Improved Dataflow Executions with User Assisted Scheduling

Daniel Goodman* Behram Khan Mikel Luján Ian Watson
University of Manchester
Oxford Road
Manchester, UK
{*goodmand, khanb, mlujan, watson*}*@cs.man.ac.uk*

***Abstract*—In pure dataflow applications scheduling can have a huge effect on the memory footprint and number of active tasks in the program. However, in impure programs, scheduling not only effects the system resources, but can also effect the overall time complexity and accuracy of the program. To address both of these aspects this paper describes and analyses effective extensions to a dataflow scheduler to allow programmers to provide priority information describing the preferred execution order of a dataflow graph. We demonstrate that even very crude task priority metrics can be extremely effective, providing an average saving of 91% over the worst case scenario and 60% over the best case naive scenario. We also note that by specifying the scheduling information explicitly based on the algorithm, not the hardware, we provide portability to the application.**

***Keywords*-Dataflow; Scheduling; Mutable State**

I. Introduction

In this paper we examine how user defined priorities can be used within dataflow programming models to streamline dataflow programs. This can reduce the levels of supporting resources required by the system and scheduler to support the dataflow program, and in the case of impure dataflow applications allow reductions to their execution time and improvements to their accuracy.

II. Why Dataflow

Physical limits have brought an end to the ever increasing performance of single core processors resulting in almost all devices now containing multiple cores. This change has forced parallel programming from being a niche area for a small number of specialists in high performance computing to a mainstream concern. The challenge of constructing parallel codes for this new target domain is compounded by the interactive unstructured nature of the required applications and the dynamic nature of the environments they execute in. For example a programmer is unable to accurately predict the future actions of a user, or the other demands on the shared resources of the environment the application is going to be used in. As a result the need for parallel applications is not only now far more ubiquitous, but also the class of programs is far harder to construct correctly. To attempt to address this complexity a large number of different programming models, libraries and tool suites have been proposed that attempt to close the gap between what programmers are capable of handling in a multi-core environment and what our existing programming models are able to offer.

In this paper we are going to consider how the scheduling strategies for the dataflow programming model [1], [2] can be extended to improve its performance and resource requirements. Dataflow programming is a model where the program is broken into blocks of code called tasks, each of which only uses a well defined set of immutable inputs and produces a set of immutable outputs. Depending on the granularity of the program these vary from a single instruction to whole functions which can include calls to other functions, allowing arbitrarily large computation units. These tasks and their relationship can be represented by a Directed Acyclic Graph (DAG) with the nodes representing the tasks and edges representing the data dependencies between these tasks. Hence the data flows through the graph as the program executes, an example of such a DAG can be seen in Figure 1. The presence of a well defined set of inputs allows tasks to be marked ready for execution once all the required data has been computed, and once started, to execute to completion without interruption. This means that unlike procedural code, where the execution is controlled by the progression of a program counter, dataflow programs are controlled by the flow of the data through the graph. As the program is a DAG by construction it is not possible for it to contain cycles, this coupled with the immutable nature of the data passed between tasks makes programs deadlock free and deterministic.

Dataflow programming has been shown to be very effective at exposing parallelism [1], [2], [3], [4], [5], [6] as the side-effect free nature means segments of code forming nodes whose inputs have been generated can be executed in parallel regardless of the actions of other segments in the graph and without effect on the deterministic result.

III. Dataflow and State

The deterministic and race condition free properties of pure dataflow programs make them very appealing as a

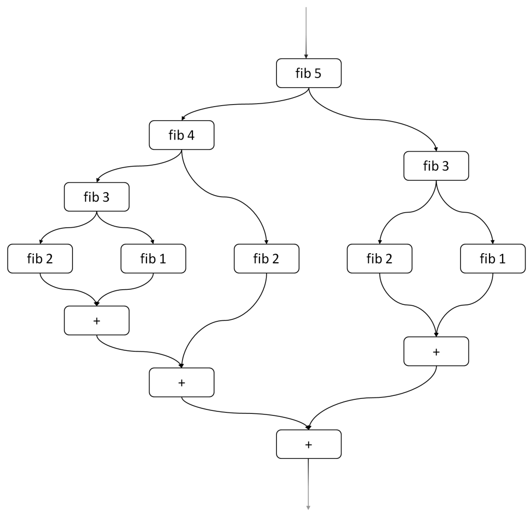


Figure 1. An example of a dataflow graph to calculate the 5th Fibonacci number.

means of constructing programs for multi-core processors. However, pure dataflow programs are limited by their determinism which prevents the construction of programs that would traditionally require shared state for either efficiency or to support unstructured interactions. Examples of such systems include banking systems, internet services and user applications. These limitations can be overcome by adding shared mutable state.

An example of a problem that requires shared state to be solved efficiently is the travelling salesman problem. This problem takes a connected graph as input, in which the nodes represent cities and the arcs represent roads with weights recording the distances between these cities, and returns a tour where every city in the graph is visited and the distance travelled is the shortest possible. Accurate solutions to this problem are used on a daily basis by logistics companies and saves millions of litres of fuel a year.

A brute force approach to this problem is not practical as there are $n!$ possible tours for n cities. Instead all efficient techniques for solving this require a shared updatable lower bound which is updated as better solutions are found. The presence of this lower bound allows these techniques to discard any solution that will exceed this lower bound before any further time is spent on it. As a result the most efficient versions are those which can quickly reduce the lower bound to represent the length of the shortest tour, and can efficiently calculate the lower bound for the partially constructed tours, so that large sets of tours can be discarded.

This change adds an additional required property to the scheduling strategy as unlike deterministic dataflow graphs where the total computation is constant, the amount of computation in this style of problem is dependent on the order in which the tasks get executed. This means that executing the tasks that will lower the bound by the largest amount first can reduce the complexity of the computation.

Ideally the compiler and runtime would be able to automatically determine the most appropriate order to run the tasks in. However, to achieve this would require access to the input data and a level of algorithm analysis that is currently not possible outside of very domain specific languages. Given our difficulties constructing self-parallelising compilers this is unlikely to be possible in the foreseeable future. Simple heuristics such as counting the number of tasks created by a task do not work, as in the case of the travelling sales person all tasks can run the same code just with different data, so all tasks would be placed in the same category.

Given the difficulties in producing automated analysis of programs the focus of this paper is simple user strategies for the providing hints to the runtime system about the order in which tasks should be executed for best average case running of an algorithm.

IV. Dataflow and Scheduling

If an unlimited number of cores are available scheduling is very simple, every time a task has all the required inputs it is assigned to a core for execution. However, as real machines do not have an unlimited number of cores some form of queuing strategy will be required if there are more tasks ready to run than cores available to execute them. Tasks waiting for inputs along with the arguments they have received will also need to be stored. As dataflow programs will typically contain many more smaller tasks than the tasks that make up a conventional program, queuing strategies can have a substantial effect on the behaviour of the program. For example, the strategy used for the queuing can affect the processor utilisation over the course of the application, the size of the pool of ready tasks, the efficiency of the memory architectures, and in the case of impure dataflow programs the accuracy and rate of convergence. We will now briefly look at each of these points.

Processor Utilisation If the critical path of any given part of the dataflow graph is insufficiently explored there is a risk that an insufficient number of tasks will be exposed for execution later. This shortfall will prevent the processor being fully utilised.

Pool Size Conversely if tasks which generate 2 or more tasks are executed at a greater rate than tasks that do not then the size of the queue can grow such that the scheduling is impeded, slowing down the overall performance of the application. For example the queue may no longer fit into the cache available.

Memory Size Every task requires its arguments be stored so that they are available to use. Once all the tasks that require those arguments have been executed, the space they occupy can be reclaimed. As such the memory footprint of a program can vary greatly depending on the order that tasks are executed in.

Memory Architecture The additional information provided by dataflow programming about the use of data allows for memory hierarchies to be used more effectively, reducing the level of cache stalls.

Accuracy and Convergence As discussed in Section III the order that dataflow tasks are executed in stateful dataflow programs can also effect the time to converge on a solution, or the accuracy with which a solution is produced.

V. Experiment

To generate a qualitative assessment of the difference that the scheduling order can make to a dataflow execution and to experiment with how such priority information may be cleanly added to the dataflow programs we used a modified version of DFScala [7] with the MUTS [8] software transactional memory to add mutable state. We then constructed several benchmarks with support for prioritised tasks and monitored the following parameters:

- Number of tasks ready for execution in the scheduler.
- Number of tasks waiting to receive all their inputs.
- Total number of tasks used in the execution of the application.

We compare the results against three naive scheduling strategies, FIFO, LIFO and Random.

- First In First Out, (FIFO) is a queue based strategy where ready tasks are executed in the order that they become ready.
- Last In First Out, (LIFO) is a stack based strategy where the task that most recently became ready is executed next.
- Random, to check that any improvements are not simply because of a randomisation the ordering of the tasks we also include a strategy where the next ready task to be executed is picked at random.

A. DFScala

DFScala [7] is a library for implementing dataflow programs in Scala [9]. Developed as part of the Teraflux project [10] it enables the construction of coarse grained dataflow programs with comprehensive type checking, nested parallelism and support for debugging tools. Importantly DFScala also allows the passing of Tokens which act as proxy's to Task arguments, removing the need to pass around tasks. This allows for simplified type signatures of methods receiving Tokens and for a less restricted style of programming.

B. Extensions

We extended the object that represents a dataflow task to include a numeric value defining its priority with higher values representing higher priorities and lower values representing lower priorities. By default tasks will have a priority of 0 representing a neutral priority. We then replaced the queue used by the scheduler with a priority queue ordered using the task priority element.

Having added the priorities and a mechanism to use the priorities we required a mechanism to set them. For this we implemented three options:

- We constructed setter and getter methods that allow a tasks priority to be assigned directly. This would typically be done by the task when the task is created, but could be done by any code that has access to the task prior to it receiving its last argument.
- The priority that a task should execute with may be dependent on the values of the arguments that it receives. However if the value is being assigned to a Token, the code performing the assignment will not be able to set the task's priority. Furthermore it may not be appropriate for the assigning code to also assign the priority. For example if it is some piece of general purpose code that should not be closely coupled to the task it is assigning to. To address this we added the ability to designate a priority argument. The task will then take whatever value is assigned to this argument and use that as its priority. Again this would typically be done when the task is created, but could be done by any part of the code that has access to the task.
- Building on this last option, the priority may not be represented by a single argument, or the argument may need to be a changed to represent the priority, for example, inverting the sign or converting the argument to an integer value. This is accomplished by passing the task a function that takes a task as its input and produces an integer output. When the task receives its last argument and becomes ready to execute this function will be run taking the task as its argument and the result will be set as the task priority. By default this function is the identity function.

Examples of all three of these can be seen in Figure 2. As dataflow tasks will only execute once all their inputs have been computed, this does not introduce a risk of priority inversion.

C. Benchmarks

The 4 benchmarks used to evaluate the different schemes are: A branch and bound Travelling Salesman Problem; a Monte Carlo Tree Search based Go playing A.I.; a recursive Fibonacci number calculator; and 0-1 Knapsack. For each benchmark we provide one or more strategies for assigning priorities to tasks.

1) Travelling Salesman Problem: As described earlier the travelling salesman problem requires the construction of a tour of a connected graph such that the tour visits every node in the graph and no shorter tours that fulfil this property exist. An example can be seen in Figure 3.

```
// Create some tasks
val t1 = createTask(foo _)
val t2 = createTask(foo _)
val t3 = createTask(foo _)

// Assign a priority of 5 to t1 directly
t1.priority = 5

// Instruct t2 to use the value passed to
// its third argument for its priority.
t2.setPriorityArg3

// Provide a function to calculate the
// priority of a task from the tasks
// arguments. In this case multiplying
// argument 2 by the length of the list
// that appears as argument 3.
t3.priority(
  (t:DFTask3[_, Int, List[String], _])
    => {t.arg2*t.arg3.length}
)
```

Figure 2. Examples of the different techniques for setting a tasks priority. All of these methods type check, generating errors if there is a problem.

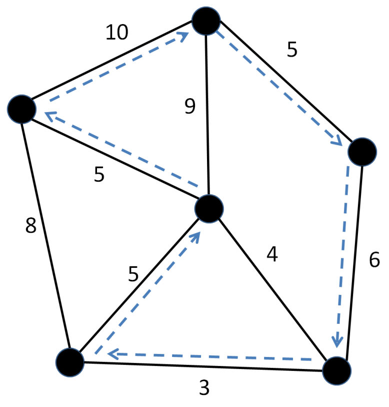


Figure 3. An example of a tour of weight 34 in a graph containing 6 nodes.

The particular implementation of branch and bound search used in this benchmark recursively takes as its input a table listing the distances between each of the n cities and a description of any parts of the tour that have already been constructed. Then if the complete tour is yet to be constructed and can still be constructed it uses a heuristic to select an edge and starts two searches, one in which this edge has been included in the tour, and one in which this edge has been removed from the graph. Each of these searches is started as a separate dataflow task. When a complete tour is constructed, it is compared to determine if it is better than the current best tour, if it is an improvement then the global

```
solve(Tour t, Graph g)
{
  //Check if the tour is complete or
  //is not possible.
  if(complete(t) || !connected(g))
    //Check the tour is an improvement
    if(improvement(t)
      updateBestTour(t)
  else {
    e = selectEdge(t, g)

    //Use the edge in the tour
    t' = addEdge(t, e)
    if(getBound(t', g) < Gobal.bound)
      solve(t', g)

    //Remove the edge from the graph
    g' = removeEdge(g, e)
    if(getBound(t', g) < Gobal.bound)
      solve(t, g')
  }
}
```

Figure 4. Pseudo code outlining the algorithm for solving the travelling sales man problem.

state is updated to reflect this. Pseudo code representation of the core of the algorithm can be seen in Figure 4.

Minimum Bound Selection: Each partial solution has a lower bound used to determine if it is worth continuing to construct tours based on this solution. This is calculated by taking the distance of the tour currently constructed and adding the values of the shortest remaining links that could be used to ensure that every node is entered and exited once. While this bound guarantees that all nodes are visited it does not guarantee that there is one tour, not two or more disjoint tours. As such this value is only a lower bound and will normally increase as the tour is constructed and these sub tours are merged into a single tour.

Scheduling Strategies: **Lowest Bound First** With this ordering we choose the task that has the lowest lower bound first. The logic for this is that the quicker the global lower bound is reduced, the more aggressive the pruning of the search tree can be, and so choosing to evaluate the problem with the lowest bound may achieve this. However, as the lower bound of a tour will rise the nearer it gets to the completion, this is not necessarily true as this can just result in prioritising the less compete tours.

Use First This strategy picks a task where the tour last added an edge in preference to a task that last removed an edge. This is done for two reasons: First the edge was chosen because the heuristics in the algorithm predict that it will be in the shortest tour; second, adding just n edges is required to construct a tour, but just removing edges requires

$n^2 - 2n$ steps to construct a tour. As such the route by which the tour is constructed will have a significant effect on the overall runtime of the application. This property is true of many stateful applications.

2) Go A.I.: This benchmark uses a Monte Carlo Tree Search algorithm [11] to select the next move in a game of Go. Unlike the other benchmarks, this benchmark does not compute a guaranteed correct solution, instead it uses the available compute time to produce an estimate of the best next move.

This is done by getting many tasks to explore a tree of possible moves randomly, spawning more tasks if a sequence of moves looks promising. After sufficient exploration each of these tasks will use an evaluation function, in this case by playing a game using conventional solvers, to determine is this position is advantageous or not. After a predetermined number of runs a move is selected from the top of the tree by picking the move that is most like to produce a winning series. Heuristics are used to guide the exploration of the tree towards the more favourable options. These include heuristics based on the results already returned. As such returning results back to the tree as soon as possible after they are computed so they can guide the tree exploration is advantageous. However, to avoid congestion at the root of the tree results are not written back using the same task that computed them, but are grouped to have their cumulative result be written back by another task.

Scheduling Strategies: For this benchmark we implemented just one scheduling strategy. This prioritises write back tasks over all other types of task. It also groups all tasks returning results to a given write back so they are executed together by giving all tasks in a group the same unique priority.

3) Recursive Fibonacci: As the name suggests Recursive Fibonacci uses recursion to calculate the Fibonacci numbers. In this case the 25th Fibonacci number. There are much more efficient ways of calculating Fibonacci numbers, but this technique is an effective means of generating a large dataflow graph in which a very large number of tasks can be active at any one time. An example of the graph can be seen in Figure 1.

Scheduling Strategies: We tested two different Scheduling strategies:

- Strategy 1 attempts to control the number of active tasks by prioritising the additions over the calculation of the lower Fibonacci numbers as these tasks do not generate any further tasks.
- Strategy 2 extends this by always calculating the task with the smallest Fibonacci number first.

4) 0-1 Knapsack: 0-1 Knapsack is an optimisation problem requiring items to be picked from a set such that their weight does not exceed a given bound while maximising the overall value of the items. This benchmark solves the problem through the use of dynamic programming. The resulting table is split into blocks, with each block being dependent on the blocks to its left and above it as shown in Figure 5. In this instance there are 1000 items and a maximum weight of 10,000.

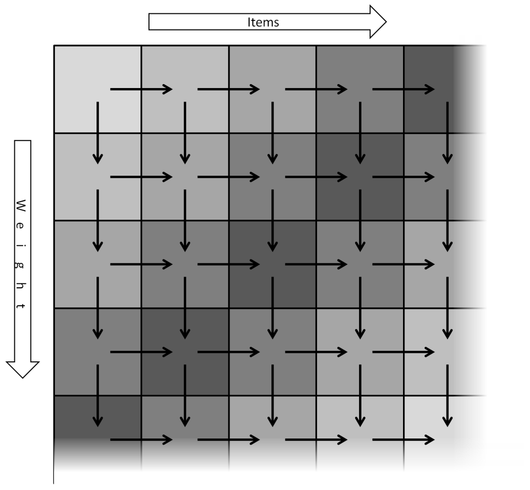


Figure 5. The data dependencies when using dynamic programming to solve a 0-1 Knapsack problem.

Scheduling Strategies: We tested two different Scheduling strategies:

- Strategy 1 prevents tasks being left behind and controls the rate at which new tasks are created by ensuring each diagonal row has all its tasks completed or executing before the tasks in the next row can begin executing.
- Strategy 2 prioritises the completion of entire rows. This does not prevent the computation of multiple rows simultaneously, but means that there will normally be no more rows being computed than the number of available cores, and only 1 row of tasks will be waiting for execution at any time.

VI. Results

We will now consider the results of running these benchmarks. These are presented in Figures 8, 6 and 7. Figures 8 shows that the improvement provided by the different strategies cannot be attributed to randomisation and that no naive strategy performs best for all the benchmarks. For pure programs the priority based strategies are able to outperform even the best naive solutions. With the Go benchmark the priority based solution is matched by a pure LIFO. Only for the TSP benchmark are less scheduling resources used by a naive solution than a priority based one, but these priority based solutions are aiming for convergence speed, and the resource requirements are simper to those of other benchmarks. As expected, using the lower bound to set the priority for TSP results in a concentration on the shorter tours and under performs. This highlights the need for

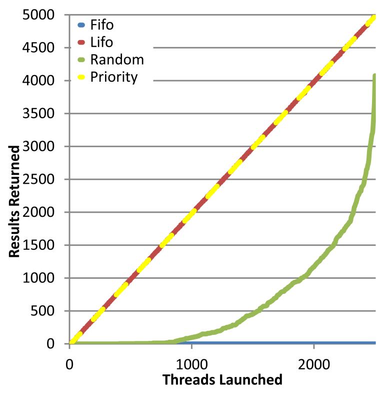


Figure 6. A graph showing the number of returned results when each task starts under the different scheduling strategies. Higher is better.

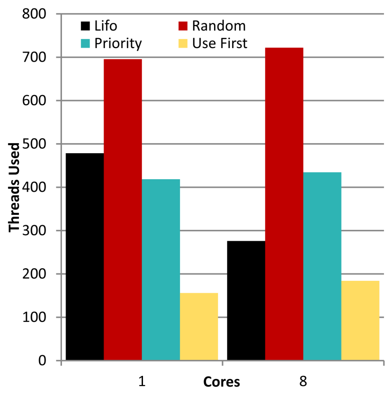


Figure 7. A graph showing the average number of tasks used and there in proportionately the execution time for the travelling salesman benchmark with the different scheduling strategies. FIFO used an average of 3072, so has been removed for reasons of clarity. Lower is better.

profiling of dataflow graphs to ensure that they are behaving as expected.

Figure 6 shows that the priority based scheme equals the best return rate for the naive solutions, providing the highest possible levels of accuracy by making the earlier results available to the later tasks. However, unlike the naive solution, the prioritised version provides assurances that this behaviour is reliable and predictable.

Figure 7 shows that the simplest priority strategy can provide over 3 times improvement in convergence over the best naive strategies.

A. *Processor Utilisation*

Looking at the risk of there being insufficient tasks, the results show that none of the scheduling strategies tried resulted in an insufficient number of tasks being available to exercise all of the cores in the system. This is expected as typically dataflow programs will generate far more tasks than there are cores to execute them, and we are only effecting which of these many tasks are executed first. As we add more cores these execute more tasks which then generate more tokens and more tasks resulting in an increase in the number of available tasks. Therefore as we increase the number of tasks we also increase the number of available tasks to run on these cores. This is particularly clear when looking at the row based strategy in for 0-1 Knapsack. Here adding extra cores increases the number of ready tasks proportionately as extra rows are made available to be computed in parallel.

B. *Resource Utilisation*

The results show that the scheduling strategy can make an enormous difference to both the number of tasks in the system as a whole and the number of ready tasks that the scheduler has to handle at any given time. No single naive scheduling strategy performed best with all the benchmarks and our priority based scheduler was able to equal or outperform the best result performance produced by the naive scheduling algorithms for all but the TSP benchmark, but for this benchmark we targeted quickest convergence instead of least storage and tasks. The average reduction in required scheduling resources can be seen below.

	Waiting	Ready	Total
FIFO	74.0%	91.1%	90.1%
LIFO	57.2%	59.7%	59.6%
Random	96.7%	59.6%	86.3%

With peaks of over 98% for Go and Fibonacci against FIFO and 91% for knapsack against LIFO. This represents a significant reduction in required system resources both for scheduling and to store transient results.

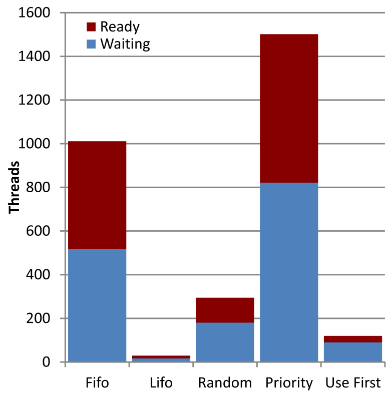
C. *Convergence and accuracy*

Aside from resources required to store and manage all dataflow programs we demonstrate how for impure dataflow programs, programmer assigned priorities can improved the performance and accuracy with the Travelling Salesman and Go Playing benchmarks. The results of these can be seen in Figures 6 and 7. They show that for the Travelling Salesman, the addition of task priorities reduced the number of steps required to reach a solution by 68% relative to the best naive scenario. As this is information that it will not be possible to automatically derive from the program, this clearly illustrates the need and benefit of providing such a system.

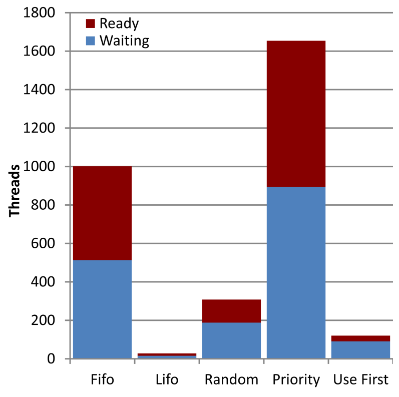
The results from the Go benchmark show that prioritising short tasks that return results over tasks that are computing further results allows the later tasks to take full advantage of the existing results, while in the worst case scenario, a FIFO queue will both maximise the required system resources and prevent any of the results being returned before the evaluations have been started.

VII. Conclusion

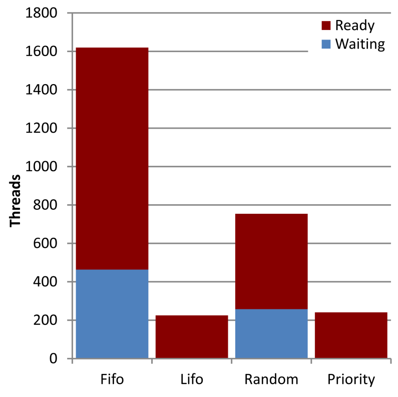
The need for impure dataflow graphs to handle many real world problems adds a new dimension to dataflow



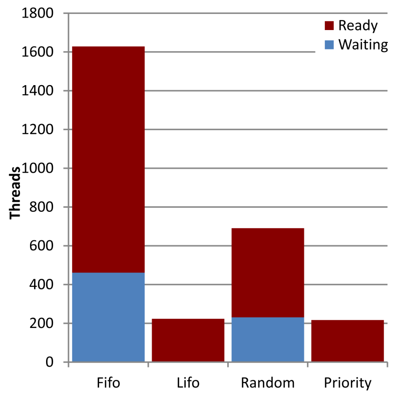
(a) TSP 1 Core



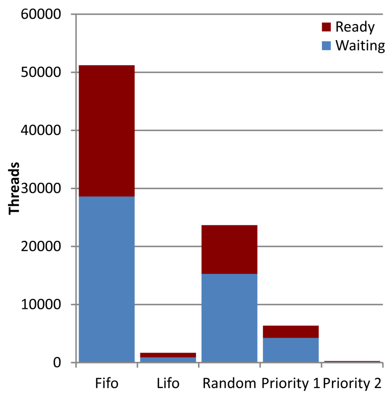
(b) TSP 8 Cores



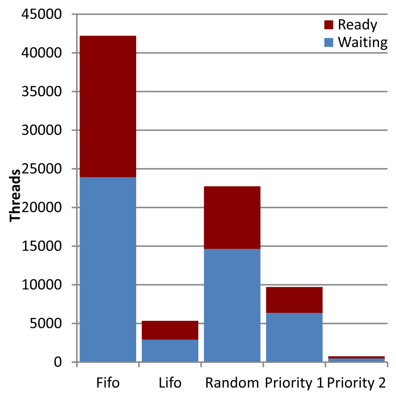
(c) Go 1 Core



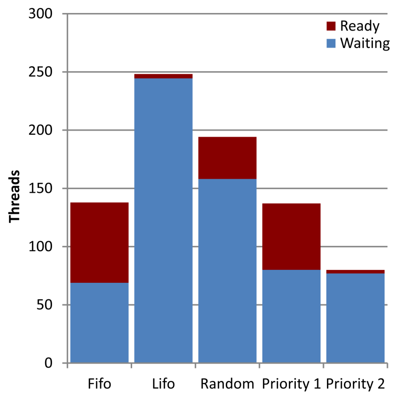
(d) Go 8 Cores



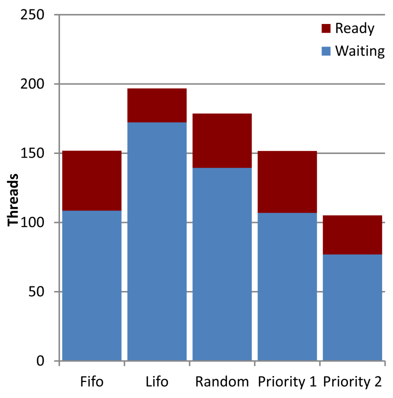
(e) Fibonacci 1 Core



(f) Fibonacci 8 Cores



(g) 0-1 Knapsack 1 Core



(h) 0-1 Knapsack 8 Cores

Figure 8. Average queue sizes using different strategies and 1 or 8 cores for each of the benchmarks. As all of these include sufficient tasks to fully exercise all available cores, lower is better in these graphs.

scheduling as now the order of execution can also affect the accuracy and execution complexity. We demonstrate this with a Go A.I. where scheduling effects the speed with which results are returned and therein their availability to other tasks to better inform their move selection as shown in Figure 6, and with TSP where scheduling reduces the number of steps to convergence as shown in Figure 7.

This paper has shown how a simple interface can allow programmers to provide priority information to tasks in dataflow programs. This information can reduce the system resources required to store and manage tasks and their arguments that are waiting to execute. Further to the saved resources, for impure dataflow graphs, this information can improve the speed of convergence and accuracy of results.

To demonstrate these effects we have constructed four benchmarks, two pure and two impure. These have shown an average saving of 91% over the worst case scenario and 60% over the best case naive scenario. As this scheduling information is explicitly provided, this information also improves performance portability as the strategy is maintained between runtime systems and can be incorporated into other scheduling policies something that was not previously possible.

VIII. Related Work

Maintaining bounds on the number of active tasks is a problem that dates back to the earliest dataflow machines [3], [6]. Initial attempts to maintain a bound on the number of concurrent tasks was done by adding explicit control statements to the program that added false dependencies to restrict the breaking up of the program into tasks. Unfortunately this was, for the most part, unsuccessful; producing difficult to program and fragile solutions that were highly dependent on the hardware and software environment that they were operating in. For example in order to maintain a sufficiently small number of tasks in one scenario the dependencies could be so constraining that in another scenario, or on another machine, there would be an insufficient level of parallelism.

Other software solutions that target specific areas include k-bounded loops [12] and dynamic software throttling [13]. K-bounded loops prevent iterative statements from exceeding the bounds of a specific machine by only allowing a predetermined number of iterations of a loop to be turned into tasks at a given time. While effective, this only works for iterative statements. Dynamic software throttling generates two versions of a program one which is serial and one which is parallel, and execution switches between them depending on whether more or less parallelism is required.

Ultimately it was felt that hardware based solutions were required, so to ensure that the programs remained within bounds, schedulers would switch between a queue based task selection (FIFO) and stack based task selection (LIFO) to either grow or shrink the pool of available tasks [14]. However, this is problematic as stack based selection grows the set of ready tasks if the problem is iterative and shrinks it if the problem is recursive, conversely queue based selection grows the set of ready tasks for recursive problems and shrinks it for iterative problems. This situation is compounded further by the fact that the set may not actually shrink with either strategy, but may simply grow less quickly.

Acknowledgment

The authors would like to thank the European Community's Seventh Framework Programme (FP7/2007-2013) for funding this work under grant agreement no 249013 (Teraflux-project). Dr. Luján is supported by a Royal Society University Research Fellowship.

References

[1] D. Cann, "Retire Fortran?: a debate rekindled," *Commun. ACM*, vol. 35, pp. 81–89, August 1992.

[2] C. Kyriacou, P. Evripidou, and P. Trancoso, "Data-driven multithreading using conventional microprocessors," *IEEE Transactions on Parallel and Distributed Systems*, vol. 17, no. 10, pp. 1176–1188, 2006.

[3] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," *Commun. ACM*, vol. 28, pp. 34–52, January 1985.

[4] S. L. Peyton Jones, C. Clack, J. Salkild, and M. Hardie, "Grip: A high-performance architecture for parallel graph reduction," in *Proc. of a conference on Functional programming languages and computer architecture*, 1987, pp. 98–112.

[5] I. Watson, V. Woods, P. Watson, R. Banach, M. Greenberg, and J. Sargeant, "Flagship: A parallel architecture for declarative programming," in *ISCA*, 1988, pp. 124–130.

[6] J. Darlington and M. Reeve, "Alice a multi-processor reduction machine for the parallel evaluation cf applicative languages," in *Proceedings of the 1981 conference on Functional programming languages and computer architecture*, ser. FPCA '81, 1981, pp. 65–76.

[7] D. Goodman, S. Khan, C. Seaton, Y. Guskov, B. Khan, M. Luján, and I. Watson, "DFScala: High level dataflow support for Scala," in *2nd International Workshop on Data-Flow Models For Extreme Scale Computing (DFM)*, 2012.

[8] D. Goodman, B. Khan, S. Khan, M. Luján, and I. Watson, "Software transactional memories for Scala," *Journal of Parallel and Distributed Computing*, vol. 73, no. 2, pp. 150–163, Feb. 2013. [Online]. Available: http://dx.doi.org/10.1016/j.jpdc.2012.09.015

[9] M. Odersky, L. Spoon, and B. Venners, *Programming in Scala: [a comprehensive step-by-step guide]*, 1st ed. USA: Artima Incorporation, 2008.

[10] *The TERAFLUX project, http://www.teraflux.org*, 2010.

[11] G. Chaslot, M. H. M. Winands, and H. J. van den Herik, "Parallel monte-carlo tree search," in *Computers and Games*, 2008, pp. 60–71.

[12] Arvind and D. E. Culler, "Managing resources in a parallel machine," in *Proc. of the IFIP TC 10 working conference on Fifth generation computer architectures*, 1986, pp. 103–121.

[13] in *Functional Programming Languages and Computer Architecture*, ser. Lecture Notes in Computer Science, J.-P. Jouannaud, Ed., 1985, vol. 201.

[14] C. A. Ruggiero and J. Sargeant, "Control of parallelism in the manchester dataflow machine," in *Functional Programming Languages and Computer Architecture, Portland, Oregon, USA, September 14-16, 1987, Proceedings*, ser. Lecture Notes in Computer Science, G. Kahn, Ed., vol. 274. Springer, 1987, pp. 1–15.