An Approach for Direct Dataflow Execution on Contemporary Multicore Systems

Dumitrel Loghin, Bogdan Marius Tudor and Yong Meng Teo
Department of Computer Science
National University of Singapore
{dumitrel,bogdan,teoym} at comp.nus.edu.sg

***Abstract*—Traditionally, imperative programming uses a series of state-based operands to model control-flow and, as a result, suffers from the well-known von Neumann bottleneck. In contrast, dataflow programs are driven only by the availability of instruction operands. However, the lack of mainstream dataflow hardware hinders direct dataflow instruction execution. On the other hand, direct execution of dataflow programs on von Neumann machines incurs a high performance cost. In this paper, we present preliminary results on the direct execution of dataflow programs on multicore systems through emulation of tagged-tokens mechanism. Compared with direct translation of SISAL programs to C code, we achieved a speedup of 44 for CPU-intensive applications and 22 for memory-bounded applications on a 48-cores AMD NUMA system.**

I. Introduction

Chip processors with tens of cores have established themselves as the backbone of computing. With each technological generation, multicore systems have an increasing number of cores integrated on a die and they are being used across different application domains. However, as the number of cores grows, it is increasingly challenging to scale the software to this level of parallelism.

Unfortunately, traditional methods for writing parallel programs, such as using C, C++ or Fortran supported by pthreads or OpenMP require the programmer to focus on the control flow of the program, rather than to understand the parallelism structure. Furthermore, it is often a tedious and complex task [14], [23]–[25] to map high-level program parallelism to low-level hardware parallelism. Thus, with each multicore generation, there is a growing gap between coarse-grain hardware parallelism and the parallelism exposed by traditional software development methods.

A *dataflow program* uses a model of computation distinct from the control-flow, or von Neumann model of computation [1], [10], [22]. In the dataflow model, a program is represented as a directed graph. Nodes in the graph represent program operations and can be simple such as arithmetic operations, or complex such as sorting. The edges represent the flow of data from a producer node to consumer nodes. A node becomes ready to execute when there is data on all its input edges. This principle of dataflow execution relieves the programmer from explicitly identifying, exposing and controlling the parallelism in a program. A *tagged-token dataflow architecture* exposes the principles of dynamic dataflow graph execution [1], [10]. Tagged-tokens method allows multiple instances of the same subgraph, where subgraph can be a function or the body of a loop, to be executed in parallel. Tokens belonging to different contexts are colored with different tags. During program execution, when a function call or a new iteration of a loop is spawned, a new tag is generated.

This paper investigates coarse-grain dataflow execution on modern multicore systems. Given a fine-grain dataflow graph, obtained from a SISAL [16] program, we perform node fusion and optimization to derive an executable dataflow graph. The C++ representation of this graph, linked together with our proposed *Runtime Dataflow Engine* (*RDE*) produces an executable for multicore systems. *RDE* exploits multithreaded architecture by distributing dataflow tasks across multiple processor cores, using runtime generation and matching of tagged-tokens. Our preliminary results show that execution on *RDE* performs better than direct translation of SISAL program into control-flow program.

The contributions of our papers are twofold. First, we propose a multithreaded emulator of dataflow execution on multicore systems. Second, we present a cost analysis for mapping dataflow parallelism to modern multicore hardware. The novelty of the analysis stems from linking the overhead of managing tokens to the cost of the synchronization operations, highlighting the non-linear impact of the number of cores and memory latency on performance. The cost analysis is derived from measurements on a 48 cores NUMA system. We show that task granularity is key to achieving good performance and discuss the conditions for a dataflow program to scale on multicore.

The rest of the paper is structured as follows. Section II presents the related work. Section III shows our proposed approach and the organization of our dataflow execution system. In section IV we evaluate the performance of the proposed system against direct translation of the same code. Finally, in section V we conclude.

II. Related Work

The principles of dataflow execution, in particular fine-grain dataflow, have been widely studied and applied in a variety of computing systems. During the last three decades, dataflow research includes the design of dataflow machines [1], [10], and languages [5], [6]. More recent effort includes the translation of dataflow programs to imperative programs for execution on hybrid CPU-GPGPU systems [2], [7], [11], [13].

The shift to multicore has brought large parallel systems to mainstream. The availability of multicore systems naturally raises the question of the extent with which dataflow programs

can benefit from this parallelism. However, currently direct dataflow execution on multicore systems is not well understood and the cost of translating dataflow code to von Neumann code remains unclear. Our approach is based on the principles of dynamic dataflow described in Arvind and Nikhil [1] and implemented in the Manchester Dataflow Machine [10]. The Manchester Dataflow Machine was the first implementation of fine-grain dataflow execution. Each ring, in the multi-ring machine, consists of four pipeline units, namely *token queue*, *matching unit*, *fetching unit* and *functional unit*. A program is executed by placing initial data tokens in the token queue. The matching unit determines if a node has all input tokens. When all tokens are available, the node fetches its instruction from the fetch unit and is ready to be executed by the functional unit. Otherwise, the node waits for its missing operands in the matching unit. Software simulation of dataflow architecture includes the work by Barahona and Gurd [9] in 1985. Generally, simulators closely models fine-grained dataflow execution and details of the dataflow machine. In contrast, we adopt a higher level of abstraction by modeling only tokens generation and consumption by independent dataflow instructions for coarse-grain execution. While the Manchester Dataflow Machine keeps a token queue and uses a hash mechanism for token matching, we maintain a map of active contexts for every node to support immediate node-context matching.

A number of studies exploit dataflow execution on non-dataflow machines. In a compiler-based approach, a dataflow program is translated into von Neumann code such as in *fsc* [5], *sisalc* [17] for SISAL and *sac2c* [6] for Single Assignment C. In this approach, the generated code is either sequential or parallel. For example, in a program with two nested for loops over the interval 1 to n, *fsc* generates n^2 lightweight threads called filaments. For superscalar processors, dataflow analysis has been used to improve instruction issue rate [3]. In distributed-memory code generation, Serot uses the tagged-token dataflow model for implementing parallel skeletons and discusses the importance of bounding the iterations and recursions resulting from parallel nesting [18]. In contrast, our approach generates an Executable Dataflow Graph for direct execution using our Runtime Dataflow Engine on shared-memory multithreaded architecture.

Hybrid dataflow-von Neumann model has also been explored. Based on dataflow granularity curve, Sterling et al. observed that medium-grain dataflow achieves the best performance [21]. However, our analysis shows that the granularity curve shifts with the number of cores and the latency of memory operations. Thus, finding the optimal granularity is more complicated in current multicore systems. More recently, Evripidou et al. proposed a new programming model called Data-Driven Multithreading (DDM) [15], based on dataflow model of execution. Subsequent works include tools for exploiting DDM on commodity multicores [19], [20], and the usage of DDM in HPC [4]. Targeting function level parallelism, Gupta and Sohi applied dataflow analysis to generate multithreaded code from imperative programs [8]. In contrast with the previous approaches, we start with a dataflow program and exploit different levels of parallelism including function calls, loop iteration and fused instructions. Our evaluation shows that different levels of granularity are important in achieving good performance on contemporary multicores.

III. Approach

A. Overview

The primary objective of our approach is to efficiently execute a dataflow program on current multicore systems through emulation of dataflow execution. Given a dataflow program, we perform a set of transformations to obtain an executable dataflow graph with different levels of granularity. This graph is directly executed on an emulated dataflow engine running on a multicore system as shown in Fig. 1. In order to

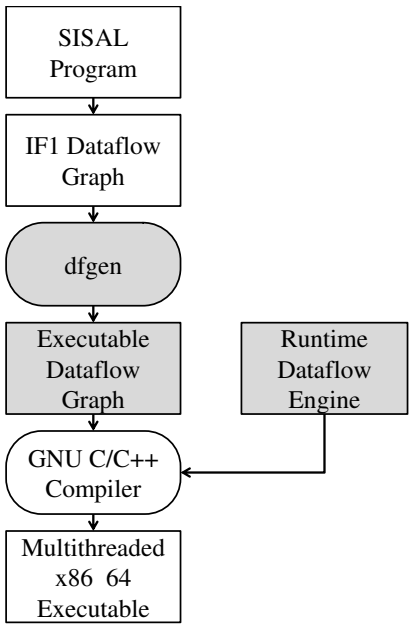


Fig. 1: Proposed dataflow approach

compare the performance of our approach with *sisalc* [17], we use SISAL as the starting point. A dataflow program in SISAL is first compiled into an *IF1* graph. Next, *dfgen* takes the IF1 graph, and fuses nodes to obtain an Executable Dataflow Graph (EDFG) with different task granularities. A *Runtime Dataflow Engine* (*RDE*) executes EDFG using a *work-pool* model. Workers execute *dataflow tasks* in parallel on different cores. The generated EDFG in C++ code is linked with *RDE* to produce a x86 executable for multicore systems.

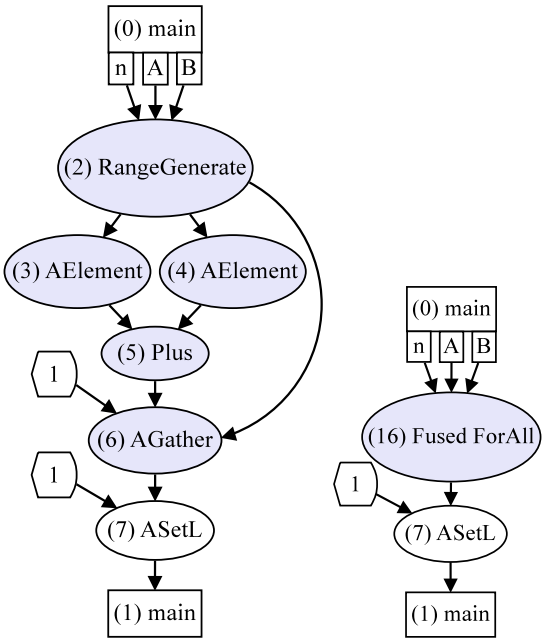
B. Executable Dataflow Graph

EDFG is a more efficient graph representation obtained from IF1 by performing two main transformations, namely, node fusion and node optimizations. EDFG consists of three type of nodes. *Simple nodes*, as in IF1, represent operations such as addition, multiplication and comparison. *Fused nodes* are composed from a sequence of simple nodes representing functions and loop iterations with different levels of granularity. *Compound nodes*, as in IF1, represent wrappers for complex instructions such as branches and loops, and nodes for function entry and exit points required in EDFG.

Node fusion is performed to obtain dataflow tasks with different levels of granularity. Subgraphs, such as functions and loop iterations, are collapsed into fused nodes to reduce token management overhead. At runtime, a fused node is executed as a dataflow task. Fig. 2a shows a SISAL program that adds

```
define main
type IntArray = array[integer];
function main(n: integer; A, B: IntArray
  returns IntArray)
    for i in 1, n
        returns array of A[i] + B[i]
    end for
end function
```

(a) SISAL program



(b) IF1 graph (c) EDFG

Fig. 2: Node fusion in loops

two arrays of size n using a for loop. By applying node fusion to loop iterations, we reduce the number of nodes from five (Fig. 2b) to one (Fig. 2c). Our evaluation shows that node fusion reduces the number of tokens by several orders of magnitude.

In node optimization, node trimming is performed to reduce redundant nodes. In IF1, branching is achieved using a *Select node*, with three subgraphs: *condition*, *then branch* and *else branch*. By default, IF1 groups these three subgraphs into one compound node. Thus, in a naive implementation, the three subgraphs are executed in parallel but only one result is selected at the end. We optimize by putting three Select start-nodes for the three subgraphs and two Select end-nodes, one for then branch and one for else branch. In the optimized form, the conditional subgraph is moved up such that it executes before entering the then and else subgraphs. We do this by adding two special nodes: Select Then and Select Else. By doing this optimization, we avoid executing unnecessary operations. Similarly, a *ForAll node* used in looping, contains three subgraphs: *generator*, *body* and *results*. Usually, a ForAll generator has a RangeGenerate node or a AScatter node. The RangeGenerate node distributes indexes and AScatter distributes array elements to loop body. We call these nodes Generator. The results subgraph usually contains a Reduce or an AGather node. The Reduce node reduces the body results into a single value. AGather node produces an array from body results. We call these nodes Result. In our approach, a Generator spawns new contexts and a Result restores the original contexts. In the first iteration, four special nodes are generated: one each at ForAll entry and exit, one at the beginning of the body subgraph and one at the beginning of the results. These four nodes can be removed by making Generator node the entry node in ForAll and by adding a new edge from Generator to Result. Through this link, Result node will be notified when the loop begins and when it reaches the end. This optimization reduces the total number of operations by 10-20% on the workloads that we evaluated.

C. Runtime Dataflow Engine

RDE is designed to achieve better dataflow execution performance using a high-level abstraction of dataflow machine. Fig. 3 shows the runtime execution of EDFG on *RDE* with four main execution steps:

1) initial data tokens arrive at the static EDFG to pick-up the corresponding nodes (dataflow tasks) for execution
2) a color is generated by a tag generator for each static node and are forwarded to the dynamic dataflow graph store
3) a node with all input tokens available (dataflow task) is ready for execution and is sent to the task pool, otherwise, it waits at the dynamic dataflow graph store
4) workers (processor cores) pick-up dataflow tasks from the task pool for execution and result tokens are then returned to the static graph.

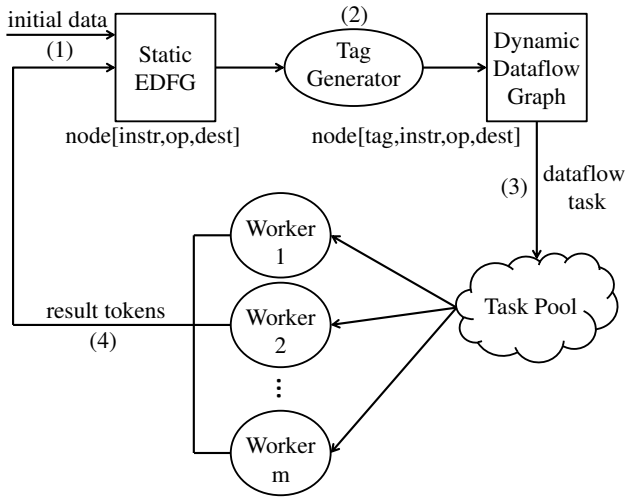


Fig. 3: Runtime Dataflow Engine

To avoid heavy usage of synchronization, *RDE* employs a lazy evaluation mechanism. Instead of pushing result tokens, *RDE* stores them in producer nodes and just notifies the consumer nodes that an operand is available, without copying the operand information. Thus, tokens are fetched only when consumer

nodes are executed, minimizing the need for inter-thread synchronization. Furthermore, the notifications mechanism uses atomic integer operations, thus avoiding the usage of costly pthreads semaphores or locks.

IV. Evaluation

In this section, we present an evaluation of our approach in two parts. First, we compare our approach for direct dataflow execution (*DFE*) against direct SISAL to C translation done by *sisalc* [17] compiler that generates sequential von Neumann code. Secondly, we analyze the cost of dataflow execution for different levels of task granularity.

We present preliminary performance results of our approach using two programs with widely different characteristics. Matrix multiplication (MM), a commonly used benchmark, multiplies two square matrices of size n using three for loops. This program has ample parallelism with regular task sizes, and is memory intensive. The second program, prime number counting (PR), computes the number of primes in the interval 1 to n. In contrast with MM, it has irregular task sizes and generates little memory traffic. Both programs allow us to test the effectiveness of dataflow node fusion on functions and loops. *RDE* and *dfgen* are implemented in C++. In *RDE*, each worker thread is bound to a core and in our experiments, the number of threads is always less or equal with the number of cores. The experiments were performed on a 48-core AMD Opteron 6172 NUMA system. The NUMA system has 64 GB RAM accessed by cores through eight memory controllers with three memory latencies: *0-hops* (local controller), *1-hop* (traversing one NUMA links) and *2-hops* (traversing two NUMA links). We have used the latest 64-bit Linux kernel, version 3.8.0 and GCC version 4.6.3. All C++ programs were compiled with optimization (`-O2` flag). Each experiment was conducted three times and the smallest execution time is reported.

A. Comparison with sisalc

Table I compares our direct dataflow execution (*DFE*) time with *sisalc* on three input sizes for each program. When

Program	n	Time [sec]		Speedup
		sisalc	*DFE*	
MM	1000	28.4	2.1	13.5
	2000	334.8	15.0	22.3
	4000	–*	176.0	–
PP	100k	51.8	2.5	20.7
	200k	211.8	6.0	35.3
	500k	1359.5	30.4	44.7

TABLE I: Comparison with sisalc

comparing to *sisalc* programs, our *DFE* has a speedup of 22 for MM(n=2000) and 44 for PR(n=500k). For MM(n=4000) *sisalc* does not run due to a memory allocation failure. The different speedup of MM and PR can be explained by the fact that MM intensively uses the memory, thus, many processor cycles are stalled waiting on memory requests. On the other hand, PR is less memory intensive and better uses the computational resources of a multicore system. For both programs, the best performance is achieved using medium granularity, as explained below.

*execution aborted due to memory allocation failure

As expected, the performance of our approach is strongly influenced by the task granularity. We describe the different task granularities used in our analysis. The average granularity of a dataflow task is defined as:

$$\Phi = \frac{DI}{FDI} \tag{1}$$

where DI is the total number of fine-grain (IF1) dataflow nodes, and FDI is the total number of dataflow tasks after applying node fusion. In the MM program with three for-loops:

$$DI_{MM} = 6n^3 + 2n^2 + 3n + 6 \tag{2}$$

The PR program consists of two for-loops, with the outer loop iterating through the numbers in the interval 5 to n, and the inner loop searching dividers for each number. Thus:

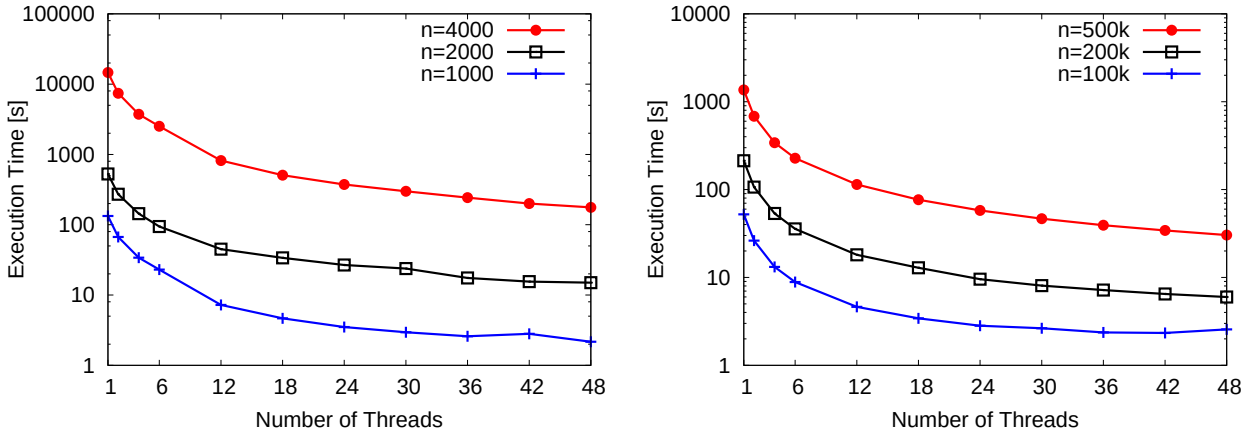
$$DI_{PR} = \frac{5}{2}n^2 - 3n - 25 \tag{3}$$

In this paper, we evaluated five types of node fusion ranging from fine-grain to coarse-grain:

- Φ_1 – no fusion;
- Φ_2 – inner-most loop iteration is fused, and function calls within for loops are fused;
- Φ_3 – everything is fused, except the outer-most loop;
- Φ_4 – node fusion is applied until the number of tasks in the outer-most loop is equal to the number of cores;
- Φ_5 – all the nodes are fused into one dataflow task.

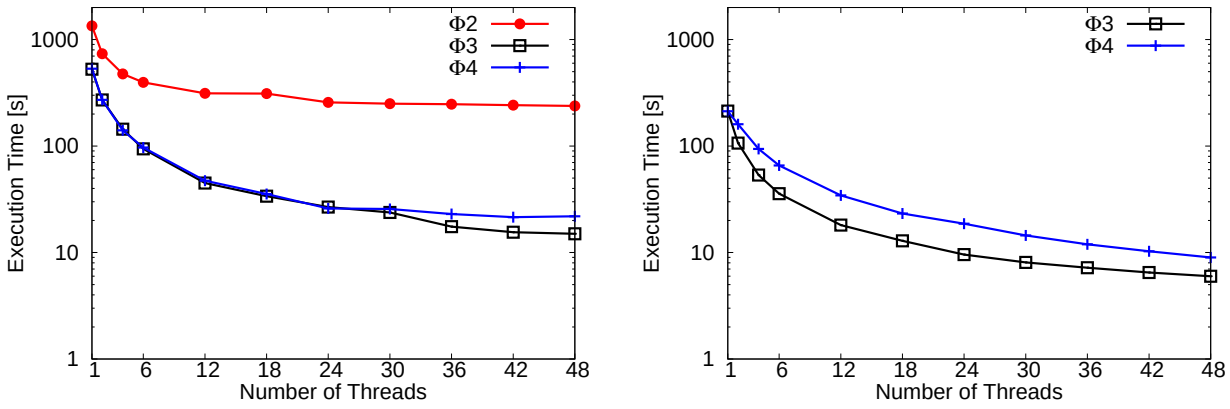
We present the execution time of our two applications on three selected input sizes for the best performing granularity Φ_3. Because the execution time on low core counts it is much larger than on large core counts, Fig. 4a and 4b are plotted using a log scale to better show the difference between the performance achieved by different core counts. Both figures show that our *DFE* implementation scales well, achieving a maximum parallel speedup of 35 for MM(n=2000) and 44 for PR(n=500k) on 48 cores.

The performance of our *DFE* is strongly linked to the granularity of the dataflow instructions. Thus, we focus on the medium task granularities Φ_2, Φ_3 and Φ_4. Their execution times are shown in Fig. 5a for MM and in Fig. 5b for PR. Φ_2 for PR does not run with large input size due to high usage of memory, thus is not plotted on Fig. 5b. For MM(n=2000), granularity Φ_2 executes on average eight times slower than Φ_3 on all core counts. For both MM and PR, Φ_3 performs better than Φ_4 due to a larger work-pool of dataflow tasks. On CPU intensive applications, such as PR(n=200k), Φ_3 performs 66% better in average. On memory bounded applications, such as MM(n=2000), the difference is smaller: 12% in average. For both task granularities we observe that executions on low core counts results in smaller performance difference between Φ_3 and Φ_4. However, for core counts larger than 24, Φ_3 consistently out-performs Φ_4. As expected, the finest grain, Φ_1, is much slower, incurring a prohibitively large execution



(a) MM (b) PR

Fig. 4: Effect of problem size on execution time, using Φ_3 granularity



(a) MM(n=2000) (b) PR(n=200k)

Fig. 5: Effect of task granularity on execution time

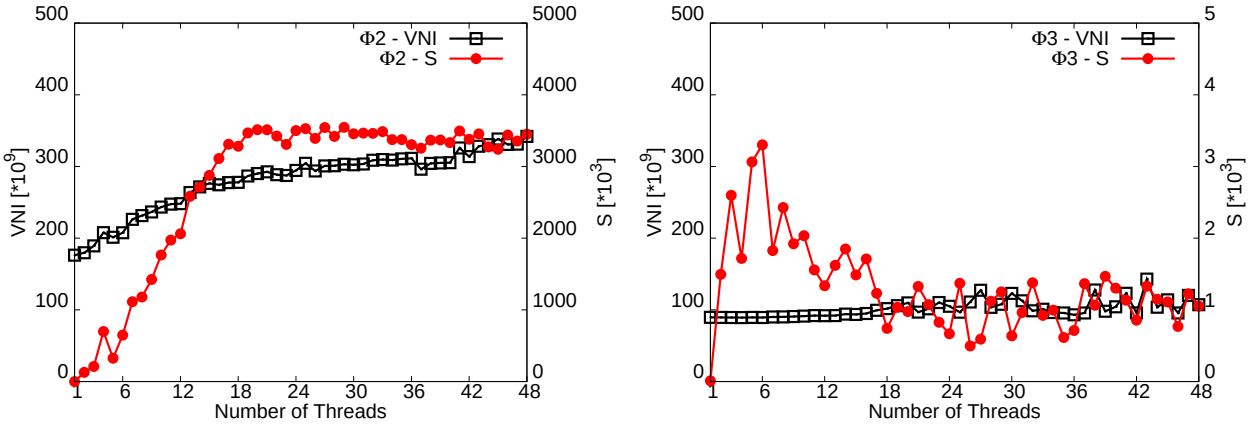
time (more than one hour when executing MM(n=1000) and PR(n=100k) on 48 cores) and incurs a very large memory usage. Φ_5 achieves a similar performance to Φ_3 and Φ_4 on one core, for both MM and PR.

B. Cost of Different Task Granularities

Motivated by the impact of dataflow granularity on performance, we analyze the cost of dataflow execution. In particular, we investigate the cause for the drop in performance when dataflow programs with small task granularity are executed on modern multicore systems. To focus the analysis, in this section we discuss the performance of MM(n=2000) using Φ_2 and Φ_3. As shown in the previous section, Φ_3 has the best performance among the granularities, but Φ_2 underperforms Φ_3 by a factor of eight. We profiled the execution of MM(n=2000) measuring the following parameters:

1) VNI – Number of machine instructions (i.e. von Neumann instructions) executed by the programs across all threads;
2) S – Number of synchronization operations performed by the OS kernel (i.e. atomic increment and decrement, mutex and semaphore lock and unlock operations);
3) T_{sync} – Average time required by the kernel to complete all synchronization operations
4) A – Number of active threads, averaged across the execution time of the program. A thread is considered active if it is not stopped at a synchronization operation;
5) T – Total execution time of the programs.

The number of von Neumann instructions VNI, average thread utilization A and total execution time T are measured using the processor hardware events counters which are sampled using the `perf` tool. The dataflow tasks FDI and the fine-grain dataflow instructions DI are measured using our own instrumentation code inserted in the *RDE*. The number of synchronization operations S and average blocking time for synchronization operations, T_{sync}, are measured using the `strace` system call.



(a) Φ_2 (b) Φ_3

Fig. 6: MM(n=2000) – Von Neumann instructions (VNI) and total synchronization operations (S) for two granularities

Our analysis shows that the execution of dataflow programs with small task granularity on contemporary multicores suffers from three performance problems:

1) The number of von Neumann instructions, VNI, is larger than the same program executed with coarse-grain dataflow tasks.
2) Due to the larger number of dataflow tasks, the total time spent by the program in synchronization operations is substantial. Furthermore, this synchronization time increases with the number of cores.
3) The number of active threads of the dataflow multithreaded execution is low, even if the dataflow program has enough parallelism and the system has enough cores.

Further, we detail the three effects. Figures 6a and 6b show the total number of von Neumann instructions, VNI, and the total number of synchronization operations, S, for Φ_2 and Φ_3 granularities of MM(n=2000). The figures show that VNI on Φ_2 is more than three times larger than on Φ_3. To understand the reason for the increase in VNI, we have profiled the entire run of the programs using *Zoom* statistical profiler. The profiler shows the breakdown of VNI in each RDE function, for both userspace and kernelspace. The profiler indicates that for Φ_2, over 70% of VNI are incurred by the OS kernel when executing system calls, and less than 30% are directly attributed to the userspace *RDE* code. Moreover, the system call responsible for 99% of the time spent in kernel mode is *futex*, which is the system call used to perform synchronization among threads. In contrast, for Φ_3, more than 99% of VNI are incurred in userspace. This analysis indicates that the breakdown in performance for Φ_2 is attributed to synchronization operations among the threads of the RDE.

To validate our hypothesis, we have profiled the total number of synchronization operations performed by the OS kernel. Figure 6a shows that for small dataflow granularity Φ_2 there is a strong correlation between the increase in number of synchronization operations, S and the increase in VNI (Pearson product-moment correlation coefficient is 0.94). In contrast, for larger dataflow granularity Φ_3, the correlation between the increase in VNI and increase in S is very weak (Pearson product-moment correlation coefficient is -0.43). This validates our hypothesis, showing that synchronization has a big impact on Φ_2 execution. However, for Φ_3 our analysis shows that *VNI* and *S* are not correlated. Moreover, *S* is about 3000 time smaller than for Φ_2, thus the effect of synchronizations is negligible.

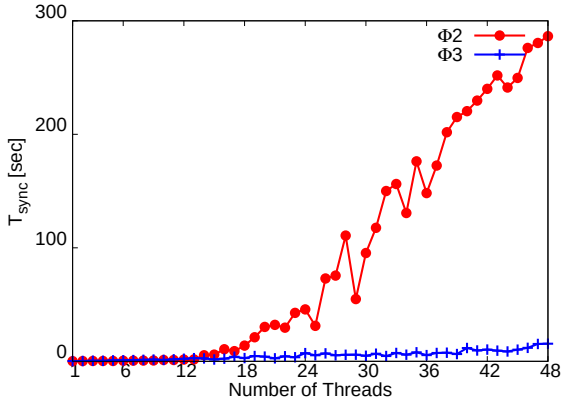


Fig. 7: MM(n=2000) – T_{sync} vs. threads

Figure 6a also shows that the number of synchronization operations increases with the number of threads. This is because in Linux, locks and semaphores are implemented on top of a construct called *fast userspace mutex* (hence the name *futex* for the system call) [12]. If only one thread attempts to lock an uncontended futex, the thread will proceed without performing a system call. Thus, for the uncontended case, the lock will resolve quickly. However, if multiple threads contend for a lock, the futex will perform a system call. The kernel will then award the futex to one of the threads, and suspend the other threads. Therefore, for the contended case, the locking operation will incur a higher cost, because the OS kernel must be involved. In our programs, the total number of lock/unlock operations is constant. However, for small number of threads the likelihood that a lock is contended by several threads is small: $20-40\%$ of all locks are contended when $m < 6$. But

for $m > 12$, more than 70% of all lock operations are observed to require a system call. Thus, even if the total number of locks operations is constant in the programs, executions on larger number of threads will result in more calls to the OS futex routines, which in turn increases the number of von Neumann instructions incurred by the programs. To further show the effect of synchronization on the execution time, we plot the total time spent in synchronization operations versus the number of threads. Fig. 7 shows the exponential increase of T_{sync}, thus canceling the gain of using more cores for useful work.

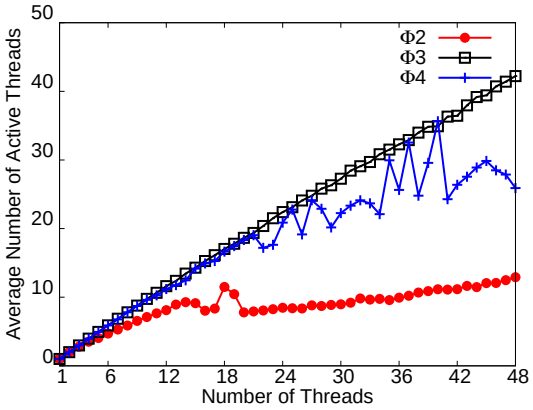


Fig. 8: MM(n=2000) – A vs. threads

Next we discuss the evolution of the number of active threads with the increase of core counts. Ideally, in a perfectly scaling program, the average number of active threads equals the core count. However, in a realistic program, data dependency among threads leads to synchronization operations that decreases the average number of active threads [23]. Fig. 8 shows the evolution of A for all three medium granularities Φ_2, Φ_3 and Φ_4 on MM(n=2000). When multiple threads are active, the program will suffer a parallelism loss due to data-dependency among threads. Even if it scales, Φ_3 suffers form the above effect, having A=44 when using 48 cores. For Φ_2, the heavy usage of synchronization has a dramatic effect on A, reducing it to less then 10 for big core counts. Φ_4 suffers from the unavailability of work. Thus, even if the number of dataflow tasks equals the number of used cores, data dependency between tasks decreases the number of ready-to-execute tasks.

The conclusion of the analysis is that the cost of token management is by far the most significant overhead in tagged-token dataflow execution on contemporary multicore systems. Selecting the right granularity is the main factor for managing this cost. On small number of cores the performance difference between medium and larger granularity is small, but on large number of cores, programs with medium dataflow granularities consistently achieve a better performance.

V. Conclusion

This paper proposes a new approach for exploiting dataflow parallelism on multicore systems. Our approach consists of two main steps. First, a SISAL program is translated into an Executable Dataflow Graph (EDFG). We applied node fusion on both loops and functions to increase the grain size and node optimization to remove redundant nodes. Our approach reduces the overhead of generating and managing tokens. In the second step, the EDFG is executed on a light-weight Runtime Dataflow Engine (*RDE*) which emulates tagged-tokens dataflow execution. *RDE* exploits multicore parallelism using a work-pool model that distributes coarser-grain dataflow tasks to cores. Our proposed execution approach surpasses *sisalc* by 44 times for prime number counting program and 22 times for memory-bounded matrix multiplication. However, key to achieving good performance is the granularity of dataflow tasks. Our evaluation shows that task granularity has a non-uniform relationship to performance. On a 48-cores AMD NUMA system, medium-size task granularities achieve best performance.

Our preliminary results motivate the extension of this work in two directions. First, we are developing runtime techniques to generate *malleable tasks* of different granularities that dynamically adapt to changes in the parallelism of the system. Secondly, a hierarchical dataflow execution engine can be implemented by breaking the EDFG into subgraphs that execute on different memory domains. This can improve the *DFE* performance on NUMA shared-memory systems, and more importantly, allow execution on distributed-memory systems and hybrid shared/distributed-memory systems.

Acknowledgments

This work is supported by the Singapore National Research Foundation.

References

[1] K. Arvind, R. S. Nikhil, Executing a Program on the MIT Tagged-token Dataflow Architecture, *IEEE Transactions on Computers*, 39(3):300–318, 1990.

[2] A. Balevic, B. Kienhuis, An Efficient Stream Buffer Mechanism for Dataflow Execution on Heterogeneous Platforms with GPUs, *Proc. of 1st Workshop on Data-Flow Execution Models for Extreme Scale Computing*, pages 53–57, 2011.

[3] A. Bracy, P. Prahlad, A. Roth, Dataflow Mini-graphs: Amplifying Superscalar Capacity and Bandwidth, *Proc. of 37th Annual International Symposium on Microarchitecture*, pages 18–29, 2004.

[4] C. Christofi, G. Michael, P. Trancoso, P. Evripidou, Exploring HPC Parallelism with Data-driven Multithreading, *Proc. of 2nd Workshop on Data-Flow Execution Models for Extreme Scale Computing*, 2012.

[5] V. W. Freeh, G. R. Andrews, A Sisal Compiler for Both Distributed- and Shared-Memory Machines, Technical report, Department of Computer Science, University of Arizona, Tucson, 1995.

[6] C. Grelck, S.-B. Scholz, Sac - From High-Level Programming with Arrays to Efficient Parallel Execution, *Parallel Processing Letters*, 13(3):401–412, 2003.

[7] J. Guo, J. Thiyagalingam, S.-B. Scholz, Breaking the GPU Programming Barrier with the Auto-parallelising SAC Compiler, *Proc. of 6th Workshop on Declarative Aspects of Multicore Programming*, pages 15–24, 2011.

[8] G. Gupta, G. S. Sohi, Dataflow Execution of Sequential Imperative Programs on Multicore Architectures, *Proc. of 44th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 59–70, 2011.

[9] J. R. Gurd, P. Barahona, Simulated Performance of the Manchester Multi-ring Dataflow Machine, *Parallel Computing*, pages 419–424, 1985.

[10] J. R. Gurd, C. C. Kirkham, I. Watson, The Manchester Prototype Dataflow Computer, *Communications of the ACM*, 28(1):34–52, 1985.

[11] A. Hagiescu, H. P. Huynh, W.-F. Wong, R. S. M. Goh, Automated Architecture-aware Mapping of Streaming Applications onto GPUs, *Proc. of 25th IEEE International Symposium on Parallel and Distributed Processing*, pages 467–478, 2011.

[12] F. Hubertus, R. Russsel, M. Kirkwood, Fuss, Futexes and Furwocks: Fast Userlevel Locking in Linux, *Proc. of 4th Ottawa Linux Symposium*, pages 479–495, 2002.

[13] H. P. Huynh, A. Hagiescu, W.-F. Wong, R. S. M. Goh, Scalable Framework for Mapping Streaming Applications onto Multi-GPU Systems, *Proc. of 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 1–10, 2012.

[14] B. Jang, P. Mistry, D. Schaa, R. Dominguez, D. R. Kaeli, Data Transformations Enabling Loop Vectorization on Multithreaded Data Parallel Architectures, *Proc. of 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 353–354, 2010.

[15] C. Kyriacou, P. Evripidou, P. Trancoso, Data-driven Multithreading Using Conventional Microprocessors, *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1176–1188, 2006.

[16] L. L. Laboratory, J. McGraw, S. Skedzielewski, S. Allan, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce, R. Thomas, *SISAL: Streams and Iteration in a Single Assignment Language. Language Reference Manual*, Lawrence-Livermore-National-Laboratory, 1985.

[17] P. Miller, Sisal Lives, http://sisal.sourceforge.net/, 2012.

[18] J. Serot, Tagged-Token Data-Flow for Skeletons, *Parallel Processing Letters*, 11(04):377–392, 2001.

[19] K. Stavrou, M. Nikolaides, D. Pavlou, S. Arandi, P. Evripidou, P. Trancoso, TFlux: A Portable Platform for Data-driven Multithreading on Commodity Multicore Systems, *Proc. of the 37th International Conference on Parallel Processing*, pages 25–34, 2008.

[20] K. Stavrou, D. Pavlou, M. Nikolaides, P. Petrides, P. Evripidou, P. Trancoso, Z. Popovic, R. Giorgi, Programming Abstractions and Toolchain for Dataflow Multithreading Architectures, *Proc. of the 8th International Symposium on Parallel and Distributed Computing*, pages 107–114, 2009.

[21] T. Sterling, J. Kuehn, M. Thistle, T. Anastasis, Studies on Optimal Task Granularity and Random Mapping, *Advanced Topics in Dataflow Computing and Multithreading*, pages 349–365, 1995.

[22] P. C. Treleaven, D. R. Brownbridge, R. P. Hopkins, Data-Driven and Demand-driven Computer Architecture, *ACM Computing Surveys*, 14(1):93–143, 1982.

[23] B. M. Tudor, Y. M. Teo, A Practical Approach for Performance Analysis of Shared-memory Programs, *Proc. of 25th IEEE International Symposium on Parallel and Distributed Processing*, pages 652–663, 2011.

[24] B. M. Tudor, Y. M. Teo, S. See, Understanding Off-chip Memory Contention of Parallel Programs in Multicore Systems, *Proc. of 40th International Conference on Parallel Processing*, pages 602–611, 2011.

[25] Z. Wang, M. F. O'Boyle, Mapping Parallelism to Multi-cores: a Machine Learning Based Approach, *Proc. of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 75–84, 2009.