CONF-9511199--1

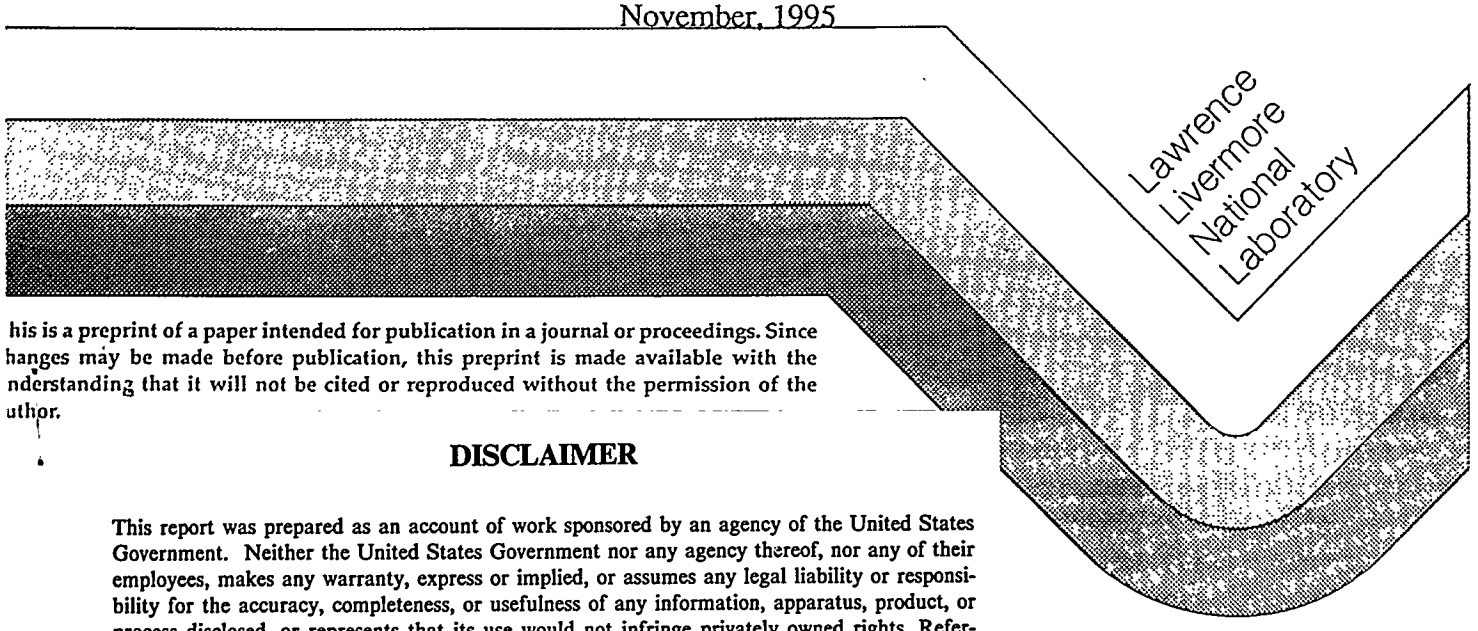
UCRL-JC-122696
PREPRINT

"Loop Parallelism on Tera MTA using SISAL"

Srdjan Mitrovic
Institute for Scientific Computing Research
Lawrence Livermore National Laboratory
Livermore, CA

This paper was prepared for submission to
International Conference on Parallel and
Distributed Processing Techniques and
Applications (PDPTA '95)
University of Georgia, Athens, Georgia
November 3-4, 1995

November, 1995



DISCLAIMER

Loop Parallelism on Tera MTA using SISAL

Srdjan Mitrovic
CRG/ISCR
Lawrence Livermore National Laboratory

Abstract

The difficulty of programming parallel computers has impeded their wide-spread use. The problems are caused by existing hardware and software tools. The software problems on shared-memory and vector computers can be solved by using deterministic high-performance functional languages like SISAL. Distributed-memory computers have even more obstacles than shared-memory parallel machines. Research indicates that multithreaded architectures can hide long latency of distributed memories and that they can solve the problems of locality. Tera's MTA multiprocessor is based on the concept of multithreading and provides the programmer with a real shared-memory model. This paper investigates the performance of parallel loops written in SISAL and executed on the Tera MTA using the Livermore Loops benchmarks.

1. Introduction

The programming of massively parallel systems encompasses problems that do not exist in single-processor machines. Race conditions, data and program partitioning, synchronization, and scalability are some of the main discouragements to a potential programmer of a massively parallel machine. Many of the problems, thought to be inherent to parallel computing can be avoided by using a deterministic language and latency-tolerant multithreading machines.

A project of the Computer Research Group at Lawrence Livermore National Laboratory is porting the compiler for the high performance functional language SISAL [1] to the Tera MTA [4]. This paper describes the initial efforts in this project.

The programming environment for the Tera MTA includes C, C++ and FORTRAN compilers as well as standard UNIX libraries. The hardware simulator is called Zebra. The results in this paper were obtained with the preeliminary F.2 release of the Tera C compiler and the simulator.

The Optimizing SISAL Compiler (OSC) generates C code and its runtime system is written in C. The porting of the runtime libraries to the Tera MTA was a simple and straightforward task. It is possible to compile and execute SISAL programs on the MTA by using the existing code generator of OSC. While the performance of that code is excellent on both vector and shared-memory multiprocessors, it is not satisfactory on the MTA. In the following sections we explain how the C

code generator has been adapted for the Tera system.

We use several Livermore Loop benchmarks to investigate the quality of the code generated for the Tera MTA using SISAL. The small size of these benchmarks is appropriate for the investigations presented.

2. SISAL on the Tera MTA

SISAL is a deterministic high-performance functional language that is suitable for programming various classes of parallel architectures [1]. An example of a SISAL function is depicted in Figure 1.

```
function Loop1( n: integer; Q, R, T: double; Y, Z: OneD returns OneD )
   for K in 1,n
       X := Q + (Y[K] * (R * Z[K+10] + T * Z[K+11]))
   returns array of X
   end for
end function
```

Figure 1: Example of Livermore Loop 1 written in SISAL

The SISAL compiler, OSC, generates concise dataflow graphs. Those dataflow graphs allow excellent sequential optimization, but their strength lies in parallel optimizations, e.g., partitioning and reordering of nodes. OSC 13.0 requires a shared-memory model, a C compiler and UNIX OS running on the target machine. All these requirements are met by Tera MTA.

Conventional multiprocessors have large-grained parallel units of execution. SISAL supports all levels of granularity. The advantage of SISAL when compared to conventional programming languages increases with the possibility of finer grained computation. On conventional multiprocessors SISAL currently exploits only loop parallelism. Other types of parallelism, e.g., pipelined and functional, will be exploited with Tera MTA.

OSC consists of several compilation phases that pass intermediate files to subsequent phases. The intermediate files contain dataflow graphs in either IF1 or IF2 representation [2] [3]. We modified the code generation phase of OSC, "if2gen". The new phase is called "if2gentera" and contains optimizations specific to the Tera architecture and to Tera C language.

3. Tera MTA

The Tera MTA [4] is a multithreading architecture with physically distributed memory that scales up to 256 processors. Every processor in the MTA can have up to 128 active threads and switches between threads after every instruction. Literature covering multithreading architectures can be found in [5] [6] [7]. The Tera MTA offers a shared-memory programming model as it is known in sequential machines, but is based on physically distributed memory. This means that we have a global address space where every processor can reach any memory location. Unlike similar machines, the MTA's multithreading capability makes program performance independent of data

distributed across separate memory modules.

Tera than for vector computers where the loops need not be vectorizable to perform well.

The MTA uses multithreading to hide memory latency. By switching between several threads of execution, long and irregular delays, caused by uncached and distant memory locations do not adversely affect the performance. An MTA instruction is moderately wide, comprising three operations: a memory operation, an arithmetic operand, and a control operation (MAC). By specifying an 'ADDMUL' in the arithmetic slot and an additional floating-point operation in the control slot, a single instruction can initiate three floating-point operations in one cycle. The maximum computation rate for a single processor is 1 GFlop with 64-bit real numbers.

We have run all benchmarks on the simulator called Zebra. While this is inconvenient in terms of execution speed, Zebra allows extensive and detailed observation of the performance and the code. It simulates the real hardware accurately. The simulator is validated by Tera's hardware groups on a daily basis [8].

4. Tera C

Contrary to most other parallel machine projects, Tera simultaneously develops hardware and software. Thus, it is not surprising that their C, C++ and FORTRAN compilers can be used before the actual hardware has been shipped and it is no surprise that their hardware and software fit together well.

The Tera C compiler is an ANSI-compliant C compiler with Tera-specific extensions. Some of the extensions are related to the description of parallel activities and some are related to the synchronization between these activities. Other extensions are pragmas that provide information to the compiler in cases in which it is not safe for compiler to assume something automatically.

Tera C and its run-time system offer more than a basic UNIX C, therefore some functions of the OSC run-time system can be deactivated, e.g., load distribution and controlling parallel processes.

The scheduling quanta in Tera MTA are threads. A thread can be generated in C explicitly with a future statement [8], or implicitly by writing a parallel loop. When Tera C recognizes a parallel loop, it generates code for lightweight threads called 'frays'. Threads that are defined with future statements have higher overhead and are more applicable for functional parallelism.

5. Tera C Code Generation with OSC

OSC produces C code that is compilable on many different machines. The C code is efficient on conventional machines where C compilers do not have the sophistication of Tera's parallelizing C compiler. The Tera C compiler offers various pragmas in cases where the compiler can't be sure about certain properties like inter-loop dependencies and aliasing. Appropriate insertion of pragma information affects the multiprocessor parallelism and allows the compiler to do software pipelining where possible. Software pipelining improves performance by reducing the loop overhead and improves the overlap of communication and computation by hoisting the loads further up.

The changes to the existing OSC code generator have been kept to a minimum. Most of the OSC optimizations for shared-memory and vector machines are being used for the Tera MTA. We

layout. The MTA is a UMA (uniform memory access) machine, whereas most shared-memory machines are NUMA (non-uniform memory access) machines. Porting and tuning a sequential program for the MTA can be accomplished by adding pragmas to assist automatic parallelization, without changing the data definition. There are no data caches, so cache coherence is not an issue. The sophisticated memory translation mechanism interleaves the content of structures across all memory modules. As Table 1 shows, when we run the first Livermore Loop on one processor using 2, 32, or 64 memory modules, the performance is nearly the same although data is being distributed across separate memory modules.

machine configuration	MFLops	difference
2-memory units	487	0 %
32-memory units	462	-5%
64-memory units	458	-6%

Table 1: Performance of Livermore Loop 1 running on one processor when varying machine configuration

In Table 2 we measure the performance of several Livermore Loops from a 1-processor to a 16-processor configuration. Since one processing element has 2 memory modules, a 16-processor configuration has 32 memory units. Changing the MTA configuration from 1-processor to 16-processor configuration forces aggregates to be spread across 32 memory modules instead of only 2. Note that the benchmarks in Table 2 always run only on a single processor. The difference between the two configuration is very small indicating that the performance is independent of data layout.

	2 memory modules	32 memory modules	difference
Loop 1	487 MFlops	464 MFlops	- 5 %
Loop 3	277 MFlops	268 MFlops	- 3 %
Loop 7	631 MFlops	629 MFlops	0 %
Loop 8	418 MFlops	407 MFlops	- 3%
Loop 9	510 MFlops	489 MFlops	- 4 %

Table 2: Difference in performance when increasing the number of memory modules from 2 to 32

The result in Table 1 and 2 indicate that programming or generating code for the MTA is much easier than for other distributed-memory or cache-coherent shared-memory machines where data decomposition and layout are major programming tasks. It is also easier to write efficient code for

must modify the internal dataflow graph representation before generating Tera C. Following a normalization of graphs, the code generator inserts new 'ADDMUL' nodes by collapsing the addition and multiplication nodes into a single node. Next we add new nodes for synchronizing 'CALL' nodes. Call nodes are being interpreted as a future statement and they need appropriate synchronization. On the last pass, the dataflow graphs are sorted. The sorting algorithm groups 'CALL' and parallel loop nodes together and increases the distance between the producing and consuming nodes. A similar approach has been described in [9].

After those modifications to the dataflow graphs, the C file is generated. The code looks very similar to the code generated for sequential computers but with inserted pragma information about aliasing and parallelism. By declaration we specify that every pointer is not aliased. This is a property of the C code generated by OSC. Every parallel loop is preceded with appropriate pragmas in order to force the C compiler to parallelize them. An example is given in Figure 2. Note that in some cases those pragmas are not necessary but OSC is inserting them.

```
{
  POINTER tmp14;
#pragma noalias tmp14
  POINTER tmp13;
#pragma noalias tmp13
.........
#pragma tera assert parallel
#pragma tera assert nodep
  for ( ; tmp15 <= tmp9; tmp15++ ) {  /* Normal Loop */
    GathATUpd( double, tmp14, (9.99859481776037e-03) );
    GathATUpd( double, tmp13, (9.99860056226325e-03) );
    }
  BldATDV( double, tmp14, (1), tmp10, tmp12 );
......
```

Figure 2: An example of C code generated by Sisal for the Tera MTA

Sisal guarantees determinacy and absence of race conditions. The use of described pragmas is therefore always safe and correct in the C code generated by OSC.

The OSC run-time system has been simplified. The load-distribution is now handled by MTA's operating system and the Tera C compiler. The memory management of OSC has been replaced by a simpler version used in combination with Tera's run-time memory allocation.

6. Loop Parallelism in Tera MTA

Loop parallelism is the simplest type of parallelism available in scientific programs. Most compilers, applications, and parallel architectures concentrate on that type of parallelism only. The loop parallelism in the Tera MTA has more possibilities than in conventional parallel architectures. Its lightweight processes, i.e., threads, allow much finer loop slicing than possible on conventional multiprocessors. Cheap synchronization using full/empty bits in shared memory allows overlapping of consumer and producer loops.

It is possible in Tera C to explicitly split a loop into slices by using the 'future' construct.

However, the overhead for futures is larger than the overhead of implicit loop parallelization that uses lightweight threads called 'frays'. Frays cannot be programmed explicitly in Tera C. The C compiler decides how many 'frays' to create for a given parallel loop. To make an optimal decision, the compiler must be able to understand the parallel properties of the loop.

Typically, nested parallel loops are rarely parallelized at all levels. For vector processors the innermost loops should be vectorized, while shared-memory multiprocessors prefer outermost loop parallelization. An outer loop iterating over small range, and the inner loop iterating over a large range is preferable for vector processors. For shared-memory processors, we want to unroll the outermost loop and remove the nesting. Software pipelining is possible only on the innermost loops. On the Tera MTA it is probably best to use the innermost loop to create work on one processor and the the outermost loop to spread work across several processors.

Loop parallelism can be exploited at three different levels within the Tera MTA. The coarsest level are loop slices that are executed on different processors. The medium level is slicing a loop into 'frays' that are executed on one processor. Those 'frays' are used to hide memory latency. The finest level is software pipelining which improves the code density and the hiding of latency by overlapping communication and computation. This is also called instruction-level parallelism (ILP). A loop that can be software pipelined and sliced into 'frays' will keep a processor busy, i.e., there will be no pipeline bubbles and every cycle executes an instruction. Software pipelining and the calculation of the number of loop slices is done by the Tera C compiler and cannot be influenced by OSC except by distributing or fusing loops. Figure 3 and 4 show how the body of the first Livermore loop is software pipelined.

```
for k in 1,n
            X[k] := Q + (Y[k] * (R * Z[k+10] + T * Z[k+11]) )
```

Figure 3: Computation of the first Livermore Loop

In Figure 4 every line represents a single Tera instruction. Every instruction contains up to three operations. The first one is a memory operation, the second one can contain 'ADDMUL' operation and the third one can contain control or a floating-point operation.

```
0006:: .....
0007::( ()                  (r15 = r4  * r21)       (r6  = r6  + 8) )
0008::( (r14 = LOAD(r6))    (r12 = r15 + r5  * r14) (r7  = r7  + 8) )
0009::( (r13 = LOAD(r7))    (r12 = r17 + r12 * r23) (r24 = r24 + 8) )
0010::( (STORE (r12 r24))   (r23 = r4  * r19)       (r6  = r6  + 8) )
0011::( (r21 = LOAD(r6))    (r12 = r23 + r5  * r21) (r7  = r7  + 8) )
0012::( (r15 = LOAD(r7))    (r12 = r17 + r12 * r3)  (r24 = r24 + 8) )
0013::( (STORE (r12 r24))   ()                      ()              )
0014:: ....
```

Figure 4: First Livermore Loop in the MAC instruction format

There are three reads from memory and one write to memory in the loop body. Two read memory operations for array 'Z' can be collapsed into one read via software pipelining. The

remaining three memory operations require three operations for incrementing the X, Y and Z indices. The computation of X[k] requires five floating-point operations which can be compacted into two 'ADDMUL' and one multiplication operation. The three memory, three integer, and three floating-point operations can be packed into three MAC instructions. Therefore, the maximum computing performance of this loop is 5/9 of a Gigaflop. Figure 4 shows how the software pipelined code for the body of the Loop 1 would look like. The results of the 'LOAD' instructions in Figure 4 are used much later, thus allowing for better latency hiding.

7. The Benchmarks

The Livermore Loops are one of the oldest benchmarks available for testing the performance of computing machines on scientific programs. They have been used since the mid-60's and represent the computation kernels typically found in large scale scientific computations [10] [11]. Their disadvantage is their short code and small problem size. Because of their simplicity and because they offer a variety of loop parallelism, the Livermore Loops are suitable to analyze code generation issues in detail. In our study we use only parallel loops.

In Table 3 we list the simulated performance of Sisal versions of the Livermore Loops on one processor. The third column shows the percentage of the measured performance to the theoretical maximum for the MTA architecture. The theoretical maximum is calculated when the loop body is completely unrolled over infinite range, the instructions optimally compacted and when one instruction can be executed every cycle. The maximum computational rate of a single processor Tera MTA is one Gflop.

Livermore Loop	Performance (MFlops)	% of theor. maximum
1	487	87%
3	277	84%
7	631	94%
8	418	86%
9	510	98%
10	131	82%
12	110	64%
18	293	~73%
21	512	91%

Table 3: Performance of parallel Livermore Loops on one-processor Tera MTA. The theoretical maximum for Loop 18 can be only estimated.

Table 3 includes various problem sizes, typically larger than required in the original report

[11]. The results show that the code generated by OSC and Tera C from 'vanilla' Sisal source code is reaching in most over 80 %.

Loop 12 is simple computation of the first difference. It generates inferior performance because the preeliminary F.2 Tera-C compiler compacts the loop body in three instead of two instructions, thus lowering the theoretical maximum to 111 MFlops. This has been fixed in the F.4 release. Loop 8 is a large loop which requires loop distribution to be able to execute optimally. The theoretical maximum for this loop is computed given the used loop distribution which is lower than for a single loop. Loop 18 contains outer sequential loops and innermost parallel loops. The theoretical maximum can be only estimated. Loop 18 contains sequential parts which lower the computational rate.

OSC has powerful loop-fusion optimization. Fusing several loops together into a single large loop has the benefit of reducing jumps and memory operations, and enhances common subexpression elimination. However, it also produces larger loops that cannot be software pipelined because of register requirements. Therefore, loop fusion is switched off when generating code for Tera or for the vector computers.

8. Future Work

We have noticed that overly complex loops can hinder analysis and parallelization unnecessarily, whereas small loops increase the overhead. The loop fusion optimization in OSC is necessary for optimizing code, but must be augmented by a loop distributor that will split large loops into several small ones. Some loop fusion and distribution is done already by Tera C compiler and some still needs yo be done in OSC.

Another optimization that will be included in OSC is the exploitation of non-strict parallelism using the cheap synchronization with full/empty bits. Assuming that a first loop produces values that are read in a following loop, we can start both loops in parallel. They will synchronize using full/empty bit associated with every value that is written in the first and read in the second loop. The threads of the consuming loop will block if they try to consume a value that has not been yet computed by the threads of the first loop. In this way we can remove performance gaps caused by synchronization barriers.

9. Conclusion

We have demonstrated some of the promising features of a new type of massively parallel architectures that will change the way we do parallel computing. The architecture offers a very simple programming model. We have learned that new optimizations are needed with OSC in order to exploit the ILP on Tera.

After becoming a powerful language for shared-memory and vector computers, SISAL is entering successfully the domain of multithreading architectures. The data dependency information provided in OSC produces excellent scalar code. Tera MTA can be programmed in FORTRAN or C, but SISAL has the ability to extract more parallelism from algorithms. This comes from the fact that internal code representation is based on concise acyclic dataflow graphs, which allow easy

dependency analysis and partitioning. This ability should offer additional threads to the Tera MTA machine. We have only scraped the surface of the potential of the MTA. Functional and pipelined parallelism, briefly mentioned above, are easily exploited and will make the Tera MTA and SISAL stand out even more.

10. Acknowledgment

The author wants to thank the members of CRG at the Lawrence Livermore National Laboratories for their help and suggestions, and S. Fitzgerald for fruitful discussions. The author is grateful for the support by Tera Computer Systems, especially from P. Briggs, and for being able to use their excellent tools.

This work was performed under the auspices of the US Department of Enery at LLNL under Contract No. W-7405-Eng.48.

References

1. J.R. McGraw, S.K. Skedzielewski, S.J. Allan, R.R. Oldehoeft, J. Glauert, C. Kirkham, W. Noyce and R. Thomas. SISAL: Streams and Iterations in a Single Assignment Language: Reference Manual 1.2, Lawrence Livermore National Laboratory, March 1985.
2. S. Skedzielewski and J. Glauert. An Intermediate Form for Applicative Languages, Lawrence Livermore Nationa Laboratory, 1984.
3. M. Welcome, S. Skedzielewski, R.K. Yates, J. Raneletti. IF2 - An Applicative Language Intermediate Form with Explicit Memory Management. Lawrence Livermore Nationa Laboratory, 1986.
4. R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield and B. Smith. The Tera computer system, International Conference on Supercomputing, 1990.
5. R. Buehrer and K. Ekhanadam. Incorporating Dataflow ideas into von Neumann Processors for Parallel Execution, IEEE Transactions on Computers, VOl. C-36, No. 12, December 1987.
6. O. Maquelin. The ADAM architecture and its Simulation, PhD Thesis, ETH Zurich, Switzerland, 1994, ISBN 3 7281 2113 4.
7. G.T. Byrd and M.A. Holliday. Multithreaded Processor Architectures, IEEE Specirum, August 1995.
8. Tera Programming Guide, Tera Computer Company, March 1995.
9. S. Mitrovic. Compiling SISAL for the ADAM architecture, PhD Thesis, ETH Zurich, Switzerland, 1993.
10. J.T. Feo. An analysis of the Computaional and Parallel Complexity of the Livermore Loops, Parallel Computing 7 (163-185), North Holland, 1988.
11. F.H. McMahon. Livermore FORTRAN Kernels: A Computer Test of the Numerical Performance Range, Lawrence Livermore National Laboratory, 1988.