtime mechanisms, however, seem more appropriate as well as using more compiler knowledge to resort to less general and more optimized functionality whenever possible.

References

1. G. Acher, H. Hellwagner, W. Karl, and M. Leberecht. A PCI-SCI Bridge for Building a PC Cluster with Distributed Shared Memory. In *Proceedings of the 6th International Workshop on SCI-Based High-Performance Low-Cost Computing*, Santa Clara, CA, September 1996.
2. R. D. Blumofe and C. E. Leiserson. Scheduling Multithreaded Computations by Work Stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS '94)*, pages 356–368, Santa Fe, NM, USA, Nov 1994.
3. D. C. Cann. The Optimizing SISAL Compiler: Version 12.0. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, April 1992.
4. P. Färber. *Execution Architecture of the Multithreaded ADAM Prototype.* PhD thesis, Eidgenössische Technische Hochschule, Zurich, Switzerland, 1996.
5. S. C. Goldstein, K. E. Schauser, and D. E. Culler. Lazy Threads: Implementing a Fast Parallel Call. *Journal of Parallel and Distributed Computing*, 37(1):5–20, 25 August 1996.
6. H. Hellwagner, W. Karl, and M. Leberecht. Enabling a PC Cluster for High-Performance Computing. *SPEEDUP Journal*, June 1997.
7. M. Ibel, K. E. Schauser, C. J. Scheiman, and M. Weis. High-Performance Cluster Computing Using Scalable Coherent Interface. In *Proceedings of the 7th Workshop on Low-Cost/High-Performance Computing (SCIzzL-7)*, Santa Clara, USA, March 1997. SCIzzL.
8. A. M. Mainwaring and D. E. Culler. *Active Messages: Organization and Applications Programming Interface.* Computer Science Division, University of California at Berkeley, 1995. `http://now.cs.berkeley.edu/Papers/Papers/am-spec.ps`.
9. J. Plevyak, V. Karamcheti, X. Zhang, and A. Chien. A Hybrid Execution Model for Fine-Grained Languages on Distributed Memory Multicomputers. In *Proceedings of the 1995 ACM/IEEE Supercomputing Conference*, San Diego, CA, December 1995. ACM/IEEE.
10. S. Skedzielewski and J. Glauert. IF1 - An Intermediate Form for Applicative Languages. Technical Report TR M-170, Lawrence Livermore National Laboratory, July 1985.
11. IEEE Computer Society. *IEEE Standard for Scalable Coherent Interface (SCI).* The Institute of Electrical and Electronics Engineers, Inc., 345 East 47th Street, New York, NY 10017, USA, August 1993.
12. K. Taura, S. Matsuoka, and A. Yonezawa. StackThreads: An Abstract Machine for Scheduling Fine-Grain Threads on Stock CPUs. In T. Ito and A. Yonezawa, editors, *Proceedings of the International Workshop on the Theory and Practice of Parallel Programming*, volume 907 of *Lecture Notes of Computer Science*, pages 121–136, Sendai, Japan, November 1994. Springer Verlag.