UCRL-LR-122353

An Implementation of SISAL for Distributed-Memory Architectures

Patrick C. Beard

June 1995

Lawrence Livermore National Laboratory

UCRL-LR-122353
Distribution Category UC-705

An Implementation of SISAL for Distributed-Memory Architectures

Patrick C. Beard
University of California, Davis

Master of Science Thesis

Manuscript date: June 1995

LAWRENCE LIVERMORE NATIONAL LABORATORY
University of California • Livermore, California • 94551

DLC

An Implementation of SISAL for Distributed-Memory Architectures

By

Patrick Charles Beard

BSME, University of California at Berkeley, 1987

THESIS

Submitted in partial satisfaction of the requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

OFFICE OF GRADUATE STUDIES

of the

UNIVERSITY OF CALIFORNIA

DAVIS

Approved:

John Feo (Chair)

Kent Wilken

[illegible]

Committee in Charge

1995

Patrick C. Beard
June 1995
Computer Science

An Implementation of SISAL
for Distributed-Memory Architectures

Abstract

This thesis describes a new implementation of the implicitly parallel functional programming language SISAL, for massively parallel processor supercomputers. The Optimizing SISAL Compiler (OSC), developed at Lawrence Livermore National Laboratory, was originally designed for shared-memory multiprocessor machines and has been adapted to distributed-memory architectures. OSC has been relatively portable between shared-memory architectures, because they are architecturally similar, and OSC generates portable C code. However, distributed-memory architectures are not standardized – each has a different programming model. Distributed-memory SISAL depends on a layer of software that provides a portable, distributed, shared-memory abstraction. This layer is provided by Split-C, a dialect of the C programming language developed at U. C. Berkeley, which has demonstrated good performance on distributed-memory architectures. Split-C provides important capabilities for good performance: support for program-specific distributed data structures, and split-phase memory operations. Distributed data structures help achieve good memory locality, while split-phase memory operations help tolerate the longer communication latencies inherent in distributed-memory architectures. The distributed-memory SISAL compiler and run-time system takes advantage of these capabilities. The results of these efforts is a compiler that runs identically on the Thinking Machines Connection Machine (CM-5), and the Meiko Computing Surface (CS-2).

Acknowledgments

This work was supported by the U.S. Department of Energy, by Lawrence Livermore National Laboratory (LLNL) under contract No. W-7405-ENG-48. Funding was also provided by the Student Employee program administered by the director's office. I would like to thank my committee members: Dr. John Feo (chair), Dr. David Culler, and Dr. Kent Wilken. Kudos to my colleagues at LLNL for their invaluable guidance: Scott Denton, Pat Miller, Cindy Solomon, Srdjan Mitrovic, Tom DeBoni, and Judy Michels. I would also like to thank the Berkeley team for letting me use (and abuse) their CM-5's, and for creating (and maintaining) Split-C: Chad Yoshikawa (Meiko Split-C), Steve Lumetta (Split-C debugger), and Arvind Krishnamurthy (T3D Split-C).

Most of all, thanks to my fiancé, Bettina Kamriani, for her wisdom, patience and confidence in me, and my parents for helping me when I needed it most.

Contents

Acknowledgments iii

List of Listings vi

List of Figures vi

List of Tables vii

1 Introduction 1

1.2 Future of SISAL 3

1.3 Scope of the Thesis 3

2 SISAL Compiler Overview 5

2.1 A Tour of the SISAL System 5

2.1.1 SISAL's Run-Time System 5

2.1.2 Back-End Code Generator 8

2.2 Distributed-Memory Programming Model 9

2.3 Implications for SISAL 10

2.4 Distributed Shared-Memory 12

2.4.1 The Split-C Language 12

2.4.2 The Split-C Programming Model 14

3 Implementing Distributed-Memory SISAL 15

3.1 Getting SISAL Running 15

3.1.1 Run-time System Changes 15

3.1.2 Code Generation Changes 19

3.2 Compiler Verification 22

3.3 Optimizations for Distributed-Memory 23

3.3.1 SISAL Run-Time System Optimizations 23

4 Results & Discussion 26

4.1 Performance Studies 26

4.1.1 Performance Studies of Life 27
4.1.2 Performance Studies of Laplace 31
4.1.3 Performance Studies of Matrix Multiplication 33
4.2 Related Work 34
4.3 Conclusions & Future Work 35
Bibliography 38
Appendix 40
A.1 SISAL Program: Life 40
A.2 SISAL Program: Laplace 42
A.2.1 Original Laplace 42
A.2.2 Modified Laplace 43
A.3 SISAL Program: Matrix Multiply 44
A.3.1 Simple Matrix Multiply 44
A.3.2 Pre-Transposed Matrix Multiply 45

List of Listings

listing 3.1: Parallel initialization of the distributed task queue 16
listing 3.2: C representation of a global pointer 17
listing 3.3: Split-C post-increment code 17
listing 3.4: Equivalent C post-increment code 17
listing 3.5: Typedef for SISAL array data type 18
listing 3.6: Typedef for global pointer to SISAL array 18
listing 3.7: Macro for addition of any types 20
listing 3.8: Typedef for global pointer to SISAL array 21
listing 3.9: Simple SISAL program to build an array in parallel 22
listing 4.1: Data type for activation records 36

List of Figures

figure 4.1: Life evolution, 10x10 initially, and after 2 iterations 27
figure 4.2: Life speedups for 64 processor CM-5 29
figure 4.3: 32 processor Meiko CS-2, 2 iterations 31

List of Tables

table 4.1: CM-5 Life performance for 100x100 & 250x250 grid size 28

table 4.2: CM-5 Life performance for 500x500 & 1000x1000 grid size 29

table 4.3: Meiko Life performance for 100x100 & 250x250 grid size 30

table 4.4: Meiko Life performance for 500x500 & 1000x1000 grid size 30

table 4.5: Original Laplace performance for 256x256 grid size 32

table 4.6: Modified Laplace performance for 256x256 & 512x512 grid size 32

table 4.7: Modified Laplace performance for 1024x1024 grid size 32

table 4.8: Matrix Multiplication without pre-transposition 33

table 4.9: Matrix Multiplication with pre-transposition 33

table 4.10: Relative speedups using pre-transposition 34

1 Introduction

"16,384 processors make short work" – MPP project motto [Sc85]

1.1 History of SISAL

SISAL (Streams, Iteration, in a Single Assignment Language) is a special purpose functional language intended to be a comfortable, expressive and efficient alternative to FORTRAN for parallel scientific computing. The language was developed in a collaborative effort between Lawrence Livermore National Laboratory (LLNL), Colorado State University, the University of Manchester, and Digital Equipment Corporation. SISAL was initially defined in 1983 (v1.0), revised in 1985 (v1.2), and branched into two main development directions [Mc93]. Version 2.0 of the language was defined by Colorado State University and LLNL, but was never implemented. SISAL-90 is being developed at LLNL and will incorporate features from SISAL 2.0, as well as features from FORTRAN-90 [Fe95].

The goals of the SISAL project, given in [Fe90], are: to create a general purpose language that lets scientific programmers concentrate more on solving problems and less on implementation issues; to invent optimization techniques for high-performance parallel computing; to provide dataflow computing on conventional hardware; and to prove that applicative (functional) computing can be used for scientific computation.

In 1995, what can we state about the success of the SISAL project? Has it met its goals? SISAL is implicitly parallel, so that while a programmer need not be concerned with the low-level details of parallel machines, his choice of algorithm will affect how much parallelism the compiler can realize. Still, all issues such as parallelism management, communication and synchronization are handled by the compiler and the runtime system, without involving the SISAL application programmer. This meets the first stated goal quite well.

In the areas of functional language optimization, SISAL implementors have pioneered in the areas of dataflow optimization for non-dataflow architectures, copy elimination and modify-in-place analysis (both critical for applicative languages to perform well), and parallelism detection and utilization [Ca92]. The current release version of the SISAL compiler supports loop-level parallelism – earlier versions provided function-level and producer-consumer[1] parallelism.

SISAL has proven the viability of functional languages for scientific computing. Worldwide, there are more than 250 users of SISAL at 75 sites, and there are yearly conferences and workshops devoted to the language. SISAL is an efficient alternative to imperative languages even running sequentially. Recent comparative studies discussed in [Mc93] show SISAL performing only 5 to 20% slower than FORTRAN when running comparable algorithms on a single processor, but 10% faster on 8 processors, without any source code changes in the SISAL program.

SISAL has also proven to be a fruitful tool for research. Several groups outside of LLNL have used the SISAL language and compiler as starting points for new languages [Sa95]. One researcher has implemented an APL compiler that generates SISAL as its target language. There are also implementations of FORTRAN, C, Ada, Pascal and Id, that generate the same intermediate form as SISAL, and use SISAL's back end.

[1] Also called streams. An experimental version of the SISAL compiler was recently released that supports stream parallelism.

1.2 Future of SISAL

Since SISAL is an implicitly parallel functional language, it has been remarkably versatile in taking advantage of disparate architectures. At present, SISAL runs on more than seventeen different platforms, from running sequentially on UNIX, DOS, and Macintosh workstations, to running in parallel on shared-memory multiprocessors such as the Encore Multimax, and utilizing vector processing on the Crays. However, SISAL has not been available for the latest generation of parallel computers, distributed-memory, massively parallel processor (MPP) machines, which require significantly different programming techniques.

MPP computers provide a large number of processing elements (PEs) which contain a memory module and one or more processors, interconnected by a "scalable" communication network. ([Ku94] provides a good survey of communication networks.) Communication between PEs is often explicitly programmed, rather than being hidden by virtual memory or cache hardware[2]. Since MPPs typically do not provide a global address space, and the current SISAL compiler only generates code for shared-memory machines, getting SISAL programs to run on MPP hardware has required a substantial engineering effort, as this thesis will show.

1.3 Scope of the Thesis

This thesis describes the modification of the Optimizing SISAL Compiler (OSC) for distributed-memory, massively parallel processing (MPP) supercomputers. Section 2 provides an overview of the SISAL system, presents some of the issues of compiling for distributed-memory architectures, and describes how these issues are addressed in the distributed-memory SISAL system.

[2]The Cray T3D does provide hardware that translates a virtual address into a node and physical address, but must still be programmed explicitly. Split-C hides this.

Section 3 describes the construction of the new distributed-memory SISAL system. The work proceeded in four phases: (1) identifying the requirements of compiling for MPP machines, choosing a target language and shared memory abstraction that would be portable between MPP machines, and studying the original shared-memory compiler to plan the modifications; (2) rewriting the run-time system, implementing the necessary changes for MPP machines and testing it on a real MPP machine (the CM-5); (3) hand-modification of the output of the SISAL compiler to function with the new run-time system, and using these results to modify the code generator to emit MPP-compatible code; (4) moving the compiler to a different MPP architecture to demonstrate the portability of the implementation.

Finally, section 4 examines the performance of distributed-memory SISAL, suggests improvements, and discusses related work.

2 SISAL Compiler Overview

Since the Optimizing SISAL Compiler (OSC) was originally designed for shared-memory multiprocessors, it had to be extended to satisfy the requirements imposed by MPP architectures. This section provides an overview of the SISAL system, discusses MPP architecture requirements, and how they can be accommodated by distributed-memory SISAL.

2.1 A Tour of the SISAL System

A large portion of the Optimizing SISAL Compiler is devoted to converting a SISAL program into data flow graphs, and improving these graphs with a variety of machine-independent optimizations. These sections of the compiler are described in detail in [Ca92a]. The discussion here will focus on the lowest level portions of the SISAL system: the run-time system and code generator.

2.1.1 SISAL's Run-Time System

SISAL's Run-Time System (SRTS, pronounce like "certs") is written in the C programming language, and provides services used by all SISAL programs. SRTS provides a `main()` function for stand-alone SISAL programs (programs written entirely in SISAL), structured sequential I/O, memory management, task scheduling, and performance monitoring.

The function `main()`, used by stand-alone SISAL programs, processes command-line arguments, initializes SRTS, reads program inputs, and then calls the user's SISAL program, which is C code generated by the SISAL back-end. After the program executes, `main()` generates performance diagnostics, and prints the program results.

Because SISAL provides a rich set of data types, representing program inputs and outputs is difficult. To make this easier, SRTS implements a language called FIBRE for expressing the inputs and outputs of a SISAL program [Ca92b]. FIBRE is used to describe all possible SISAL values, which can be scalar values (character, integer, floating point), arrays of scalar values, strings (arrays of characters), arrays of arrays (to support multi-dimensional arrays), unions, and records. Since user-defined data structures vary from program to program, FIBRE routines provide primitive I/O operations for use by compiler generated routines for user-defined data. Since SISAL is a purely functional, and deterministic language, all SISAL I/O is performed sequentially; inputs are read at the start of the program, and outputs are written at the end.

When SRTS is initialized, (by default) a large block of memory is allocated for the exclusive use of the running SISAL program. If the program needs more memory than is allocated at startup, the run-time system signals an error and terminates the computation. In response to this, the user can specify that more memory be allocated with a command line option. Once this memory is allocated, it is never given back to the operating system, but is instead managed by the run-time system. Instead of deallocating unused memory, blocks of unused memory are placed on various free lists, so they can be reused quickly. This was done to increase the performance of memory management, as general purpose memory allocators provided by the host operating system were found to be too slow.

A running SISAL program consists of sequential sections that run on only one processor (the master), and parallel sections that run on all available processors (the workers). SISAL is an *implicitly* parallel language, and parallelism is realized in the form of loops in which all iterations can be executed independently. To execute a loop in parallel, SISAL "slices" the loop into subtasks which compute a fraction of the total number of iterations of the loop. The SISAL compiler specifies how loops are sliced, but the run-time system distributes loop slices to the workers as independent tasks. At the beginning of each parallel loop, these tasks are created by SRTS, and placed in a queue for the workers to execute. To help avoid memory contention (hot-spots), SRTS uses a separate queue for each worker. When the workers finish the tasks, they notify the master, which can proceed only after all tasks complete.

The simplest strategy for task allocation is that a loop of *n* iterations is divided up among *p* processors into *n/p* loop slices of contiguous loop indices, and each worker is given a single slice. If necessary, the user can request strided (non-contiguously indexed) and variable sized slices (e.g. using guided self-scheduling), by issuing appropriate command line options to the SISAL compiler and run-time system. The run-time system also supports multiple levels of loop slicing, where nested parallel loops are further sub-divided when there is sufficient parallel work to warrant it.

The above architecture requires that SRTS maintain shared data structures for memory management (the free lists) and task distribution (the work queues), as well as shared lock variables for synchronization. MPP architectures do not provide direct support for such shared data structures, and require that programs written in a shared-memory style be rewritten using explicit communications. This would require a major rewrite of SRTS, doing away with shared data structures, and using messages for all data exchange and synchronization. Since the data structures used to represent loop slices use pointers to the actual data needed to perform the computation, using messages rather than shared data structures would require copying this data into message buffers, and could present significant overhead. Moreover, a loop body might not touch all the elements of input data, thus the cost of communication might be wasted. What is needed is a way to support shared data structures on MPP machines[3] in a way that does not require such drastic changes and provides a degree of portability so that SISAL can be used on a wider variety of machines.

2.1.2 Back-End Code Generator

The Optimizing SISAL Compiler (OSC) translates SISAL source programs into a data-flow graph intermediate form called IF [Sk85, We86]. Each successive phase of the compiler reads in the IF representation, applies a series of optimizations and transformations, and then writes out the results for the next phase. The final phase of the compiler, known as IF2GEN, translates IF into C, which is then passed to the host operating system's C compiler.

IF2GEN generates a single C source file consisting of several distinct sections. The important sections are the file prologue, the functions and loop slice bodies, and the file epilogue. The file prologue contains `#include` directives, data type declarations, and array copy functions. The second major section contains code that implements the user's program, including functions to compute the loop slices. The file epilogue contains global data declarations and initializations, the function `SisalMain()` (if a stand-alone SISAL program), and, finally, FIBRE functions for reading and writing the user-defined data structures (records and unions).

The next section examines the requirements of programming for distributed-memory architectures, and suggests how these requirements can be met by an enhanced SISAL system.

[3]Cray has predicated the design of its T3D MPP on providing efficient support for shared data structures. Hopefully, this will become a trend. Tera's Multithreaded Architecture (MTA) goes even further.

2.2 Distributed-Memory Programming Model

Distributed-memory architectures provide a programming model that is fundamentally different from shared-memory architectures. In the distributed-memory model, processors can only communicate by explicitly exchanging messages and cannot access each other's memory directly. This explicit exchange of messages is programmed using "send/receive" communication primitives, which are provided in a library by the system manufacturer. A message passing program is structured so that for every send that occurs on one processor, a corresponding receive must eventually occur on the processor that is the target of the send, otherwise the program will never complete. A detailed discussion of message passing programming is given in [An91].

In contrast, in the shared-memory programming model, any processor can read or write arbitrary memory locations, independently of other processors, with the trade-off that consistency (by mutual exclusion) must be explicitly programmed using synchronization primitives (semaphores, lock variables, for atomic operations). Message passing programming does not require explicit synchronization, as the send/receive operations provide implicit "rendezvous" synchronization [An91].

Although the power of the two models is equivalent, it is generally easier to implement message passing in terms of shared-memory operations (via message queues and locks), rather than vice-versa. However, since distributed-memory architectures do not support shared-memory operations in hardware, programmers are usually required to use message passing on these machines.

Another issue that arises in the message passing model is the lack of a standard message passing library that runs on multiple MPP platforms. Each vendor provides its own implementation, designed especially for its platform. While all message passing systems are similar, the details of each has made portability difficult. Recently, two standards have been gaining acceptance, the older PVM, and the newer MPI, but their use is not universally accepted, as they tend to under-perform the vendor-supplied libraries (performance still matters).

Still, despite the proliferation of libraries for message passing programming, the shared-memory programming model, if it can provide reasonable efficiency, tends to be preferred by most programmers (and it is the model that SISAL was originally designed for). One reason for this is that many algorithms are easier to write (and understand) for shared-memory machines, especially those problems which are most naturally represented using linked data structures (e.g. graph algorithms). This is important – no matter how fast a machine is, how easy it is to program is a serious consideration.

Perhaps MPP manufacturers should concentrate on providing communication architectures on which either programming model can be built. This was asserted in [Ei93] as the rationale for the design of a new communications architecture known as "active messages" which supports both message passing and shared-memory programming models, and is designed specifically for MPP machines. Active messages has formed the basis of work on new programming languages that hide the details of interprocessor communications, and provide programmers with a unified shared-memory view[4].

2.3 Implications for SISAL

Since the SISAL system was originally designed for shared-memory hardware, the above discussion suggests several approaches for targeting distributed-memory hardware:

[4]Active messages are now supported directly by Thinking Machines' CM-5.

1. Use a standard message passing library (e.g. PVM or MPI) to perform all interprocessor data movement.

2. Implement and use shared-memory operations on top of message passing.

3. Retarget the compiler to a higher-level language that provides shared-memory operations on distributed-memory hardware, based on active messages.

The first approach would have required drastic changes to the existing SISAL source code. To use straight message passing, every shared-memory pointer dereference would have to be replaced with send/receive operations. This large a change would result in a version of the SISAL system that would have to be maintained separately from the shared memory versions. The ability to incorporate the changes for MPP machines into the main sources enhances the maintainability of the system.

The second approach, implementing a shared-memory abstraction on top of a message passing system, has been explored in [Ha93a]. Their VISA system used software address translation to map virtual addresses on to a message passing system. This technique gave the flexibility of creating replicated and distributed data structures, but proved to be too costly due to the software address translation for every memory reference.

We chose the third approach – to retarget the SISAL compiler to a new dialect of C, Split-C, developed at U. C. Berkeley, that provides shared-memory operations at the language level [Cu93]. This choice lets us transport SISAL easily to any platform that supports Split-C. Currently, Split-C runs on the TMC CM-5, the Meiko CS-2, the Cray T3D, and Networks of Workstations (NOW), and more platforms are on the way. Since each of these platforms has very different communications architectures (from the CM-5's custom hardware interconnects to NOW's TCP/IP over ATM networks), Split-C provides abstracts the underlying communications hardware, but provides a powerful programming model.

The next section describes the Split-C programming language and the support it provides for distributed shared-memory.

2.4 Distributed Shared-Memory

SISAL's run-time system (SRTS) manages a variety of shared data structures in order to coordinate the parallel execution of a SISAL program. Therefore it is important to provide efficient shared-memory support so that the run-time system does not become a bottleneck to good performance. The following sub-sections describe how Split-C supports a shared-memory abstraction on distributed-memory hardware.

2.4.1 The Split-C Language

Recently a group at U. C. Berkeley, led by Dr. David Culler, created Split-C, a dialect of the C programming language for distributed computing [Cu93, Lu94]. Split-C maps a shared-memory address space on to the multiple memories of a distributed-memory computer. This mapping provides a two-dimensional view of distributed-memory: the first dimension indicates which processor a memory location resides on, and the second gives the actual address within a given processing element's memory. Split-C gives the programmer ultimate control over how memory will be allocated to a program's data structures, so that parallel algorithms can be tuned to keep memory accesses local as much as possible. Split-C provides a fairly low-level programming model, but its powerful primitive operations make it well-suited as a target for compiling high-level languages such as SISAL.

One important benefit Split-C provides is that it imposes no performance penalty (over regular C) when a computation accesses only local data[5]. This is accomplished by extending C's type system with two new pointer types: *global pointers* (declared as `type *global identifier`) extend the programmer's reach by providing a way to access memory locations on *any* processor; and *spread pointers* (declared as `type *spread identifier`) support data structures that are distributed across multiple processors. Split-C retains the semantics of regular C pointers, to address local memory locations.

An important constraint of MPP architectures is that remote memory operations can take between 10 to 1000 times longer than local memory operations. Two major techniques have been developed to tolerate these longer latencies: multiple threads of execution can be used to keep a processor busy during high-latency operations, suspending the current thread when remote memory operations initiate and resuming when they complete[6]; software pipelining transforms programs by moving non-blocking communications to an earlier point in a program, so that computations using local data can be performed concurrently with communications. The limiting factor in multi-threading is how many threads a program can be divided into, keeping the processor sufficiently busy[7].

Split-C directly supports the second technique by providing non-blocking, "split-phase" communications. A new assignment operator, :=, provides asynchronous put/get operations. If a global pointer is dereferenced on the left-side of :=, we have a put, otherwise a get. These operations are weakly ordered,

[5]Starvation is still a problem, however. See §2.4.2 for a discussion of polling.

[6]This has spawned a variety of new architectures that support fast thread switching in hardware, such as MIT's Alewife project, and Tera's MTA.

[7]This approach has been used in another implementation of distributed memory SISAL [An95] – see §4.3 for a comparison.

so a new language statement, `synch`, is provided to force the completion of all pending puts and gets.

2.4.2 The Split-C Programming Model

Split-C provides a Single-Program-Multiple-Data (SPMD) programming model. This is a hybrid of the Single-Instruction-Multiple-Data (SIMD) and Multiple-Instruction-Multiple-Data (MIMD) parallel programming models. Although it can be shown that SIMD machines can emulate MIMD machines, and vice versa [Hi85], the MIMD model gets better processor utilization when executing non data-parallel programs. In the SPMD model, all processors load the same program image, but, unlike SIMD machines, the instruction streams are not synchronized across all processors. Moreover, SPMD programs can emulate MIMD programs, by choosing the instructions to execute as a function of the processor number.

While Split-C emulates shared-memory on distributed-memory hardware, the emulation is incomplete in some respects, and goes beyond shared-memory in others. Split-C is an incomplete emulation of shared-memory on two counts: 1) global objects are not shared, sharing is only done via global pointers to objects (this is not really a problem, but it must be understood to use Split-C effectively); 2) it is possible for processor *i* to starve processor *j* if the processor *i* is involved in only local computation (no global pointers are in use), and processor *j* attempts to dereference a global pointer that refers to processor *i*'s memory. This second limitation can be alleviated if processor *i* periodically polls for remote memory requests.

3 Implementing Distributed-Memory SISAL

The following sections describe in detail the implementation of distributed-memory SISAL (DMS). Section 3.1 details the changes made to get SISAL running correctly on distributed-memory, and section 3.2 describes the procedures used to verify the run-time system and compiler. Section 3.3 discusses some simple optimizations that were tried to improve distributed-memory performance. (Section 4 contains an extended discussion of future work.)

3.1 Getting SISAL Running

The following two sub-sections illustrate the components of the Optimizing SISAL Compiler (OSC) that were changed for running on distributed-memory architectures. These are respectively, SISAL's run-time system, and code generator.

3.1.1 Run-time System Changes

SISAL's Run-Time System (SRTS), described in §2.1.1, provides support for the execution of SISAL programs in parallel. It provides sequential formatted I/O, memory management, synchronization mechanisms, and task management.

Transition to SPMD Execution

The first major change to SRTS was the conversion to an SPMD execution model. Earlier versions of SRTS were designed to begin execution of a parallel program on a single processor, which "spawns" the worker processor programs. Split-C's SPMD execution model requires that SRTS run on all processors, symmetrically. SRTS now begins execution at a new main entry point, `splitc_main()`, which processes command-line arguments and initializes the run-time system on all nodes.

The effects of the SPMD programming model change rippled throughout SRTS, requiring many changes, and permitting a few improvements. In many cases SRTS maintains arrays of data structures, one element per processor, such as the work queue. In the old run-time system, these arrays had to be initialized sequentially, by the master processor. Split-C supports distributed arrays, in which element *i* resides on processor *(i mod p)*. In the new run-time system, these arrays are initialized in parallel, which is not only simpler, but faster. Listing 3.1 shows an example of parallel initialization in which each instance of the runtime system is responsible for initializing its portion of a distributed data structure.

```
void InitReadyList()
{
#ifdef DISTMEM_MPP
    /* initialize in parallel & locally. */
    MY_INIT_LOCK(&ARList[MYPROC].Mutex);
    ARList[MYPROC].Head = 0;
    ARList[MYPROC].Tail = 0;
    WorkAvailable[MYPROC] = FALSE;
#else
    register int Index;

    ARList = (ActRecCachePtr)SharedMalloc(
                  SIZEOF(struct ActRecCache) * NumWorkers );

    for ( Index = 0; Index < NumWorkers; Index++ ) {
      MY_INIT_LOCK( (&(ARList[Index].Mutex)) );
      ARList[Index].Head = 0;
      ARList[Index].Tail = 0;
    }
#endif
}
```

listing 3.1: Parallel initialization of the distributed task queue

Global Pointers Everywhere

Split-C global pointers support the illusion of shared-memory on distributed-memory architectures by extending the semantics of C pointer dereferencing to include interprocessor communication. To explain how global pointers affect

SRTS, we will examine how they are implemented. Global pointers can be represented in regular C as a struct, as shown in listing 3.2.

```
struct GlobalPointer {
      int proc;         /* processor number */
      void *addr;       /* local address */
};
```

listing 3.2: C representation of a global pointer

When a global pointer is dereferenced, the Split-C compiler implicitly generates a test to determine if the pointer represents an on-processor location, and if not, a call to an appropriate communication primitive. To illustrate, listings 3.3 and 3.4 show code for post-incrementing an integer referenced to by global pointer, in Split-C and in the equivalent C code.

```
int post_increment(int *global x)
{
      int old_x = *x;
      int new_x = old_x + 1;
      *x = new_x;
      return old_x;
}
```

listing 3.3: Split-C post-increment code

```
int post_increment(GlobalPointer x)
{
      int old_x = (x.proc == MYPROC ?
                    *(int*)x.addr : __i_read(x.proc, x.addr);
      int new_x = old_x + 1;
      if (x.proc == MYPROC)
            *(int*)x.addr = new_x;
      else
            __i_write(x.proc, x.addr, new_x);
      return old_x;
}
```

listing 3.4: Equivalent C post-increment code

SISAL-generated code is well-suited to use global pointers, because it makes extensive use of dynamically allocated data structures. Getting the SISAL run-time system running under Split-C required converting pointer variables to global pointer variables. To simplify this task, C typedefs were used whenever possible to change the meaning of pointer types from local to global. Many of the type names used by the run-time system were defined with C typedefs (originally macros) that defined the pointer to the type, such as the type defined for SISAL arrays shown in listing 3.5.

```
typedef struct Array *ARRAYP;
```

listing 3.5: Typedef for SISAL array data type

Listing 3.6 shows the change to the pointer type to use global pointers.

```
typedef struct Array *global ARRAYP;
```

listing 3.6: Typedef for global pointer to SISAL array

This change seems simple, but it exposed portability problems in the original run-time system. The pervasive assumption that pointers are the same size as integers, and the lack of ANSI C prototypes[8] for functions that accept pointers became a problem in the new run-time system, because global pointers are in general larger than local pointers (see listing 3.1), and hence global pointers are larger than integers. Since there were no function prototypes in the original sources, global pointers would be implicitly truncated to integer size, essentially corrupting their values. Creating prototypes for all exported functions solved this problem. However, there were also numerous instances of pointers to built-in C data types (i.e. `char*`), that had to be converted by hand to (`char *global`).

Memory Management

The original run-time system was written for systems with a single shared address space. This assumption permitted the use of a simple memory management algorithm: pre-allocate a large block of memory, allocate memory by incrementing a pointer into this block, and deallocate by placing free blocks into "free lists," which are searched using a best-fit or first-fit criteria for

[8]Prototypes are generally placed in C header files, and provide a way to ensure consistent calling conventions between the caller of a function and the function itself.

subsequent allocations. Unfortunately, this type of memory management algorithm is not well-suited to distributed-memory machines, because it assumes all memory can come from a central pool. This centralized pool is impractical due to the limited amount of memory available in each node. By allocating memory on all nodes larger problems can be solved by using more nodes. A central pool would also be a performance bottleneck.

The first version of the distributed run-time system used a simple adaptation of this algorithm, except that a block of memory was allocated on each processor, and an independent local pointer was used by each processor to keep track of allocations. This initial design did not support deallocation, so it was limited in the size of problems that could be solved. (§3.2 discusses simple improvements in the design of the memory allocation that removed this restriction.)

Distributed Synchronization

Split-C provides barriers and atomic operations (`test_and_set`, and `fetch_and_add`) for performing synchronization. While these mechanisms are required by SRTS, they are much more expensive to use in a distributed-memory system, and in many cases their use can be minimized. (See §3.2 for a discussion of why their use should be minimized and how this was accomplished.)

3.1.2 Code Generation Changes

The Optimizing SISAL Compiler (OSC) is a multiple pass compiler that performs a wide variety of optimizations for producing efficient code. The final pass of the compiler, IF2GEN, generates code in the C programming language. This section describes how IF2GEN was modified to generate Split-C compatible code.

Global Pointer Conversion

The same pointer conversions that took place in the run-time system were required in the compiler generated code, so typedefs were used to create pointer types that encapsulate the use of global pointers. IF2GEN generates C structures to represent the parameter lists of every function in a SISAL program. Since pointers to these parameter structures are used extensively in the generated C code, these had to be converted to global pointers. Rather than changing instances of (`struct Params *`) to (`struct Params *global`) everywhere in the compiler, typedefs were generated so that (`ParamsPtr`) could be used instead.

Another vital piece of code used by every SISAL program was the C header file "`sisal.h`." IF2GEN relies on C macros contained in this file, which define a simple abstract machine code. A benefit of this design is that changes can be made to the operation of the generated code by editing this header file without modifying the compiler[9]. This header file also had to be modified to be Split-C compatible. Many macros were of the form shown in listing 3:7.

```
#define AddOp(type, dest, op1, op2) \
      *(type*)dest = *(type*)op1 + *(type*)op2
```

listing 3.7: Macro for addition of any types

Given this macro definition, the statement `AddOp(float, &x[i], &y[j], &z[k]);` results in the floating-point addition of the `j`-th element of array `y` to the `k`-th element of `z`, and stores the results in `x[i]`. When the statement is compiled by Split-C, the operands can come from anywhere in distributed-memory, so the (`type*`) cast is not correct in general. Thus, for every macro of this form, all instances of (`type*`) had to be converted to (`type *global`). This code is perhaps overly general, because it imposes some additional tests for every pointer operation (see listing 3.3), but it is correct.

[9]A drawback is that the resulting C code is often impenetrable, and hard to debug.

Structure Copying

A problem with Split-C occurs when using global pointers on both sides of an assignment statement, as in listing 3.8.

```
struct stype { int x, y; };
struct stype s1, s2, *global gp1, *global gp2;
gp1 = &s1;
gp2 = &s2;
*gp1 = *gp2;
```

listing 3.8: Typedef for global pointer to SISAL array

In any given statement, Split-C only supports either one or more global pointer reads from, or a single global pointer write. Many macros in "`sisal.h`" contained this type of structure copying code. To produce correct code under Split-C, a local variable must be used to read from the first global location, and then the local copy is written to the second global location. This transformation is so straightforward, that the Split-C compiler could be easily modified to do this automatically.

Run-time Global Initialization

IF2GEN creates complex global data initialization statements in which elements of structures are initialized to addresses of other global variables. Split-C does not support the compile-time initialization of global pointers, and so this initialization had to be moved to run-time. Fortunately, a routine that initializes global data (`InitGlobalData()`) was already being generated – additional code to perform the global pointer initializations was added to this routine.

3.2 Compiler Verification

The correctness of the new compiler implementation was verified empirically by running a set of standard test programs. The simplest parallel program used is shown in listing 3.9.

```
define main

function ArrayOfN(n : integer returns array[integer])
  for i in 1,n
  returns array of i
  end for
end function

function main(n : integer returns array[integer])
  ArrayOfN(n)
end function
```

listing 3.9: Simple SISAL program to build an array in parallel

This program was chosen for its extreme simplicity and for the ease of understanding the resulting C program generated by the compiler. This permitted the hand modification of the compiler output in order to verify the correctness of the run-time system, and to guide the modifications to the code generator.

Once the run-time system was working, the compiler was modified to generate code identical to the hand-modified C code. The next step was to choose a more ambitious test program, which uncovered latent bugs in the run-time system and code generator. This iterative process of choosing more and more complicated test programs worked well and progress was steadily made toward a fully functional system.

Another way to ensure the correctness of the compiler was to get it running on a completely different platform. As soon as the compiler was stable enough, it was ported from the CM-5 to the Meiko CS-2, which helped to uncover additional bugs.

3.3 Optimizations for Distributed-Memory

This section examines a few simple optimizations to the distributed-memory SISAL run-time system, after it was running correctly. §4.1 analyzes the benefits of these optimizations and suggests future work.

3.3.1 SISAL Run-Time System Optimizations

SISAL's run-time system contains several shared data structures that were considered for simple optimizations. These are shared signal variables, the work queue, and the memory manager.

Replicated Condition Variables

The run-time system uses shared variables to control the execution of a SISAL program. The master processor uses shared boolean variables to signal the workers when work is available, and when to exit the program. The initial version of the distributed run-time system allocated these condition variables on the master processor, requiring all other processors to read them from across the network. This generated network traffic, but produced no useful work.

This type of traffic, caused by continual polling, can be reduced to nothing by taking advantage of the fact that this is one-way communication only, from the master to the workers. If these condition variables are replicated, and made local to each worker processor, all polling traffic can be eliminated, until the master sends the signal. This optimization was implemented by replacing each signal variable with a distributed array, and having each processor poll its local element. The master processor now signals a particular condition by looping over all the elements of the distributed array, and writing the appropriate value.

Distributed Task Queue

The first version of distributed-memory SISAL's run-time system (DM-SRTS) used a central work queue, located in the master processor's memory. Worker processors would periodically inspect this queue to see if any work is available. Before any worker could safely inspect the queue, it would first have to lock the queue by performing an atomic update of a shared lock variable (a global pointer initialized at startup of the run-time system). After obtaining the lock, the worker would then examine the contents of the queue, and take an item of work (if available), and then unlock the queue by clearing the lock variable.

This design had several deficiencies, which are compounded by a distributed-memory architecture. First, having a centrally located queue means that the master processor has to service every request (via polling) to permit the worker processors to examine the lock variable and queue. Second, a large amount of network traffic is generated even when there is no work to do! Two enhancements were made to this design.

The first enhancement was to transform the central work queue into a distributed array of queues, one per processor. Each worker then examines its own work queue for work to do, and generates no network traffic when no work is available. The master processor puts work into the workers' queues, rather than a worker getting work from the master. The second enhancement was to add a replicated signal variable to indicate the availability of work, to save the workers' having to examine their queues when no work is available. This allows the workers to avoid having to lock their queues until there is something in them. To ensure consistency, the master only asserts the condition while it holds the worker's queue lock, and the workers only clear the condition while they hold the lock.

Distributed Memory Management

To correct the memory management limitations of the first version of DM-SRTS, a new algorithm was developed. This algorithm, designed for Split-C, uses the standard C library routines `malloc()` and `free()` to perform memory management on each node. When one processor (the allocator) allocates a block of memory on behalf of another processor, the pointer is converted[10] to a global pointer. When a processor is finished with remotely allocated memory, it places the pointer in a free list located in the allocator's memory. The next time the allocator needs more memory, it first checks the free list to see if any blocks have been placed there, and calls `free()` on them before trying to perform new allocations. This should give acceptable performance as reasonable implementations of `malloc()` and `free()` will coalesce adjacent free blocks to minimize heap fragmentation. This is also beneficial in that it takes platform-specific knowledge out of SRTS, which enhances portability.

[10]Split-C pointers can be converted from local to global by type casting.

4 Results & Discussion

This section will provide a preliminary examination of the performance of distributed-memory SISAL, on a small set of parallel programs from the SISAL literature. The programs have been run with versions of the run-time system with the simple optimizations discussed above, to measure their effectiveness. The section closes with a comparison to related work, suggestions for future work, and conclusions.

4.1 Performance Studies

The greatest difficulty in producing meaningful performance measurements is choosing problems that are simple enough to fully analyze, and yet representative of real problems. For this reason, problems were chosen from the SISAL literature that are readily understandable, and yet are similar to more complex problems. The first problem is an implementation of John Conway's famous cellular automaton, Life, which simulates an abstract biological system. The second problem, an iterative Laplace heat equation solver, was used in [Ha93b] to study the performance of another implementation of distributed-memory SISAL. The final problem studied was matrix multiplication, which exhibits a large amount of parallel work, but has pessimal communication patterns.

4.1.1 Performance Studies of Life

The SISAL program for Life is included in the appendix, §A.1. The high-level algorithm used in the Life program is as follows:

```
I. Create a random starting configuration of R rows and C columns. This
consists of 1's and 0's placed randomly in an RxC grid. 1 means a cell is
alive, and 0 means a cell is dead (or absent). Pad the grid with 0's all
around the perimeter (thus grid is actually (R+2)x(C+2).
II. Compute the next generation according to the following rules:
    A. if a cell is alive, and >5 of its neighbors are alive, the cell dies
    from overcrowding.
    B. if a cell is absent, but has >=3 neighbors, a cell is born.
    C. otherwise, nothing happens.
III. Repeat for N iterations.
```

If $R = C = 10$, and $N = 2$, and we have the initial random configuration shown in the left half of figure 4.1, the final configuration will be that shown in the right half.

0	0	0	0	0	0	0	0	0	0	0	0
0	1	1	0	1	0	1	0	0	0	0	0
0	1	0	0	1	1	1	0	0	0	0	0
0	0	1	0	0	1	1	1	0	1	0	0
0	1	1	0	0	0	1	1	1	0	1	0
0	1	0	0	0	1	0	1	0	1	0	0
0	0	1	0	0	1	0	0	0	1	0	0
0	1	1	0	0	1	1	1	1	0	0	0
0	1	1	1	0	0	1	0	1	0	0	0
0	0	1	0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	1	0	1	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

0	0	0	0	0	0	0	0	0	0	0	0
0	0	1	1	1	0	1	0	0	0	0	0
0	1	0	0	0	1	1	1	1	0	0	0
0	1	0	0	0	1	1	1	1	0	0	0
0	1	0	0	1	1	0	1	1	1	0	0
0	1	1	1	1	1	1	0	0	1	0	0
0	1	0	0	1	1	0	0	0	0	1	0
0	1	0	0	0	0	0	0	1	0	0	0
0	1	0	0	0	0	0	1	1	0	0	0
0	1	1	1	1	1	1	0	0	0	0	0
0	0	1	0	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0	0	0	0	0

figure 4.1: Life evolution, 10x10 initially, and after 2 iterations

If we study the `Generate` function in section A.1, we see that it implements step I of the algorithm. Although each row is built sequentially (`for initial` loops are sequential), the outer loop, ranging over the number of rows, gathers the results of each inner loop (which is a single row) into an array of arrays, allowing the rows to be built in parallel. On a system with W workers, the SISAL run-time system will give each worker R/W rows to create. Since SISAL represents two-dimensional arrays as "arrays of arrays", the resulting grid will be built as a distributed data structure, with the rows created by a given worker residing in that worker's memory.

The `DoWork` function implements step II of the algorithm. As in the `Generate` function, the outer loop traverses over all the rows, and the inner loop computes new rows. Here the inner loop is not constrained to be sequential, if there were enough processors, we would assign a single processor to compute each new cell, and compute successive generations in constant time. By default, though, the compiler only slices outer parallel loops, thus the rows will be traversed in `DoWork` by the processor that created them in Generate[11].

Tables 4.1 and 4.2 show the running times and speedups for Life running on a 64-node CM-5, for various problems sizes from 100x100 up to 1000x1000. The speedups are based on the single-node performance of the Split-C runtime system on the CM-5, as no sequential SISAL implementation was available.

	Problem Size:	100x100	Problem Size:	250x250
PEs	Time (sec)	Speedup	Time (sec)	Speedup
1	5.33596	1.0	32.9267	1.0
4	2.01455	2.6487	11.5079	2.8612
8	1.00511	5.3088	5.22836	6.2977
16	0.581456	9.1769	2.73545	12.037
32	0.447704	11.9185	1.85134	17.785
64	0.349898	15.250	1.16183	28.3404

table 4.1: CM-5 Life performance for 100x100 & 250x250 grid size

[11]This is not completely correct, as each processor will access a single row above and below their slice boundaries, but this is unavoidable.

PEs	Problem Size:	500x500	Problem Size:	1000x1000
	Time (sec)	Speedup	Time (sec)	Speedup
1	145.592	1.0	574.0	1.0
4	49.639	2.93	196.114	2.93
8	22.2091	6.5555	85.1081	6.74
16	10.8963	13.3616	40.8928	14.037
32	6.56673	22.1712	24.9315	23.02
64	3.9476	36.881	14.4458	39.74

table 4.2: CM-5 Life performance for 500x500 & 1000x1000 grid size

This problem has good locality, and reasonable speedups. Figure 4.2 shows the speedups curves for the data given in tables 4.1 and 4.2.

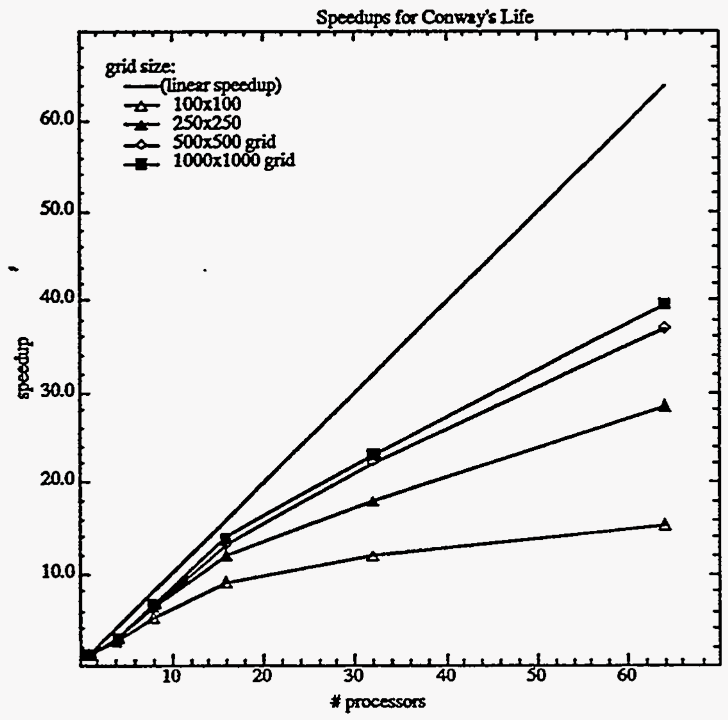

figure 4.2: Life speedups for 64 processor CM-5

This machine could not run the 1000 square problem size on a single node, so that result is extrapolated from an average of 2.93 speedup for 4 processors.

The plot shows a slightly super-linear speedup between 4 and 8 processors for the 1000x1000 grid size. This can be explained by better cache performance as the amount of data per node is reduced. Efficiency goes up as problem size increases, not surprisingly, as there is more work for each processor to do that is local.

The same program was run on a Meiko CS-2 for the same problem sizes, on 32 processors. Tables 4.3 and 4.4 give the results. While the Meiko gives better absolute performance (approximately 3-5 times faster) for the same program and run-time system (no source code changes to either), the speedup curve in figure 4.3 shows much lower efficiency.

PEs	Problem Size: 100x100 Time (sec)	Problem Size: 100x100 Speedup	Problem Size: 250x250 Time (sec)	Problem Size: 250x250 Speedup
1	1.57881	1.0	9.75142	1.0
4	1.50254	1.050761	7.12026	1.369531
8	0.868915	1.816990	3.88358	2.510936
16	0.699814	2.256042	2.38497	4.088697
32	0.598727	2.636945	1.73075	5.634216

table 4.3: Meiko Life performance for 100x100 & 250x250 grid size

PEs	Problem Size: 500x500 Time (sec)	Problem Size: 500x500 Speedup	Problem Size: 1000x1000 Time (sec)	Problem Size: 1000x1000 Speedup
1	70.0468	1.0	153.821	1.0
4	29.1789	2.400598	111.18	1.383531
8	14.2844	4.903727	54.0704	2.844828
16	8.33382	8.405125	28.1333	5.467578
32	4.86964	8.405125	16.2469	9.467714

table 4.4: Meiko Life performance for 500x500 & 1000x1000 grid size

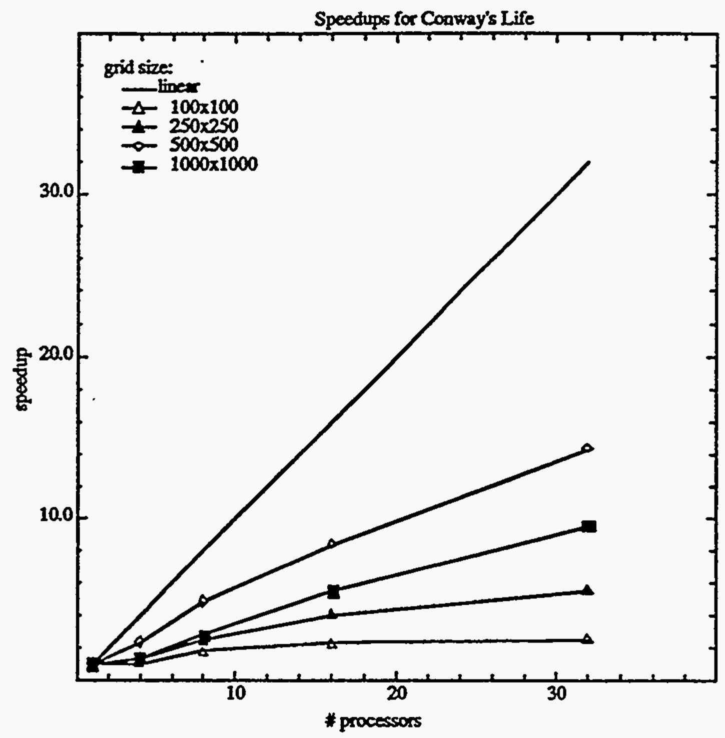

figure 4.3: 32 processor Meiko CS-2, 2 iterations

4.1.2 Performance Studies of Laplace

The Laplace program (given in the appendix §A.2) was used to study the performance of another implementation of distributed-memory SISAL in [Ha93b]. An interesting puzzle came up while studying this program. If we compare Laplace with Life, they have very similar structure. However, the performance of the original program as given in [Ha93b] is very different from Life. Simply put, Life goes faster with more processors, whereas Laplace actually slows down.

The original Laplace program from [Ha93b] is provided in §A.2.1. This version of the program does not exhibit parallel speedup, even when compared to its sequential running time with the distributed run-time system. Table 4.5 shows this program's dismal performance on a 256x256 grid.

PEs	Matrix Size: 256x256 Time (sec)	Speedup
1	21.406	1.0
4	107.771	0.198625
8	105.969	0.20200
16	108.416	0.19744
32	106.702	0.20062

table 4.5: Original Laplace performance for 256x256 grid size

Several simple transformations to the program were attempted, with little success. However, when the program was changed to take advantage of the constant boundary conditions of the problem, we find that the first and last rows and first and last columns are in fact loop invariants. This observation lets us remove the test

```
if (I=1 | I = N | J=1 | J=N) then
```

from the inner loop of the Laplace function. This simple optimization was enough to make the difference between no parallel speedup and actual speedups. The results are shown in tables 4.6 and 4.7.

	Matrix Size:	256x256	Matrix Size:	512x512
PEs	Time (sec)	Speedup	Time (sec)	Speedup
1	13.9495	1.0	64.7404	1.0
4	9.15779	1.523239	29.8246	2.170705
8	6.05997	2.301909	19.6748	3.290524
16	4.33537	3.217603	12.237	5.290545
32	3.69366	3.776606	9.38018	6.901829

table 4.6: Modified Laplace performance for 256x256 & 512x512 grid size

	Matrix Size:	1024x1024
PEs	Time (sec)	Speedup
1	223.66	1.0
4	112.216	1.993120
8	61.8501	3.616162
16	41.4315	5.398308
32	25.1166	8.904868

table 4.7: Modified Laplace performance for 1024x1024 grid size

Comparison to Single Node SISAL

The Laplace program was also compiled with a single node version of SISAL on the Meiko, and the distributed version was able to outperform the single node version using more than 16 processors. A 1024x1024 grid (10 iterations) took 19.3 seconds on 64 processors, 25.6 seconds on 32 processors, 38.4 seconds on 16 processors, and 38.4 seconds on single node SISAL.

4.1.3 Performance Studies of Matrix Multiplication

Matrix multiplication is a very important parallel application that does not always perform well on distributed-memory machines. A simple formulation of the algorithm in SISAL appears in §A.3.1. This is a naive way to perform this algorithm in SISAL, because traversing a single column is much slower than traversing a row, because of the representation of two-dimensional arrays in SISAL. A very simple change, pre-transposing the right hand matrix (§A.3.2), allows the elements of the matrices to be traversed row-wise, and gives much better performance, as the results in tables 4.8 and 4.9 show (kmax = 1).

	10x10	50x50	100x100
PEs	Time (sec)	Time (sec)	Time (sec)
1	0.0165464	1.6049	12.6117
4	0.131157	15.3429	115.951
8	0.105396	12.221	97.2437

table 4.8: Matrix Multiplication without pre-transposition

	10x10	50x50	100x100
PEs	Time (sec)	Time (sec)	Time (sec)
1	0.0063352	0.0602764	0.222308
4	0.0167822	0.341258	1.24258
8	0.017565	0.289028	0.963906

table 4.9: Matrix Multiplication with pre-transposition

Although running the algorithm sequentially gives better performance in both cases, the speedups of the pre-transpose algorithm are striking. These are summarized in table 4.10.

	10x10	50x50	100x100
PEs	Speedup	Speedup	Speedup
1	2.61182	26.625678	56.730752
4	7.815245	44.959825	93.314716
8	6.000342	42.283101	100.885045

table 4.10: Relative speedups using pre-transposition

Other algorithms exist that perform better than this naive matrix multiply. These algorithms divide each matrix into sub-matrices, which are distributed among more processors (up to n^2 processors can work on the matrix rather than n processors) [Ku94]. These algorithms can and should be adapted to SISAL, because they should perform better on distributed-memory machines.

4.2 Related Work

There have been several distributed-memory SISAL projects. [Ha93a] describes an implementation of SISAL for the nCube, which uses a software-based virtual addressing scheme on top of message passing, to provide a global shared address space for OSC. They also extend the run-time system with multi-threading and hierarchical task distribution, techniques which allow the run-time system to adapt itself to the characteristics of a particular machine, or application.

[Pa93] describes retargeting SISAL to Intel Touchstone i860 systems (Gamma, Delta and Paragon). They argue that loop-level parallelism is insufficient to employ the high degree of parallelism available on distributed-memory machines, and concentrate on scheduling algorithms for functional parallelism. Their work involved modifying the phase of the compiler responsible for parallelizing SISAL programs, and adding scheduling of functional parallelism. They also adapted the run-time system to use message passing to perform interprocessor data exchange.

In [An95], a more recent effort is discussed which uses a combination of fine-grained parallelism (stackless threads called "filaments"), with virtual memory hardware initiated communication (i.e. when a page fault occurs, if the page is "remote" it is requested from the "owner" of the page). The latency of a page fault is tolerated by providing lots of fine-grained threads to do work during communication. This approach is limited by the inherent parallelism of a problem. Also, the use of virtual memory "protected" pages is not as portable as using Split-C to provide a global address space. In fact, an implementation of Split-C uses this scheme to provide global addressing on networks of workstations (NOW).

We are not aware of other work that enjoys the same portability as our distributed-memory SISAL built upon Split-C. The use of Split-C as an abstract machine insulates our compiler from the multitude of incompatibilities between different distributed-memory architectures.

4.3 Conclusions & Future Work

This project has laid the foundation for the development of potentially higher performance MPP SISAL implementations. The work has concentrated on the restructuring of the run-time system, because these improvements benefit all programs.

The portability of distributed-memory SISAL has been demonstrated. SISAL now runs on the Thinking Machines CM-5 and the Meiko CS-2. Once the CM-5 version was working and fully debugged, the Meiko version was up and running within two days. As soon as Split-C compilers appear on other platforms, distributed-memory SISAL will soon follow.

More improvements to the run-time system can and should be made. An important enhancement will be the dynamic allocation of one-dimensional arrays as distributed spread arrays. This will help to reduce communications when an array is created by multiple workers; each worker will perform only local-memory writes. Spread arrays will also help to reduce memory contention that occurs when an array is read by multiple processors[12]. Since arrays of greater than one-dimension are implemented as arrays of pointers, a similar optimization can be performed for these arrays, by distributing outer dimension arrays, which contain pointers to the inner dimension arrays.

Another run-time system improvement will involve the mechanism for distributing work to the workers. Loop slices are represented in a data structure called the "activation record" (listing 4.1).

```
struct ActRec {
    GLOBAL_POINTER ArgPointer;   /* TASK ARGUMENT */
    int AuxArgument;             /* AUXILIARY TASK ARGUMENT */
    void (*ChildCode)();         /* TASK ADDRESS */
    int SliceBounds[3];          /* LOOP SLICE CONTROL INFO */
    ActRecPtr NextAR;            /* FORWARD QUEUE LINK */
    int Done;                    /* IS THIS TASK DONE YET? */
    int pid;
    int Flush;
};
```

listing 4.1: Data type for activation records

The C data type `struct ActRec` contains all of the information necessary to execute a loop slice: the field `ArgPointer` contains a global pointer to an application-specific record containing the inputs to the slice, and the outputs produced by the slice; `ChildCode` holds a pointer to the C function representing the slice body; `SliceBounds` provides the lower and upper bounds of the loop,

[12]SISAL's single-assignment semantics also allows another optimization: data replication. However, the cost of the extra communication to do the replication might be too prohibitive.

and the loop stride. A simple improvement to the run-time system would be to pre-allocate activation records on the processor they will be used on. Then, when the master processor creates loop slices, the run-time system can use *put* operations to broadcast the activation records, asynchronously.

All of the improvements suggested above require the capability to allocate memory on all processors simultaneously. The Split-C library provides a function, `all_spread_malloc()`, which allocates spread arrays dynamically, but must be called by all processors, like a barrier. To utilize this, an additional phase will have to be added to the sequential-parallel execute cycle. This phase will be an "allocate" phase in which the master processor instructs the other processors in how many calls to `all_spread_malloc()` should occur, and with what parameters. In a sense, this is just another type of parallel task, but these tasks must be performed strictly sequentially and in synchrony.

Additional performance gains will be obtained by performing distributed-memory specific optimizations on the compiler generated code. Here are some possibilities:

• Cache/replicate read-only data structures, such as the `ArgPointer` field of slice activation records, to avoid repeated remote references. This has to be done in the compiler generated code because its layout is not known to the run-time system.

• Reorder communications and unroll loops so that communication and computation can be overlapped. This can be done by using optimization techniques being developed for the Tera MTA [Mi95].

• Improve local memory access performance by converting pointers to local addresses when possible. This is important because every global pointer reference imposes a test (as shown in listing 3.1) which results in branch pipeline stall. Several techniques can be used to address this problem: setting branch prediction bits; compiling different loops for local and remote memory accesses, hoisting the tests outside of the loops; generating optimal code at run-time, using dynamic code generation.

Bibliography

[Sc85] David H. Schaefer. "History of the MPP." In *The Massively Parallel Processor*. J. L. Potter, (Ed.), The MIT Press, Cambridge, Massachusetts, 1985, pp. 1-5.

[Sk85] Stephen Skedzielewski and John Glauert. IF1: An Intermediate Form for Applicative Languages. Manual M-170, Lawrence Livermore National Laboratory, Livermore, CA, July 1985.

[Hi85] W. Daniel Hillis. 1985. *The Connection Machine*. The MIT Press, Cambridge, Massachusetts. p. 24.

[We86] Michael Welcome, Stephen Skedzielewski, Robert Kim Yates, and John Ranelletti. IF2: An Applicative Language Intermediate Form with Explicit Memory Management. Manual M-195, Lawrence Livermore National Laboratory, Livermore, CA, December 1986.

[Ca92a] D. C. Cann. The Optimizing SISAL Compiler: Version 12.0. Technical Report UCRL-MA-110080, Lawrence Livermore National Laboratory, Livermore, CA, April 1992.

[Ca92b] D. C. Cann. SISAL: 1.2 A Brief Introduction and Tutorial. Technical Report UCRL-MA-110620, Lawrence Livermore National Laboratory, Livermore, CA, May 1992.

[Cu93] David E. Culler. Introduction to Split-C. Computer Science Division, University of California, Berkeley, CA, December 1993.

[Fe90] John T. Feo, David C. Cann, and Rodney R. Oldehoeft. A Report on the Sisal Language Project. *Journal of Parallel and Distributed Computing* **10**, (1990), 349-366.

[An91] Gregory R. Andrews. 1991. *Concurrent Programming – Principles and Practice*. The Benjamin/Cummings Publishing Company, Inc. Redwood City, California.

[Ei93] T. H. von Eicken. *Active Messages: an Efficient Communication Architecture for Multiprocessors*. Ph.D. thesis, Computer Science, Graduate Division, University of California, Berkeley, CA, 1993.

[Ha93a] Matthew Haines and Wim Böhm. On the Design of distributed-memory Sisal. Technical Report CS-92-144, Computer Science Department, Colorado State University, Fort Collins, CO, January 1993.

[Ha93b] Matthew Haines and Wim Böhm. A Virtual Shared Addressing System for Distributed Memory Sisal. In *Proceedings Sisal '93* , pp. 151-163, October 1993.

[Mc93] J.R. McGraw. Parallel Functional Programming in Sisal: Fictions, Facts, and Future. Technical Report UCRL-JC-114360, Lawrence Livermore National Laboratory, Livermore, CA, July 1, 1993.

[Pa93] Santosh S. Pande, Dharma P. Agrawal, and Jon Mauney. Sisal on Distributed Memory Machines. In *Proceedings Sisal '93* , pp. 134-150, October 1993.

[Lu94] Steve Luna. *Implementing an Efficient Portable Global Memory Layer on distributed-memory Multiprocessors*. Master's thesis, Computer Science Graduate Division, University of California, Berkeley, CA, May 1994.

[Ku94] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis. *Introduction to Parallel Computing*. The Benjamin/Cummings Publishing Company, Inc. Redwood City, California.

[An95] Gregory R. Andrews and Vincent W. Freeh. Fsc: A Sisal Compiler for Both Distributed and Shared-Memory Machines. In *Proceedings of High Performance Functional Computing* , pp. 164-172, April 1995.

[Fe95] John T. Feo, Patrick J. Miller, Stephen Skedzielewski, Scott M. Denton and Cindy J. Solomon. Sisal 90. In *Proceedings of High Performance Functional Computing* , pp. 35-47, April 1995.

[Mi95] Srdjan Mitrovic. Personal communication, 1995.

[Sa95] J. Sargent, S. J. Hooton and C. C. Kirkham. UFO: Language Evolution and Consequences of State. In *Proceedings of High Performance Functional Computing* , pp. 48-62, April 1995.

Appendix

A.1 SISAL Program: Life

This program is derived from an example given in [Ca92b], with minor corrections. The original version only functioned with square grids. The program given here does not include the random number package used to create the initial grid.

```
define Main, rans, ranf

%
% John Conways Game of Life. Values of the Grid are 0s or 1s.  A given
% cell has 8 neighbors. On each iteration, the cells are updated as
% follows:
%  -- If a cell contains a 1 and more than five of its neighbors contain
%     1s, then it should become a 0.
%  -- If a cell contains a 0 and from three to five of its neighbors
%     contain 1s, then it should become a 1
%  -- otherwise the value of a cell remains unchanged.
% The simulation of life iterates Iterations times. The boarder of the
grid
% is always zeros.
%
% Main(Iterations,Rows,Columns)
%


function Convert( Seed:OneDim returns integer, OneDim )
let
  Number, NewSeed := ranf( Seed );
  V := if Number < 0.5D0 then 1 else 0 end if;
in
  V, NewSeed
end let
end function

function Generate( Rows,Columns:integer returns Grid )
let
  Seed_Stream := rans(Columns,1);
  First := array_fill(0,Columns+1,0);
  Last  := array_fill(0,Columns+1,0);
  Core  := for i in 1,Rows
             Row := for initial
                      j := 1;
                      V,Seed := Convert( Seed_Stream[i] );
                    while ( j < Columns ) repeat
                      j := old j + 1;
                      V,Seed := Convert( old Seed );
                    returns array of V
                    end for;
```

```
           returns array of array_addl(array_addh(Row,0),0) % [0, Row,
0]
           end for
in
  array_addl(array_addh(Core,Last),First)
end let
end function

function Compute( G : Grid; I : integer; J : integer returns integer )
let
  Total := G[I-1,J-1] + G[I-1,J] + G[I-1,J+1] +
           G[I,J-1]        +       G[I,J+1]    +
           G[I+1,J-1] + G[I+1,J] + G[I+1,J+1];
in
  if ( Total > 5 ) then 0
  elseif ( Total >= 3 ) then 1
  else 0 end if
end let
end function

function DoWork( G:Grid; Rows,Columns:integer returns Grid )
let
  First := for i in 0,Columns+1 returns array of G[0,i] end for;
  Last  := for i in 0,Columns+1 returns array of G[Rows+1,i] end for;

  Core  := for I in 1, Rows
              Mid := for J in 1, Columns
                     returns array of Compute(G,I,J)
                     end for;
              Row := array_addl( Mid, 0 );
            returns array of array_addh( Row, 0 )
            end for;
in
  array_addl(array_addh(Core,Last),First)
end let
end function

function Main( Iterations,Rows,Columns:integer returns Grid,Grid )
let
  Gin := Generate(Rows,Columns);
in
  Gin, for initial
         Count := Iterations;
         G := Gin;
       while ( Count > 0 ) repeat
         Count := old Count - 1;
         G := DoWork( old G, Rows, Columns );
       returns value of G
       end for
end let
end function
```

A.2 SISAL Program: Laplace

This program was used in [Ha93b] to measure the performance of another distributed-memory SISAL implementation. §A.2.1 contains the original program. §A.2.2 is a slightly modified version that exhibits parallel speedup.

A.2.1 Original Laplace

This version of Laplace fails to exhibit parallel speedup in a distributed-memory setting. The next section provides a version that does.

```
% laplace0.sis
define main

type   OneD = array[double_real];
type   TwoD = array[OneD];

function TwoD_fill (N : integer returns TwoD)
  for I in 1,N cross J in 1,N
    el :=
      if (mod(I + J, 2) = 0) then
        double_real(1.0)
      else
        double_real(N)
      end if
    returns array of el
  end for
end function % TwoD_fill

function Laplace (Init_M : TwoD; N, KMax : integer returns TwoD)
  for initial
    K := 1;
    M := Init_M;
  repeat
    K := old K + 1;
    M :=
      for I in 1,N cross J in 1,N
        nM :=
          if (I=1 | I = N | J=1 | J=N) then
            old M[I,J]
          else
            old M[I,J] / double_real(2.0) +
            (old M[I-1,J] + old M[I+1,J] + old M[I,J-1] + old M[I,J+1])
/
            double_real(8.0)
          end if
        returns array of nM
      end for
    until K >= KMax
    returns value of M
  end for
end function % laplace
```

```
function main (N, KMax : integer returns TwoD)
  let
    M := TwoD_fill(N)
  in
    Laplace(M, N, KMax)
  end let
end function % main
```

A.2.2 Modified Laplace

This version of Laplace optimizes the original slightly – the conditional statements are removed from the inner loop of the Laplace function. This version gives parallel speedups. See §4.1.2 for a discussion of this result.

```
% laplace4.sis
define main

type   OneD = array[double_real];
type   TwoD = array[OneD];

function TwoD_fill (N : integer returns TwoD)
  for I in 1,N cross J in 1,N
    el :=
      if (mod(I + J, 2) = 0) then
        double_real(1.0)
      else
        double_real(N)
      end if
  returns array of el
  end for
end function % TwoD_fill

function Laplace (MIn : TwoD; N, KMax : integer returns TwoD)
  for initial
    K := 1;
    M := MIn;
  while (K < KMax) repeat
    K := old K + 1;
    MFirst := for J in 1,N returns array of old M[1,J] end for;
    MLast := for J in 1,N returns array of old M[N, J] end for;

    MInner :=
      for I in 2,N-1
        Row :=
          for J in 2,N-1
            el :=  old M[I,J] / double_real(2.0) +
                   (old M[I-1,J] + old M[I+1,J] +
                   old M[I,J-1] + old M[I,J+1]) / double_real(8.0)
          returns array of el
          end for;
      returns array of
        array_addh(array_addl(Row, old M[I, 1]), old M[I, N])
      end for;

    M := array_addh(array_addl(MInner, MFirst), MLast);
  returns value of M
  end for
end function % Laplace

function main (N, KMax : integer returns TwoD)
  let
    M := TwoD_fill(N)
  in
    Laplace(M, N, KMax)
  end let
end function % main
```

A.3 SISAL Program: Matrix Multiply

Two versions of the matrix multiply algorithm are given. The second is a simple modification of the first which pre-transposes the right hand matrix.

A.3.1 Simple Matrix Multiply

```
% mmult0.sis
define main

type TwoDim = array [ array [ double_real ] ];

function Gen( n : integer returns TwoDim, Twodim )
  for i in 1, n cross j in 1, n
  returns array of double_real(i)/double_real(j)
          array of double_real(i)*double_real(j)
  end for
end function % Gen

function Mmult( n : integer; A, B : TwoDim returns TwoDim )
  for i in 1, n cross j in 1, n
    c := for k in 1, n
           t := A[i,k] * B[k,j]
         returns value of sum t
         end for
  returns array of c
  end for
end function % Mmult

function main (n, kmax : integer returns TwoDim )
  for initial
    k := 1
    A, B := Gen( n )
  while (k < kmax) repeat
    k := old k + 1;
    A := Mmult(n, old A, B)
  end for
end function % main
```

A.3.2 Pre-Transposed Matrix Multiply

```
% mmult1.sis
define main

type TwoDim = array [ array [ double_real ] ];

function Gen (n : integer.returns TwoDim, Twodim)

  for i in 1,n cross j in 1,n
  returns array of double_real(i)/double_real(j)
        array of double_real(i)*double_real(j)
  end for

end function % Gen

function Mmult (n : integer; A, BT : TwoDim returns TwoDim)
  % assumes that BT is already transposed.
  for i in 1,n cross j in 1,n
    c := for k in 1,n
           t := A[i,k] * BT[j,k]
         returns value of sum t
         end for
  returns array of c
  end for
end function % Mmult

function Transpose (n : integer; M : TwoDim returns TwoDim)
  for i in 1,n cross j in 1,n
  returns array of M[j,i]
  end for
end function % Transpose

function main (n, kmax : integer returns TwoDim)
  for initial
    k := 1;
    A, B := Gen(n);
    BT := Transpose(n, B);
  while (k < kmax) repeat
    k := old k + 1;
    A := Mmult(n, old A, BT);
  returns value of A
  end for
end function % main
```