SAC Goes Cluster

From functional array programming to distributed memory array processing

Thomas Macht[1,2] and Clemens Grelck[2]

[1] VU University Amsterdam
De Boelelaan 1105, 1081 HV Amsterdam, Netherlands
`t.macht@student.vu.nl`
[2] University of Amsterdam
Informatics Institute
Science Park 904, 1098 XH Amsterdam, Netherlands
`c.grelck@uva.nl`

Abstract. SAC (Single Assignment C) is a purely functional, data-parallel array programming language that predmoninantly targets compute-intensive applications. Thus, clusters of workstations, or more generally distributed address space supercomputers, form an attractive compilation target. Notwithstanding, SAC today only supports shared address space architectures, graphics accelerators and heterogeneous combinations thereof.
In our current work we aim at closing this gap. At the same time we are determined to uphold SAC's promise of entirely compiler-directed exploitation of concurrency, no matter what the target architecture is. It is well known that distributed memory architectures are going to make this promise a particular challenge.
Despite SAC's functional semantics, it is generally not straightforward to infer exact communication patterns from memory architecture agnostic code. Therefore, we intend to capitalise on recent advances in network technology, namely the closing of the gap between memory bandwidth and network bandwidth. We aim at a solution based on an implementation of software distributed shared memory (SDSM) and large per-node software-managed cache memories. To this effect the functional nature of SAC with its write-once/read-only arrays provides a strategic advantage that we aim to exploit.
Throughout the paper we further motivate our approach, sketch out our implementation strategy and show preliminary experimental evaluation.

1 Introduction

Single Assignment C (SAC) [9] is a functional data parallel language specialised in array programming. The goal of the language is to combine high productivity programming with efficient parallel execution. Data parallelism in SAC is based on array comprehensions in the form of `with`-loops that are used to create immutable arrays and to perform reduction operations. At this point, we can compile SAC source code into data parallel programs for shared memory architectures, CUDA-enabled graphics accelerators including hybrid systems and the MicroGrid architecture. However, the SAC compiler and runtime system do not yet support symmetric distributed memory architectures like clusters.

Our goal is to add efficient support for distributed memory architectures to the SAC compiler and runtime system. We aim to achieve competitive speedups for high-performance computing applications.

In a shared memory system, all nodes share a common address space. By contrast, in a distributed memory system, each node has a separate address space. In order to access remote data in a distributed memory programming model, the programmer must be aware of the data item's location and use explicit communication. While distributed memory systems can scale to greater size, the shared memory model is simpler to program. Distributed Shared Memory (DSM) aims to combine both models; it provides a shared memory abstraction on top of a distributed memory architecture. DSM can be realised in software or in hardware; hybrid solutions also exist. Partitioned Global Address Space (PGAS) is a programming model that lies in between the local and global view programming models. PGAS logically partitions a global address space such that a portion of it is local to each process, thereby exploiting locality of reference. PGAS is the underlying model of programming languages like Chapel [5].

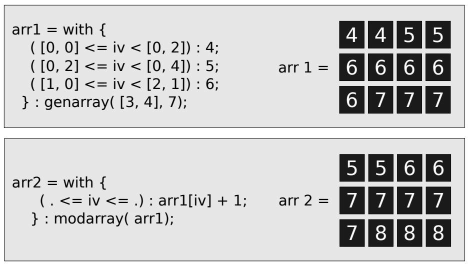
In the remainder of this paper we will first give an introduction to the SAC language and then motivate our current work, SAC for clusters. Subsequently, we will discuss our implementation and show preliminary performance results.

2 Single Assignment C

Single Assignment C (SAC) is a data parallel language for multi- and many-core architectures. For an introduction to SAC see [9]. The language aims to combine the productivity of high-level programming languages with the performance of hand-parallelized C or Fortran code. As the name suggests, the syntax is inspired by C. Other than C, however, SAC is a functional programming language without side-effects.

SAC is specialised in array programming; it provides multi-dimensional arrays that can be programmed in a shape-independent manner. While the language only includes the most basic array operations it comes with a comprehensive library. Conceptually, SAC's functional semantics requires to copy the full array whenever a single element is updated. To minimise the resulting overhead, SAC uses reference counting. This facilitates in-place updates of data structures when they are no longer referenced elsewhere. See [12] for SAC's memory management.

Array operations are typically implemented by `with`-loops, a type of array comprehension, which comes in three variants. See Figure 1 for examples. Both `genarray` and `modarray` `with`-loops create an array; `modarray` does so based on an existing array. For individual indices or sets of indices, expressions define the value of the corresponding array element(s). Independently for each index, the associated expression is evaluated and the corresponding array element is initialised. The third `with`-loop variant, `fold`, performs a reduction operation over an index set. As we have do not distribute these `with`-loops, we will not discuss them in this paper.



```
arr1 = with {
   ( [0, 0] <= iv < [0, 2]) : 4;
   ( [0, 2] <= iv < [0, 4]) : 5;
   ( [1, 0] <= iv < [2, 1]) : 6;
 } : genarray( [3, 4], 7);
```

```
arr2 = with {
     ( . <= iv <= .) : arr1[iv] + 1;
   } : modarray( arr1);
```

Fig. 1: Examples of genarray and modarray `with`-loops and resulting arrays

All variants of `with`-loops have in common that the compiler may evaluate individual expressions independently of each other in any order and that write-accesses are very restricted. These properties allow us to parallelise `with`-loops in an efficient way. While `with`-loops denote opportunities for parallelism, the decision whether they are actually executed in parallel or not is taken by the compiler and runtime system. At all times, program execution is either sequential or a `with`-loop is processed in parallel.

The SAC compiler is a many-pass compiler and emits platform-specific C code. The compiler spends a lot of effort on combining and optimising `with`-loops [10]. Currently, the compiler includes backends for symmetric multi-cores [8], GPUs (based on CUDA) [14] and the MicroGrid many-core architecture [15]. Heterogeneous systems are supported as well [6] and there have been experiments with OpenMP as a compilation target.

3 Motivation

In this section we argue why it is useful to add support for distributed memory architectures to SAC, why we followed a software DSM-based approach and why we decided to build a custom compiler-integrated DSM system.

3.1 Why support distributed memory architectures?

Distributed memory architectures are more cost-efficient, more scalable, and distributed memory architectures dominate high-performance computing. Currently, 86% of the TOP500 supercomputers are clusters and 14% have a Massively Parallel Processing (MPP) architecture [1] which is also a type of distributed memory system. While they are still predominant in commodity hardware, typical shared memory architectures have long vanished from the TOP500 list: single processors by 1997 and Symmetric multiprocessing (SMP) architectures by 2003 [1].

Message passing, and in particular MPI, is still the prevailing programming model for distributed memory systems [7]. While such a local view or fragmented programming model meets the performance requirements, it lacks programmability [5]. The programmer is responsible for the decomposition and distribution of data structures. Algorithms operate on the local portion of data structures and require explicit communication to access remote data. Data distribution and communication statements obscure the core algorithm.

By contrast, global view programming represents a higher-level alternative. In this model, the programmer works with whole data structures and writes algorithms that operate on these whole data structures. Data transfers and work distribution are handled implicitly. The algorithm is specified as a whole and not interleaved with communication. SAC offers a global view of computation to the programmer. By adding support for distributed memory architectures to SAC, we can utilise its global view programming model to make programming for distributed memory systems more efficient.

3.2 Why a software DSM-based solution?

Distributed Shared Memory (DSM) provides a shared memory abstraction on top of a physically distributed memory. An overview of issues of Distributed Shared Memory (DSM) systems can be found in [17]. DSM can be realised in software or hardware; hybrid systems also exist. In the context of this work, we focus on software solutions. According to [19], the first software DSM system was Ivy which appeared in 1984. Until the early 1990's, several other software DSM systems were proposed. Examples include Linda, Munin and Shiva [17].

These early DSM systems have not been adopted on a large scale due to shortcomings in performance. Explicit message passing, and in particular MPI, remain the predominant programming model for clusters. However, Bharath et al. suggest that it is time to revisit DSM systems [18]. They argue that early DSM systems were not successful because of slow network connections at the time. In the meantime, the picture has changed. Network bandwidth is comparable to main memory bandwidth and network latency is only one order of magnitude worse than main memory latency. According to Bharath et al. these developments reduce DSM to a cache management problem. They propose to use the improved network bandwidth to hide latency. As we will discuss in Section 4, our implementation uses that trick as well.

3.3 Why a custom DSM system?

In order to support distributed memory systems, we could run a SAC program on top of an existing software DSM system. Instead, we decided to integrate a custom DSM system into the SAC compiler and runtime system. This allows us to exploit SAC's functional semantics and its very controlled parallelism in `with`-loops. Since variables in `sac` have write-once/read-only semantics, we do not have to take into account that they could change their value. Furthermore, parallelism only occurs in `with`-loops and while arbitrary variables can be read in the body `with`-loop, only a single variable is written to.

4 Implementation of our distributed memory backend

We added support for distributed memory architectures to the SAC compiler and runtime system based on a page-based software DSM system. Every node owns part of each distributed array and the owner computes principle applies. All accesses to remote data are performed through a local cache. To abstract from the physical network and provide portability, we utilise existing one-sided Remote Direct Memory Access (RDMA) communication libraries. Currently, we support GASNet [4], GPI-2 [13], ARMCI [16] and MPI-3. In order to add support for a communication library, one only has to provide implementations for a small set of operations. These include initialisation and shut down of the communication system, an operation to load a memory page from a remote node and barriers.

4.1 Distributed arrays and memory model

Distributed execution is triggered by `with`-loops that generate distributed arrays. The runtime system decides whether an array is distributed based on the size of the array, the number of compute nodes and the execution mode at allocation time (see Section 4.4 for execution modes). Arrays are always distributed block-wise along their first dimension. The minimum number of elements per node such that an array gets distributed can be configured at compile time.

In memory, a distributed array does not form one contiguous block, but instead it is split into *number-of-nodes* blocks of memory corresponding to the elements that are owned by each node. We will motivate the choice for this memory model in Section 4.5.

For an illustrative example of the memory model, see Figure 2. The example uses two arrays, denoted by different colours, with fourteen and eight elements, respectively, and four compute nodes. The numbers in the boxes denote the element indices. Every node owns a share of each distributed array. The portion of the array that is owned by a node is located in that node's shared segment (e.g. elements 0 - 3/0 - 1 of the first/second array on Node 0). Note that the array sizes were chosen to simplify the example; in practise only arrays that are some orders of magnitudes bigger would be distributed. Furthermore, we assume for this example that a memory page can hold three array elements only.

Each node's DSM memory consists of memory for the shared segment and memory for the local caches. At program startup, each node pins a configurable amout of memory for its shared segment and reserves an address space of the same size for the caches of each other node's data. (De-)allocation of distributed arrays in DSM memory is taken care of by an adapted version of the SAC private heap manager [11]. When a distributed array is allocated, the runtime system

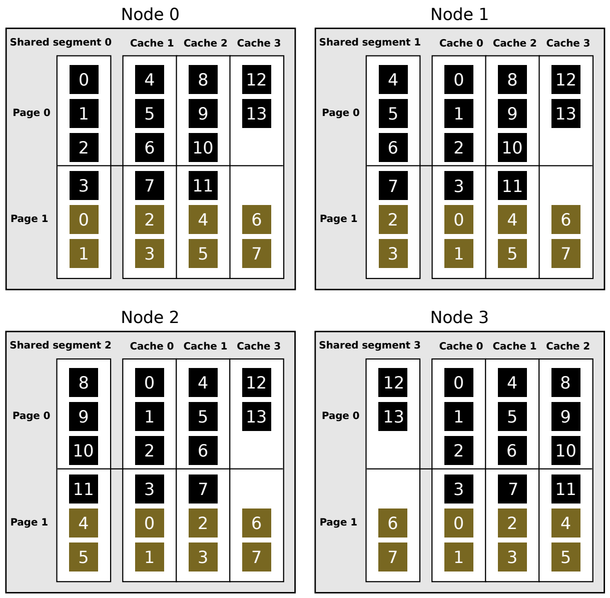


Fig. 2: Memory model for two distributed arrays (distinguished by different colours)

also reserves an address space of the same size within the local caches for all other nodes. To simplify locating array elements, the shared segment and caches are aligned. In the example, the second array starts at offset 4 in the shared DSM segment and all three cache segments on all four compute nodes. Non-distributed arrays, scalars and array descriptors are not allocated in DSM memory.

4.2 Array element pointer calculations

SAC supports multi-dimensional arrays; the translation of multi-dimensional array indices into vector offsets for memory accesses is taken care of by the compiler [3]. For the remainder of this paper, we assume that this conversion has already taken place. As explained in Section 4.1, a distributed array does not form a contiguous block of memory. The runtime system, therefore, needs to translate an offset to an array element to a pointer to the actual location of the element. This section describes how this is done and how we optimise this process.

In SAC, arrays have descriptors that hold a reference counter and, if not known at compile time, shape information. For each distributed array, we add two fields to the array descriptor: `first_elems` and `arr_offs`. The value of `first_elems` is the number of elements that are owned by each node except for the last node, which owns the remaining elements. The value of `arr_offs` is the offset at which the array starts within the shared segment of its owner node and within the cache for the owner at each other node. The formula for the pointer calculation is shown in Listing 1. The variable `segments` contains pointers to the local shared segment and the local caches; the rank of a node is the index of its segment within `segments`. The value of `elem_offs` is the offset of the requested element within the array assuming that the array would be allocated as one contiguous block of memory.

```
(segments[elem_offs / first_elems] + arr_offs) + (elem_offs % first_elems)
```

Listing 1: Formula for array element pointer calculations

In a naive implementation we would have to perform this pointer calculation for every access to an array element. However, we implemented three optimisations for write accesses, remote read accesses and local read accesses, respectively, so that the calculation can be avoided in most cases.

When writing distributed arrays we know that the elements we are writing to are local to the writing node because of the owner computes principle. We, therefore, simply keep a pointer to the start of the local portion of the array.

For remote read accesses we implemented a pointer cache. For each distributed array, we keep a pointer to the start of the array within the local cache for the node that owns the least recently accessed remote element of that array. In addition, we keep the offset of the first and last element that are owned by the same node.

For local read accesses we use the same pointer to the start of the local portion of the array that we use for write accesses. In addition, we keep the offset of the first and the last element that are local to the current node.

When a read access to an array element occurs, we first check whether the element is local to the current node by comparing its offset to the offsets of the first and last node that are local to the current node. If the element is local, we can use the pointer to the start of the local portion of the array in the local shared segment.

If the element is not local, we check whether it is owned by the same node as the last remote element of the same array that was accessed by comparing the offsets. If that is the case, we can use the pointer to the start of the array in the local cache for that node. Otherwise, we have to perform a pointer calculation as shown in Listing 1 to update the pointer cache.

4.3 Communication model and cache

According to the owner computes rule, a node only writes array elements that it owns. By contrast, every node can read all elements of a distributed array, including remote elements. This section describes the required communication for read accesses to remote array elements.

As mentioned in Section 4.1, the address space for the caches of remote elements is reserved when a distributed array is allocated. Initially, the caches are protected page-wise against all accesses by means of the `mprotect` system call. When a node tries to access remote data through its local cache, a `SIGSEGV` signal is raised. A custom handler then copies the appropriate memory page from the remote node into the local node's cache and allows accesses to it. Subsequent accesses to the same memory page can then be served directly from the local cache. The signal handler can calculate the requested array element and its location from the address where the segfault occurred. See Listing 1 for how to calculate the memory location of array elements.

When part of the cache becomes outdated, the corresponding memory pages are protected again. Distributed arrays are written in `with`-loops and we do not need any communication to trigger the required cache invalidations. Every node participates in the write operation and, therefore, knows that it has to invalidate the cache for that array on completion.

When a remote element is not in the local cache yet, we always load entire memory pages rather than single array elements. For an example, see Figure 2. When Node 0 first accesses Element 8 of the first array, Elements 9 and 10 will also be fetched from Node 2. Likewise, when Node 1 accesses Element 4 of the second array for the first time, Element 5 of the second and Element 11 of the first array will also be fetched from Node 2.

The rationale for loading entire pages is that thanks to advances in network technology, available bandwidth has increased so much that we can use it to hide latency [18]. Furthermore, the page-based approach allows us to use the operation system's memory page protection mechanism to decide whether an element is present in the cache or not.

4.4 Execution modes and barriers

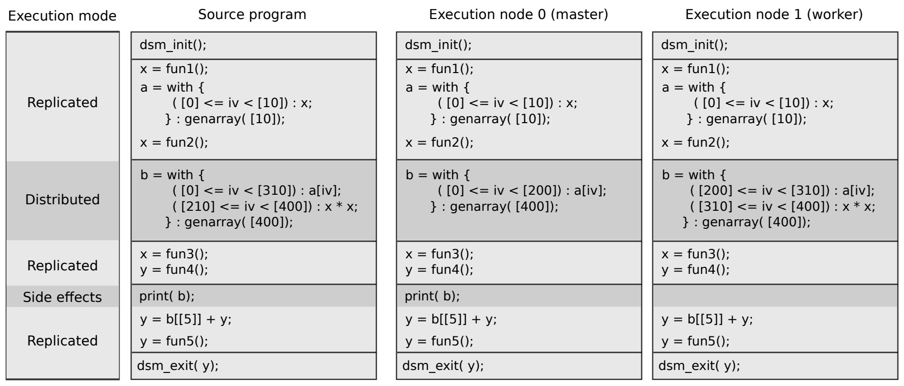
A distributed memory SAC program is always in one out of three execution modes: replicated, distributed or side effects execution mode. See Figure 3 for an illustrating example. In the following, we call the node with rank 0 master node and the remaining nodes worker nodes.

Program execution starts in replicated execution mode in which every node executes the same instructions on the same data. This way all nodes maintain the same execution environment without requiring communication.

In distributed execution mode, each node works on its share of the data. Currently, `genarray` and `modarray` `with`-loops are distributed iff the result array is distributed. Distributed memory SAC supports one level of distribution, an array and the `with`-loop that writes that array are not distributed if the program is already in distributed execution mode when the array is allocated.

In side effects execution mode, only the master node is executing and the workers are waiting until it is done. This is important because functions that have side effects, such as I/O, must not be executed more than once. If functions with side-effects yield any results, they are broadcast to the workers when the master is done.

In some cases we need barriers to preserve the correctness of the program in a distributed environment. For examples see Figure 3; the horizontal bars



```
dsm_init();
x = fun1();
a = with {
      ( [0] <= iv < [10]) : x;
    } : genarray( [10]);
x = fun2();
b = with {
      ( [0] <= iv < [310]) : a[iv];
      ( [210] <= iv < [400]) : x * x;
    } : genarray( [400]);
x = fun3();
y = fun4();
print( b);
y = b[[5]] + y;
y = fun5();
dsm_exit( y);
```

```
dsm_init();
x = fun1();
a = with {
      ( [0] <= iv < [10]) : x;
    } : genarray( [10]);
x = fun2();
b = with {
      ( [0] <= iv < [200]) : a[iv];
    } : genarray( [400]);
x = fun3();
y = fun4();
print( b);
y = b[[5]] + y;
y = fun5();
dsm_exit( y);
```

```
dsm_init();
x = fun1();
a = with {
      ( [0] <= iv < [10]) : x;
    } : genarray( [10]);
x = fun2();
b = with {
      ( [200] <= iv < [310]) : a[iv];
      ( [310] <= iv < [400]) : x * x;
    } : genarray( [400]);
x = fun3();
y = fun4();

y = b[[5]] + y;
y = fun5();
dsm_exit( y);
```

Fig. 3: Execution modes and barriers (horizontal bars)

denote barriers. In general, we require barriers after program startup and before program termination, before and after a distributed `with`-loop and before a function application with side effects.

The barrier after a distributed `with`-loop ensures that no stale data is read by other nodes because there were write accesses to the distributed array in the `with`-loop. The barrier before a distributed `with`-loop ensures that, in case memory is reused (see [12] for SAC's memory management), no other node needs to read the old data anymore before it is overwritten.

4.5 Motivation for memory model

As described in Section 4.1, distributed arrays do not form one contiguous block, but instead are split into *number-of-nodes* blocks of memory corresponding to the elements that are owned by each node. We explained in Section 4.2 that it is relatively expensive to calculate pointers to array elements with this memory model and proposed a pointer cache as a solution. Given this disadvantage, why do we propose the described memory model? For our argumentation we will assume that we use a *page-based* DSM system. We will, therefore, first elaborate on the reasons why we decided to build a *page-based* DSM system: to hide latency and to avoid overheads when checking whether an element is present in the cache.

On a cache miss, we fetch a whole memory page rather than a single element from the remote node that owns the element. Subsequent accesses to neighbouring elements can then be served from the cache. This allows us to use the available bandwidth to hide latency. In addition, if we fetch whole memory pages, we can use the operating system's page protection mechanism to decide whether a page is present in the cache or not. If a page is not present in the cache, a `SIGSEGV` signal is raised when we try to access it and the fetch from the remote node is taken care of by our custom signal handler. If a page is present in the cache, however, the access simply returns the data. The alternative to using the page protection mechanism would be to keep track of the cached elements ourselves, but that would involve a search in a possibly large data structure. This search would incur additional overheads, also in the case that an element is already present in the cache.

Having decided that we want to use a page based DSM system, why do we use the described memory model? SAC supports multi-dimensional arrays and `with`-loops that generate multi-dimensional arrays are compiled to complex nested loop structures with a loop for each array dimension. We need to make sure that the distribution happens along a single dimension; in practise along the outermost dimension. Otherwise, the iteration of the index space becomes impractically complex, especially when considering that the size and dimensionality of arrays is often not known at compile time.

We have established that we want to use a page-based DSM system and that the distribution of the array should happen along the outermost dimension. If an array was to form a single contiguous form of memory we would then have to partition it at memory page borders. However, we have also established that the distribution should happen along the outermost dimension. Unfortunately, these two demands generally cannot be met at the same time.

Another benefit of our memory model is that it allows us to solve larger problems. With contiguous arrays, we would need to allocate the entire array within the DSM segment so that remote nodes can read the local portion of it. Unfortunately, the size of the DSM segment is limited by hardware constraints. In any case, it cannot be larger than the node's physical memory. By contrast, in our memory model, we only allocate the local part of the array within the DSM segment. The caches are allocated outside of the DSM segment using `mmap`. Until a memory page is accessed for the first time, only an address space is reserved but no phyiscal memory is provided.

5 Evaluation of our distributed memory backend

We evaluate the performance of our distributed memory backend for SAC by means of experiments in the areas of image convolution, matrix multiplication and N-body simulation. In the following, we will first describe the experimental setup and then discuss the results of the individual experiments.

5.1 Experimental setup

All experiments were performed on the VU cluster side of the DAS-4 supercomputer system [2]. The VU cluster side consists of 74 dual quad-core 2.4 GHz compute nodes with 24 GB of memory each. The nodes are interconnected by Gigabit Ethernet as well as high speed InfiniBand. We used the following versions of the supported communication libraries for our experiments: GASNet 1.24.0, GPI-2 1.1.1, ARMCI as included in Global Arrays 5.3 and the Open MPI 1.6.5 implementation of MPI-3.

In our experiments, we compare the runtimes of the program compiled for our distributed memory backend (`dm`) to the runtimes of the sequential SAC program (`seq`). With the distributed program, we start each process on a separate compute node. For $N \leq 8$ (as the nodes of the DAS-4 system have eight cores), we also compare the performance of our distributed memory backend program run by multiple processes on a single node (`dm-sn`) to the performance of the multi-threaded SAC program `mt`.

For all included measurements, we compared the output of the distributed memory backend program to the output of the sequential program to ensure that the program yields correct results. We measure the kernel execution time of the calculations and not the total execution time of the program. The reason is that the setup of the communication libraries and the printing of the result arrays to check the correctness take a considerable amount of time and that would otherwise distort our results. For real-world applications, the compute time would be much longer, whereas the setup time remains nearly constant and, thus, can be neglected.

We performed all experiments at least three times or more often if there was a high variance in the results. From all measurements, we take the minimum execution time for each program version rather than the average execution time. Our justification is that there may be background processes running on the compute nodes that have an influence on our experiments. All reported speedups are with respect to the sequential SAC program (`seq`).

5.2 Image convolution

First, we present our image convolution experiments. We include image convolution in our evaluation because it is a simple application where array element accesses show a high degree of locality. We have optimised our implementation for that by fetching entire pages on a cache miss and by using optimisations such as array pointer caches (see Section 4.2).

The `gaussBlurOpt` test program performs twenty iterations of a 3 x 3 kernel Gaussian blur on a 50,000 x 8,000 = 400,000,000 elements integer array. Figure 4 shows the performance results for `gaussBlurOpt`. For this program, we achieve speedups of more than 80% of linear for up to sixteen nodes.

5.3 Matrix multiplication

We also include experiments with matrix multiplication, because, compared to image convolution, it requires more communication. In this way, the matrix multiplication experiments are a stress test for the communication performance of our distributed memory backend for SAC.

The `matmulBigDiff` program performs ten iterations of a multiplication of two matrices with 2,000 x 2,000 = 4,000,000 double-precision floating point elements each. Implementation-wise, we first transpose the second matrix before we calculate the result matrix. Figure 5 shows our measurements for the

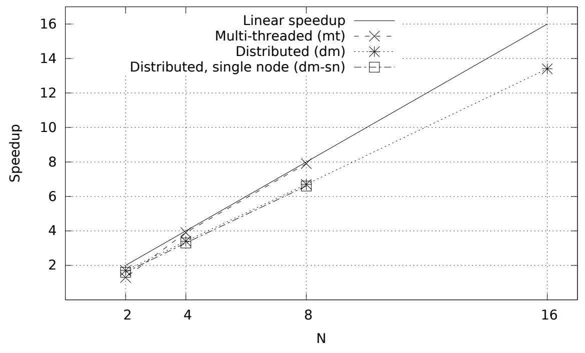


Fig. 4: Speedups for the `gaussBlurOpt` program (twenty iterations of a 3 x 3 kernel Gaussian blur on a 50,000 x 8,000 = 400,000,000 elements integer array)

`matmulBigDiff` program: for eight nodes we achieve a speedup of 3.2 (40% of linear) and for sixteen nodes a speedup of 4.2 (26% of linear).

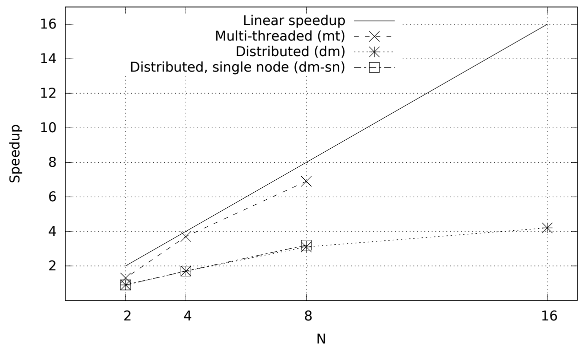


Fig. 5: Speedups for the `matmulBigDiff` program (ten iterations of a multiplication of two matrices with 2,000 x 2,000 = 4,000,000 double-precision floating point elements each)

5.4 N-Body simulation

Finally, we present the measurements for our all-pairs N-body problem experiments. The SICSA N-body challenge simulates the movements of a system of planets in three-dimensional space over time. Our program is based on the SAC implementation proposed in [20].

The `nbodyBig` program performs fifty iterations for 16,384 planets. Figure 6 show the measurements for the `nbodyBig` program. We achieve approximately 50% of linear speedups for up to sixteen nodes.

For the `nbodyBig` program, we also compare the minimum runtimes with the different communication libraries GASNet, ARMCI, GPI-2 and MPI-3. In Figure 7, we can see that MPI shows the weakest overall performance. Overall, GASNet is slightly faster than ARMCI and GPI-2.

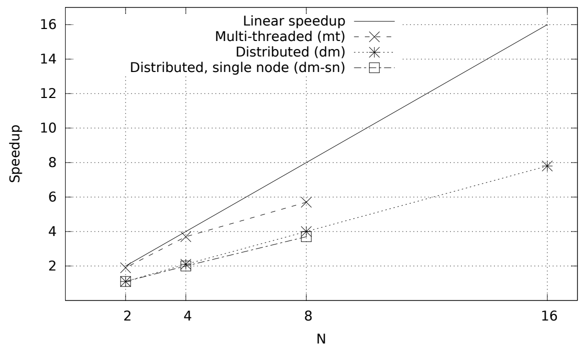


Fig. 6: Speedups for the `nbodyBig` program (N-body simulation: movements of 16,384 planets, 50 iterations)

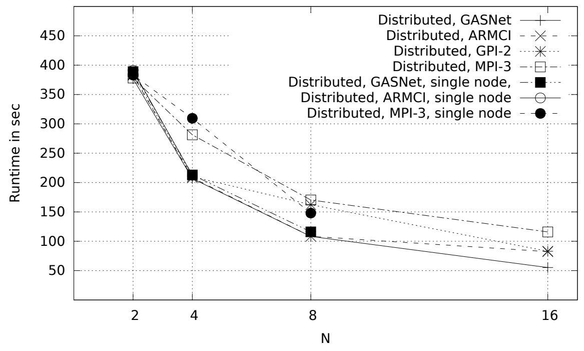


Fig. 7: Minimum runtimes with different communication libraries for the `nbodyBig` program (N-body simulation: movements of 16,384 planets, 50 iterations)

6 Conclusions and future work

6.1 Conclusion

In this paper, we have presented our implementation of a new compiler backend for SAC that supports symmetric distributed memory architectures like clusters of workstations. A particular challenge in doing so is upholding SAC's promise of entirely compiler-directed exploitation of concurrency.

We propose a DSM-based implementation where all accesses to remote data go through large local caches. Initially, the caches are protected by means of the `mprotect` system call. When a memory page is first accessed, a `SIGSEGV` signal is raised. A custom signal handler fetches the requested data from the remote node and subsequent accesses to the same data can be served directly from the cache.

While there is a lot of work to be done, our first results are promising. For our convolution experiments, we achieve 80% of linear speedups, for our N-body simulation approximately 50% of linear speedups and for matrix multiplication about one third of linear speedups.

6.2 Future Work

Possible future research directions lie in the areas of general performance improvements, the combination with multi-threading, cache eviction and distributed I/O. In the following, we briefly elaborate on these topics.

We want to improve overall performance by reducing the number of barriers. Furthermore, we want to make read operations to distributed arrays more efficient by avoiding locality checks and/or reducing overheads caused by them.

To fully utilise clusters of multi-core compute nodes, we want to combine the distributed memory backend with SAC's multi-threaded execution facilities [8]. We expect that we can achieve higher speedups with a hybrid solution that combines distributed execution and multi-threading.

Other than speeding up program execution, distributed execution has another advantage: It allows us to solve problems that do not fit into the memory of a single node. This is already possible to some extent in our solution, but to support the general case, we would need to add a cache eviction scheme.

Currently, functions that have side effects including I/O are only executed by the master node. We decided for this implementation to ensure that existing SAC libraries work correctly with the distributed memory backend. However, in many situations it would be more efficient to distribute I/O operations.

References

1. TOP500 supercomputer sites (2014), `http://top500.org/`, accessed on 19 February 2015
2. DAS-4: Distributed ASCI supercomputer 4 (2015), `http://www.cs.vu.nl/das4/home.shtml`, accessed on 25 July 2015

3. Bernecky, R., Herhut, S., Scholz, S.B., Trojahner, K., Grelck, C., Shafarenko, A.: Index Vector Elimination - Making Index Vectors Affordable, pp. 19–36. Implementation and Application of Functional Languages, Springer (2007)
4. Bonachea, D.: GASNet specification, v1. 1. Tech. rep., University of California at Berkeley (2002)
5. Chamberlain, B.L., Callahan, D., Zima, H.P.: Parallel programmability and the Chapel language. International Journal of High Performance Computing Applications 21(3), 291–312 (2007)
6. Diogo, M., Grelck, C.: Towards heterogeneous computing without heterogeneous programming, pp. 279–294. Trends in Functional Programming, Springer (2013)
7. Dongarra, J.J., der Steen, A.V.: High-performance computing systems: Status and outlook. Acta Numerica 21, 379–474 (2012)
8. Grelck, C.: A multithreaded compiler backend for high-level array programming. In: Applied Informatics. pp. 478–484 (2003)
9. Grelck, C.: Single Assignment C (SAC): High Productivity Meets High Performance, pp. 207–278. Central European Functional Programming School, Springer (2012)
10. Grelck, C., Scholz, S.B.: SAC: off-the-shelf support for data-parallelism on multicores. In: Proceedings of the 2007 workshop on Declarative aspects of multicore programming. pp. 25–33. ACM (2007)
11. Grelck, C., Scholz, S.B.: Efficient heap management for declarative data parallel programming on multicores. In: 3rd Workshop on Declarative Aspects of Multicore Programming (DAMP 2008), San Francisco, CA, USA. pp. 17–31 (2008)
12. Grelck, C., Trojahner, K.: Implicit memory management for SAC. In: Implementation and Application of Functional Languages, 16th International Workshop, IFL. vol. 4, pp. 335–348 (2004)
13. Grünewald, D., Simmendinger, C.: The GASPI API specification and its implementation GPI 2.0. In: 7th International Conference on PGAS Programming Models. vol. 243 (2013)
14. Guo, J., Thiyagalingam, J., Scholz, S.B.: Breaking the GPU programming barrier with the auto-parallelising SAC compiler. In: Proceedings of the sixth workshop on Declarative aspects of multicore programming. pp. 15–24. ACM (2011)
15. Herhut, S., Joslin, C., Scholz, S.B., Grelck, C.: Truly nested data-parallelism: compiling SAC for the Microgrid architecture. Draft proceedings of the 21st International Symposium on Implementation and Application of Functional Languages (2009)
16. Nieplocha, J., Carpenter, B.: ARMCI: A portable remote memory copy library for distributed array libraries and compiler run-time systems, pp. 533–546. Parallel and Distributed Processing, Springer (1999)
17. Nitzberg, B., Lo, V.: Distributed shared memory: A survey of issues and algorithms. Distributed Shared Memory-Concepts and Systems pp. 42–50 (1991)
18. Ramesh, B., Ribbens, C.J., Varadarajan, S.: Is it time to rethink distributed shared memory systems? In: Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on. pp. 212–219 (2011), iD: 1
19. Ramesh, B.: Samhita: Virtual shared memory for non-cache-coherent systems (2013)
20. Šinkarovs, A., Scholz, S.B., Bernecky, R., Douma, R., Grelck, C.: SAC/C formulations of the all-pairs N-body problem and their performance on SMPs and GPGPUs. Concurrency and Computation: Practice and Experience 26(4), 952–971 (2014)