Work-stealing optimizations for Macro-Dataflow Runtime Systems[1]

Sağnak Taşırlar, Vivek Sarkar

Department of Computer Science, Rice University
6100 Main Street, Houston TX 77005
{sagnak,vsarkar}@rice.edu

***Abstract*—Modern parallel programming models perform their best under the particular patterns they are tuned to express and execute, such as MPI for bulk synchronous parallelism, OpenMP for loop-level parallelism, and Cilk for divide-and-conquer parallelism. In cases where the model does not fit the problem, shoehorning of the problem to the model leads to loss of both programmability and performance, most often by introducing unnecessary dependences. In addition, some of these models, like MPI, have a machine abstraction that by necessity is exposed in the solution to the application problem at hand.**

We postulate that an expressive parallel programming model should not over-constrain the problem it declares and should not require the application programmer to code for the underlying machine and thereby sacrifice portability. In our former work, we proposed the Data-Driven Tasks model for macro-dataflow parallelism, which represents an approach to expressive and portable parallelism that only requires the application programmer to declare the inherent dependences in the application. In this work, we build on another instantiation of macro-dataflow, the Open Community Runtime (OCR) with work-stealing support for directed-acyclic graph (DAG) parallelism.

This paper makes the following contributions. First, we demonstrate that the more portable macro-dataflow model can use DAG parallelism to match the performance of hand-tuned parallel libraries on today's architectures. Second, we address granularity optimizations for DAG parallelism and show how work stealing schedulers can be extended to deliver better performance in the presence of complex dependence graphs. Finally, we study the impact of locality optimizations on work-stealing schedulers for DAG-parallel applications.

***Index Terms*—futures; macro-dataflow; task parallelism; work-stealing**

I. Introduction

Most popular architectures of our day (x86, ARM, etc) are modifications to the original von Neumann machines[2]. Inherent to this design are: the program counter (what to do next) and the state of the memory (what has happened before). Imperative languages declare a sequence of instructions to the machine, where what will happen next and what has happened before are how programs are declared. Hence the evolution of programming languages favored imperative languages, as they provide a better fit for the underlying architecture.

Firstly, let us observe the implications of imperative programming before delving into parallelism. An imperative program consists of a sequence of actions declared by the programmer, where implicitly in between every statement is a state change in the underlying machine. To a programmer, the sequence in a program may seem an arbitrary choice and a different order could instead have been chosen just as easily[3]. However, from the viewpoint of the programming environment all the way from the compiler to the chip, that order is fixed, and is considered to be the true meaning of the program. Optimizations on all these levels have to reverse-engineer the programmer's intent and check if these statements can be reordered, or otherwise manipulated, to deliver better performance while maintaining the same semantics as the original program.

We argue that an imperative program is an arbitrary topological sorting of the inherent dependence graph of that program. This restricts optimization decisions to the motto: 'Everything is banned unless it is permitted by dependence analysis'. We instead propose dependence as a user-lever construct to allow better optimizations, and changing the motto to: 'Everything is permitted unless it is banned by dependence declarations'.

The conditions on ordering and state becomes more problematic, once we consider the implications of parallel programming for parallel architectures. Given a machine with N execution units, there are N program counters to decide what will happen next and this makes utilization (load-balancing) a bigger concern. Secondly, these N execution units can have an intractably large [4] number of possible states (schedules, and therefore states of the machine). Both these problems, in addition to the inherited pitfalls of imperative programming, led to parallel programming models that either constrain the expressiveness of the model for performance and safety of that particular subset, or relinquish a lot of control (but also performance pitfalls) to the programmer.

Additionally, given that the necessary ordering constraints between objects are declared as dependences, the legal schedules prevent any ordering hazards. Though, we burden the programmer with expressing these dependences, the programmer does not have to guess what the underlying semantics for

[1]The text in this paper has been derived from the first author's PhD dissertation[1] but has not been published in any other venue

[2]For the pedantic reader, they are modified Harvard architectures [2], [3]

[3]Any problem can be viewed as a partially-order set of tasks, where the partial order relation is the dependence relation. By the order-extension principle, there is at least one strictly total order relation (one legal topological-sort of this relation).

memory orderings are. The legal orderings must all obey the dependences specified by the programmer.

We argue that explicit dependence declaration allows further scheduling opportunities than popular programming models by not constraining how problems can be declared. In particular, first-order dependence constructs can enable any arbitrary task graph to be built.

A. Nested fork/join parallelism

Task parallel models introduce new constructs to control flow for explicit parallelism but in a finer granularity than thread based models, hence the name 'task'. The user declares parallel procedures and how they are synchronized, leaving the mapping and scheduling aspects to the underlying runtime system.

The most common task-parallel constructs are flavors of *fork* and *join*. A fork operation creates an alternate and parallel flow of control to the context from which it is called. In contrast, a join operation merges more than one flow of control into one. These constructs occur in thread based models, too but the differentiating factor is the granularity used for best practices.

Nested use of *fork* and *join* constructs yield task graphs that are series-parallel [5]. A series-parallel graph can be formulated inductively by defining a series or a parallel composition applied to two series-parallel graphs. The base case for this induction is the unit series-parallel graph, which only consists of a *sink* and a *source* node. Since series-parallel graphs are a subset of partially-ordered sets, one can think of sink and source nodes as greatest and least elements of a partial-order. A *series* composition of two series-parallel graphs, (g_1, g_2), merges the sink node of g_1 and the source node of g_2. A parallel composition of series-parallel graphs (g_1, g_2) merges the source node of both graphs as the resulting graph's source and merges the sink node of both graphs as the sink node of the resulting graph.

II. Overview of the Open Community Runtime (OCR)

In this paper, we build on the Open Community Runtime, henceforth abbreviated as OCR, as a macro-dataflow model to address the challenges stated in section I. The OCR API enables the (manually or automatically generated) client program to declare unrestricted DAG parallelism that is executed by the underlying runtime system. The separation of concerns provided by this API allows runtime research to be conducted on separate platforms with several different objectives.

A. Application Programming Interface

Open Community Runtime allows a client program to declare macro-dataflow parallelism using the following library calls, which will also cover the concepts utilized:

1) ocrTaskCreate: is used to create a parallel task. This task may have dependences declared via `ocrAddDependence` calls and preserved by the runtime system. Since the user may not know the underlying implementation, it is not safe to assume any implicit ordering among tasks or the permanence of the stack variables across stack invocations. This interface requires the user to pass the function to be executed, the function parameters and how many of them there are, and how many dependences it will eventually declare.

2) ocrEventCreate: This function creates an *event* object, which can be used to declare dependences between tasks. The event construct is a more general version of a future, in that it does not know its producer or the value it will carry. Since events are the dependence abstraction for this model, they are single assignment, i.e., they can only be satisfied once.

3) ocrDbCreate: This function is used to create a *data-block*. A data-block can be described as a contiguous chunk of memory managed by the runtime. They can be used to satisfy events, declare data-dependences, and support race-free data accesses. All data outside of data-blocks are user-managed, and could be the source of data races as a result of programming errors.

4) ocrEventSatisfy: As events can be used to build a dependence graph, this interface informs the runtime that the dependence has been satisfied. If the dependence is a data-dependence, this function declares what data is flowing through this dependence via data-blocks. If the dependence satisfied is not a data-dependence, the event may be satisfied with any object value.

5) ocrAddDependence: is how a task declares that it is a sink of a dependence that is passed as an argument. The user is required to enlist all the shared data across tasks to be declared as a dependence to guarantee safe access through the synchronization provided by dependences.

6) ocrScheduleEDT: is how a user declares that the listing of the dependences for a particular task is over and now the runtime can take control over it.

B. Runtime Library

Any runtime system that implements the OCR APIs described above, can be labeled an OCR library, thus there is not one single OCR library. For our explorations in this paper, we have used an OCR runtime library implementation that is heavily influenced by Habanero-C [6] that implements the user interface from OCR version 0.7 [7].

The runtime library is implemented in C, which does not natively support language constructs like abstract classes, interfaces or inheritance. Therefore we instead have provided a poor man's version for these constructs by providing base structures for modules that we anticipate the runtime implementer would have to extend, with function pointer tables mimicking a virtual function table. The modules are for data-blocks, events, task pools, workers, executors (abstracting the underlying execution unit), schedulers and *policy-domains* (abstracting a mini-runtime, for hierarchical runtimes).

1) Habanero-like runtime for OCR v0.7: The scheduling semantics for this implementation are as follows: an OCR task utilizes non-blocking scheduling semantics by requiring its registered dependent events to be satisfied with the data the task consume. In contrast common futures allow unrestricted resolution from any context, but if the resolving task has not

completed, they block. This preserves the calling context at the cost of delaying the continuation that is not dependent on the future's resolution and tying up an execution unit in case it is not the one resolving the future.

When an OCR task is declared, a frame to contain its context gets implicitly created, just like a common task. Additionally, a list of registered events gets passed to this task that serves as this task's synchronization frontier. Eagerly the task tries to register to the first unready event by iterating over its dependences, checking the satisfaction conditions. Once an unready event is reached, the task registers itself to that event and the control returns to the parent task. If all the dependences have been met at the time of creation, the task is simply passed onto the scheduler, just like a normal task would.

When an event is satisfied by the producer task creating a value to satisfy that event, the producer task grabs the list of pending tasks and iterates their synchronization frontier, as described in the paragraph above. If the pending tasks have all their dependences met, they are passed onto the scheduler, if not they linger in the heap to be picked by their following dependences' producers.

This scheduling follows the semantics of dataflow; the tasks are fired when their data becomes ready. Most parallel programming models require the data dependences to be met at the point of task creation, burdening the programmer to structure their code accordingly, following the spirit of the imperative-language causing topological-sort argument covered in the section I above.

Regarding other modules, like data-blocks, workers, executors and policy-domains, we implemented bare necessities. Data-blocks are implemented as wrappers for contiguous memory on a shared memory machine that does not move or get tracked by the runtime. Workers execute a loop of popping, work-stealing when pop is failing, executing extracted work, just like Habanero workers. Executors are abstracting the underlying cores with an attached PThreads instantiations. Policy models are not utilized as we have not needed explicit hierarchies for our observations.

III. Granularity Optimizations

Popular work-stealing runtimes for nested fork-join parallelism [8], [9] have utilized lazy task creation [10] in order to avoid the runtime being swamped by eagerly created tasks. Lazy task creation can be interpreted as a sequential-by-default depth-first exploration of the task tree, in which multiprocessor thread scheduling is achieved by taking tasks from the unexplored list in the depth-first traversal. Since this depth-first traversal has a much smaller frontier than an alternative traversal (e.g. breadth-first), the number of tasks available to the scheduler is tightly bound.

For example, in Multilisp, one of the earliest implementations of work-stealing schedulers, the stealing heuristic employed is to steal the *oldest task* from the victim, just like the Cilk [9] implementation that followed. An *oldest* task would be the task that would be the first task to make it to the backtracking list of a depth-first traversal by a worker, and therefore the last one to be utilized for further exploration.

In a series-parallel task graph, stealing a task that was put aside to be explored provides the source node of another series-parallel graph. By definition series-parallel graphs are recursive structures, and if a task is put aside to be explored, it can only come from a parallel composition of multiple series-parallel graphs.

For a divide-and-conquer algorithm with a cutoff, like the one depicted in figure 1, stealing the oldest task from a thread provides the coarsest grained series-parallel graph that thread has to offer. For simplicity, we depict the task graph as a tree with spawn/fork edges, and omit the mirror image consisting of join edges. As work starts dissipating from a single source node, stealing would build a binary reduction tree of splitting and mapping subsets of the task graph, as seen in figure 1. Cilk or Mul-T [10] implementations also depend on this property of probabilistic work-stealing. The increased granularity of steals reduces the number of steal attempts and improves performance since steal operations introduce more runtime overhead, contention and increase idle time.

We used work stealing runtimes for macro-dataflow models on our former work for dynamic scheduling and load-balance. However, since task graphs that may be expressed by macro-dataflow models are more general than series-parallel graphs the implicit granularity achieved by stealing the oldest task for work-stealing runtimes do not necessarily hold anymore.

In a series-parallel graph, stealing one task from the victim gives us the source node of another series-parallel graph. If you ignore the join edge symmetry, a series-parallel graph is tree. So a steal returns the root task of a tree of tasks. Since that root task enables all the descendent tasks, if none of them is stolen from the thief, the whole tree is executed on that thief. Therefore stealing one task is analogous to stealing a subtree of a task tree. If (for the sake of this argument) we assume the computational cost of tasks are uniform, for a divide and conquer problem stealing the oldest task heuristic leads to stealing almost half the work available on the victim.

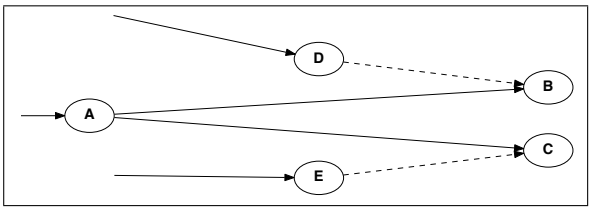

Fig. 2. Snapshot of a stolen $task_A$ and its immediate successors $task_B$ and $task_C$

In contrast, stealing a single task from an arbitrary DAG may not result in getting a root of a tree of tasks. Pathologically, a stolen task may not *dominate*[4] any of its descendant tasks. Let us observe figure 2. If a $task_A$ is stolen with successors $task_B$ and $task_C$, and if $task_B$ and $task_C$ each

[4]A $node_n$ *dominates* a $node_m$, if all the paths from the source of the graph to $node_m$ passes through $node_n$.

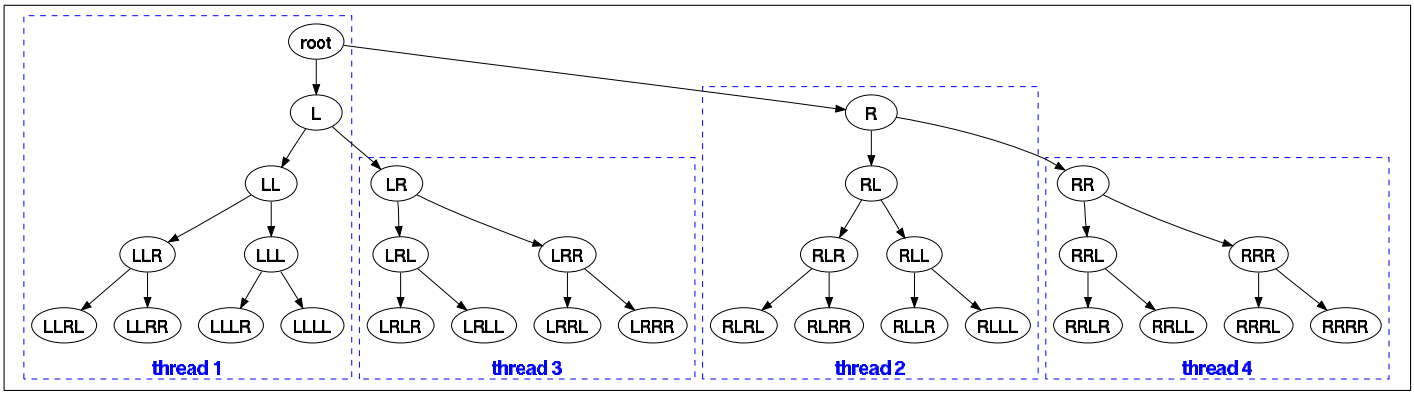

Fig. 1. Possible decomposition and mapping of a divide and conquer problem, figure credit [10]

depends also on $task_D$ and $task_E$ respectively where $task_D$ and $task_E$ are not $task_A$'s ancestors, based on a particular schedule $task_D$ and $task_E$ may not yet be available. In that case $task_A$ would not lead to any new descendant computations at all, and would eventually lead to another steal attempt.

If we wish to translate the implicit *steal half the work* policy for nested fork/join work-stealing models to macro-dataflow work-stealing in order to minimize steal attempts, we will need an explicit steal-half policy as in [11], [12]. For graph algorithms, granularity optimization to achieve half through batching can be employed [13].

For arbitrary DAG task graphs, one can still annotate or calculate the number of dominated descendants for a task. However, the number of descendants a task may lead to is schedule dependent. A task can lead to descendant tasks only if it satisfies their respectively last unsatisfied dependence, so dominance relation is a function of a runtime schedule. In contrast, for nested fork/join models, since tasks are roots of trees of tasks, the values can be computed bottom up or can be determined statically by counting the tasks they dominate. For divide-and-conquer problems we would have an almost balanced full binary tree, where stealing the oldest task converges to stealing half the work.

Since we can not calculate half the tasks on a dynamically unfolding task-graph with schedule dependent number of descendants, we restrict our heuristic to static assumptions, just like nested fork/join models. We explore two extremes of the spectrum on the number of descendant tasks, one pessimistic and one optimistic. A pessimistic heuristic assumes a task can only lead to a number of tasks it statically dominates. On the contrary, an optimistic heuristic assumes all the descendants will have all their other dependences satisfied by the schedule and a task can lead to all its descendants.

Experimental results for the granularity optimizations introduced in this section are included in Section VI.

IV. Locality Optimizations

In previous sections, we discussed why macro-dataflow parallel programming models support more general task-graphs than nested fork/join models. The support for more general task-graphs required us to address the granularity aspect of the underlying runtime work-stealing algorithm, which we covered in the previous section.

Work-stealing algorithms for series-parallel graphs not only have inherent granularity benefits, but also locality benefits. Let us recap the discussion on the granularity benefits of restricting task-graphs to series-parallel graphs and stealing the *oldest* task first. A thread exploring the task graph traverses the data structure in a depth-first fashion and enables unexplored paths to be stolen by idle threads. Series-parallel task graphs are declaring *control* dependences between tasks because of their imperative nature. Since data dependences have to have been satisfied for a child task at creation time, the control dependence graph is also a superimposed data-dependence graph. So as tasks get further decomposed deeper on the tree, these tasks' input data are also getting decomposed, and therefore the memory footprint is anticipated to get smaller. As data footprint gets smaller, the data is likelier to fit in the closest hierarchy in memory. As a thread traverses the task graph in depth-first fashion, and leaving a last-in first-out (*youngest* to *oldest*) trail of unexplored paths, the thread is executing tasks that are *closest* on the task graph topology. If the data decomposition *closeness* matches the task decomposition, this algorithm has tight locality bounds with a least-recently-used eviction policy. Additionally, since stolen tasks are also sources of series-parallel graphs (i.e. roots of task-trees), the properties hold for stolen sub-trees of tasks recursively.

We argued stealing the *oldest* task has granularity benefits and it is expected on average to have the tasks available for stealing from a thread are implicitly ordered from coarse to fine grain. The same property can be extended to locality. The newest tasks are likelier to consume data that is close to their sibling task's data that have left the cache favorably dirty for the newest tasks. If the cache holds more data and uses least recently used eviction policies, the cache is likelier to hold data from closer levels of their pedigree than further ones.

As macro-dataflow models utilize data dependence as a first-level construct, application programmers can declare computations that are unstructured DAGs, which are a superset of nested series-parallel graphs. Optimizing locality is more

challenging in the more general context of DAG parallelism, relative to fork-join parallelism [14]. We discussed for figure 2 that we can not statically deduce the availability of predecessors for a task, since in our model there can be more than one successor per task. A task can only become ready when all of its dependences are satisfied; the order these dependences may be satisfied is schedule dependent and can only be deduced at runtime. This impacts locality, because now the *closeness* in the task graph is also schedule dependent. Additionally the distance to a single predecessor, a constant, is sufficient for a locality metric for series-parallel graphs, where the distance metric is n-dimensional for a task with n predecessor tasks.

We will explore data-structures and policies adopted by work-stealing runtimes and propose ameliorations for better locality results for event-driven runtimes using work-stealing.

A. Explicitly prioritized data-structures

Work-stealing runtimes utilize deques for ready task data-structures, which provides an implicit granularity and locality prioritization for nested fork/join models. We argued that deques do not implicitly order tasks for locality and granularity for non-series-parallel graphs. Therefore, we postulate that we need task graphs where the ordering, or classifying, of tasks are more explicit and schedule dependent since locality metrics are schedule dependent for DAG parallelism.

One possible locality measure for tasks, is the dynamically calculated cost of bringing the data to consumer task. As our models have explicit producer/consumer relationships through dependences, and since shared data objects among tasks are also explicitly expressed, we can keep track of these dynamically at the runtime system. Therefore we can estimate where the data, that a task depends on, are and how much it would cost to bring all that data in at scheduling time for that task. If we employ a priority queue, instead of a deque, and use the cost of data movement per task as the priority, we will have a task queue that has a most local to least local ordering of tasks. These costs are computed when a task becomes ready and remains constant as long as it stays on the same queue. This simplifies the implementation at the cost of accuracy, but saves us from the cost we would incur from simulating caches.

One major assumption here is knowing where dependences have been satisfied and where the data that satisfies the dependence is located. Since dependences are monotonic structures that can be satisfied only once, we can deduce where the dependences have been provided. However, what matters is knowing where the data that satisfied the dependence lives. For simplicity, a one-to-one correspondence can be assumed initially, as if the 'producer' of the data is also the 'creator' or 'allocator' (or 'over-writer' if the dependence is a storage dependence). On a cache-architecture, if a data is consumed in two separate contexts, the data can be replicated to live in more than one place, making the placement tracking more complicated. For simplicity, we will initially focus on the intrinsic data transfer effects of locality and assume that each worker thread runs on a core with an unbounded eviction-free cache. This ignores the impact of capacity and conflict misses.

This locality ordering for a task per thread is 'dynamic', as it is delayed until creation time. However, another policy decision to make is to decide if the costs (and therefore priorities) should also change dynamically. One option is to simplify the design at the cost of inaccuracy and not update the cost of a task based on data eviction and replication, the other is to simulate a cache in software to update costs accordingly at runtime. Initially, we opt for not updating cost of a task once it is enqueued.

V. Task scheduling heuristics

Series-parallel programming models that constrain the task graphs expressed to trees, have a clear parent-child relationship between the tasks. They may employ eagerly executing a child task and leaving the continuation pending; this is labeled as work-first work-stealing policy and is a left-most depth-first traversal of the task graph. Alternatively, child tasks may be left pending as the continuation is being executed; this is the right-most depth-first traversal of the task graph, labeled the help-first strategy.

In our work, we explore the effect of non-parent/child predecessor/successor relationships and the challenge of scheduling tasks with these more general relationships. In our previous work [15], we did not provide an implementation to utilize work-stealing on non-series-parallel graphs. The implementation for work-stealing support for tasks with multiple predecessors is analogous to *currying*. Every predecessor of a tasks perform a *partial function application* till all the dependences are met. The last predecessor changes the descendant task to a zero arity function, making it *ready*. We treat this as if the last predecessor created a child task. Since we use a help-first policy on OCR, the descendant task gets pushed into the local ready task queue. The task graph frontier for execution is schedule-dependent because this currying is schedule dependent.

This simplification provides an implicit scheduling heuristic, namely *most recent satisfied dependence first*. When a task satisfies a dependence, for that dependence it walks through all the awaiting tasks synchronization frontiers. If all other dependences are satisfied for a task, that task is enabled and scheduled to be executed. Hence the last satisfied dependence is the enabling one, it leads to the enabled task to be enqueued on the same worker. Though this may help with simplifying the implementation and synchronization concerns, it may not necessarily provide the best performance, as this choice may not provide minimal data retrieval cost, and therefore locality performance. For example, a task with n dependences may have its first $n-1$ dependences met at $thread_i$, where the last one is met at $thread_j$, where $i \neq j$. If this task is executed on $thread_j$, all the necessary data may be replicated on $thread_j$, may incur possible cache misses and may evict data that is local to tasks waiting to be executed on $thread_j$. In experimental results, we will include results comparing pushes to 'closest' workers versus the *most recent satisfied dependence first* worker.

A. Task stealing heuristics

The differentiator of work-stealing runtimes from the rest, as the name implies, is utilizing the idle workers to do the load balancing by letting them steal tasks from busy workers. There are two-tiers of heuristics to stealing; first, victim selection (from where) and secondly, which tasks to extract from the selected victim.

1) Victim selection: A common implementation choice for a victim selection heuristic is the random victim selection [16], under the observation that a uniform random distribution reduces the average number of steal attempts to find work and is used to also to prove the theoretical bounds of work stealing. Though random victim selection has proven guarantees for load balance, it fails to address locality concerns. On a machine with a deep memory hierarchy, it may be favorable to prioritize the most local tier of workers to be the first set of victims. Then, the workers that are further in the memory hierarchy can be traversed as potential victims with lower priority.

a) Hierarchical traversal: As we suggested before, cache-based machines are mostly hierarchical, e.g. multiple threads sharing a cache, on a multi-cache socket, on a multi-socket machine. We can take this into account on our traversal to replicate the locality inherent to this hierarchy by traversing bottom-up for better locality as covered in [17]. Following the aforementioned sample machine, initially, a thread would try to steal from the threads it shares a cache with, then it would traverse threads it only shares a socket with and then explore threads on other sockets.

2) Task extraction: Once a victim is chosen, the second policy aspect to pin down is to decide which tasks to steal. We have argued for granularity optimizations in previous sections, in the absence of locality concerns. Taking locality optimizations into account does not invalidate the case for granularity. So for now, let us assume that we are employing the steal-half policy that we have been advocating.

Given that we are pursuing locality optimizations by classifying and ordering tasks in ready task queues, which changes task queue choices, which in turn changes how a steal-half heuristic would work. Secondly, another policy to consider is *which* half to steal. Granularity optimizations offered before did not differentiate between subsets of tasks, as long as they are half the size of the queue. With locality optimizations, we have an opportunity to decide *which* half.

a) altruistic stealing: Locality optimizations are pursued through ordering (partially or totally) ready tasks by how much data retrieval cost they would incur. By default and historical convention, in order to reduce the synchronization cost on the task queue, stealing is done from the alternate end of the queue, rather than the end the owner thread uses. So when a thief extracts half the work from the other end of an ordered queue, they are likely to get the tasks with high data retrieval costs to the victim, hence the name *altruistic*. This policy is achieved implicitly, just by using ordering for locality and by stealing with a coarse grain.

VI. Experimental Results

A. Fibonacci

We addressed how divide-and-conquer algorithms and languages that express series-parallel task graph match, and competitive performance can be achieved even with this ease of expression. We want to show that divide-and-conquer problems can also be easily expressed through declaring this restricted subset of possible directed acyclic graph and also provide competitive performance.

We will use an inefficient (non dynamic-programming) approach to calculating the N^{th} element of the Fibonacci sequence. The Fibonacci sequence can be described inductively as follows for non-negative values of n:

$$fib(n) = \begin{cases} fib(n-1) + fib(n-2) & n \geq 2 \\ n & n < 2 \end{cases}$$

$\#core$	1	2	4	8	18	36
$Cilk$						
$time(s)$	68.43	35.41	19.15	10.87	4.93	2.51
$speedup$		1.93	3.57	6.30	13.87	27.24
OCR						
$time(s)$	68.60	35.18	19.16	10.89	4.99	2.51
$speedup$		1.95	3.58	6.30	13.76	27.32

Fig. 3. $fib(50)$ with cut-off 25 for the Xeon

$\#core$	1	4	8	16	48	60
$Cilk$						
$time(s)$	78.65	19.69	10.07	5.05	1.76	1.44
$speedup$		3.99	7.81	15.59	44.79	54.54
OCR						
$time(s)$	78.69	19.70	9.91	4.95	1.67	1.34
$speedup$		3.99	7.94	15.91	47.15	58.60

Fig. 4. $fib(45)$ with cut-off 25 results for the XeonPhi

Figures 3 and 4 show the comparable performances of OCR and Cilk in execution time and scaling.

We argued that for a series-parallel task graphed benchmark our explicit steal half heuristics would converge to the default work-stealing algorithm. Our experimental results show that to be case within a 10% error margin for median number of steals. As there is no locality to be exploited for this application, we do not address locality heuristics for this benchmark.

B. Sequence Alignment

This benchmark is used in bioinformatics field to match amino-acid sequences of proteins, or nucleotide sequences of sites for local and global alignment to trace evolutionary paths, homology and etc. We see this benchmark frequently in the parallelism, high performance computing literature to showcase non-series-parallel graphs, and are known colloquially as the diamond graph. We can see the dependence graph of a string matching benchmark of 2 strings sized 4, or 2 strings

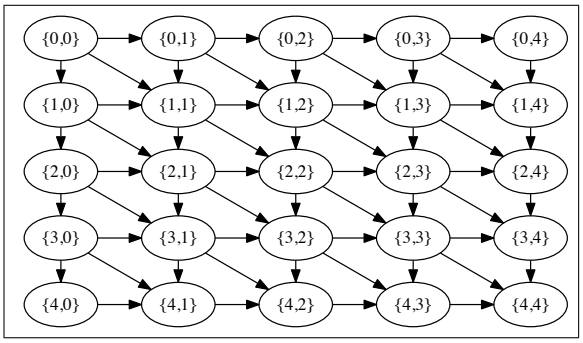

Fig. 5. Dependence graph for a 4 by 4 tiled string matching

with square tiles of quarter the size of the original string on figure 5. The first row and column is to keep track of the alignment with a gap instead of a nucleotide at the beginning of the first or the second string, hence the need for a fifth row and column.

In order to introduce some granularity to the leaf computation tasks, we tile the strings, so that a task calculates the values of a tile rather than a single entry. We performed a sweep for tile sizes to choose which tile size achieve the highest operations per second, and the results reported below use those tile sizes.

$\#core$	1	2	4	8	18	36
$time(s)$	39.69	20.35	11.14	6.34	2.15	1.52
$speedup$		1.95	3.56	6.27	18.44	26.06

Fig. 6. Matching strings of size 135K with tile size 576 for the Xeon using OCR v0.7

$\#core$	1	4	8	16	48	60
$time(s)$	135.32	34.02	17.13	8.68	3.23	2.74
$speedup$		3.98	7.90	15.59	41.96	49.4

Fig. 7. Matching strings of size 67.5K with tile size 432 for the XeonPhi using OCR v0.7

Explicit steal-half heuristics with totally/partially-ordered ready task queues do not have a drastic impact on the execution time and steal attempts for this benchmark.

One improvement is employing hierarchical work stealing by stealing amongst sockets first and across sockets after approach on a multi-socket Xeon machine. This provides at least a 17% decrease in L3 cache misses controlling for other parameters as in steal granularity or ready task queue data structure. Using data retrieval cost as a priority for tasks cut L2 misses 1%-2% for the minimum case and shortens execution time by 1% for the minimum of 10 runs. The impact of hierarchical work-stealing on this case ranged between -1% to 8% fewer L3 misses. Pushing tasks to the most local worker rather than the one enabling it, can provide 1%-2% percent reduction in L2 misses compared to the base case.

Putting all these together, minimum result for the median values for L2 misses are achieved by using priority queues using data retrieval cost as the priority metric, with flat victim selection for work-stealing and stealing half the work according to pessimistic descendance. The minimum result for execution time is achieved by using sorted priority queues, with hierarchical victim selection for work-stealing and stealing half the work according to optimistic descendance.

C. Cholesky Decomposition

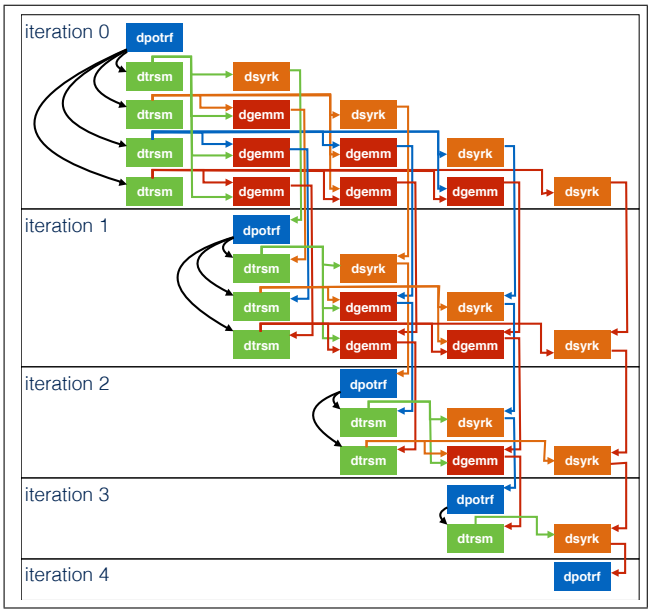

Fig. 8. Dependence graph for a 5 by 5 tile cholesky factorization

Given a symmetric, positive definite matrix A, cholesky decomposition calculates a lower triangular matrix L such that $A = LL^T$ and can be considered a special case of LU factorization where the upper triangular matrix is the lower triangular matrix's conjugate transpose. The computational complexity of the calculation is $O(n^3)$ and for a serial, in-place implementation the memory footprint is $O(n^2)$. Our parallel implementation through array-expansion, exposes the iteration-space as the third dimension and gets rid of the antidependences to expose further parallelism, which increases the memory footprint to $O(n^3)$.

The dependence graph of a 5 by 5 blocked cholesky factorization is depicted on figure 8 with tasks annotated with the LAPACK routines applied on said tiles. On a given iteration the top-left most tile has a sequential cholesky(`dpotrf`) applied, where that result enables a column of triangular solves(`dtrsm`) below it. A trisolve indexed i feeds data to triangular(`dsyrk`) or square(`dgemm`) matrix multiplications on row or column i. The resulting matrices of these matrix multiplications feeds in to the next iterations domain, depicted as vertical arrows crossing the iteration boundary on the figure.

As it can be seen on the figure, the dependence graph is an unstructured directed acyclic graph, and does not remotely resemble a series-parallel computation. Hence it is a motivating example for our macro-dataflow model.

We have implemented the benchmark with tasks serving as a wrapper to serial Intel Math Kernel Library(MKL) calls. Since MKL library calls are destructive writes to their input data, we use events to synchronize these writes on to the same data-block. Additionally, to provide coarser granularity into the tasks, we use a blocked version of the cholesky decomposition, where the tile size is a user provided runtime parameter. As auto-tuning and providing performance models for different architectures are not within the scope of this work, we do a tile sweep to calculate the tile size that gives the highest floating point operations per second(flops).

$\#core$	1	2	4	8	18	36
$OCR\&MKL$						
$time(s)$	16.58	8.41	4.25	2.37	1.25	0.67
$speedup$		1.97	3.91	6.98	13.29	24.95
$\|\|\ MKL$						
$time(s)$	13.87	6.85	3.48	1.94	1.07	0.75
$speedup$		2.03	3.99	7.14	13.02	18.47

Fig. 9. Cholesky decomposition results for a 12K by 12K matrix for the Xeon

$\#core$	1	4	8	16	48	60
$OCR\&MKL$						
$time(s)$	27.85	7.04	3.57	1.83	0.72	0.64
$speedup$		3.96	7.80	15.19	38.78	43.37
$\|\|\ MKL$						
$time(s)$	11.64	3.93	2.16	1.11	0.43	0.35
$speedup$		2.96	5.38	10.53	27.39	33.55

Fig. 10. Cholesky decomposition results for a 6K by 6K matrix with Intel MKL for the XeonPhi machine

Figure 9 shows that with a tile parameter tuned for the maximum core case, serial MKL kernels scheduled by OCR can out-perform a parallel MKL implementation in Xeon and get to half the flops for XeonPhi. The parallel MKL version has dynamic tiling which makes the speedup results look less stellar as it chooses the best tiles per available cores for execution.

For the Xeon runs of this benchmark, using a steal half the amount of ready work according to pessimistic or optimistic descendance relations lead to 4% or 5% improvement in throughput respectively for the 36 core case for median execution times of 10 runs. Number of steals are consistently fewer but by a 10% or 6% margin for the 36 core case for partially ordered task queues.

We did not observe a huge impact of steal half heuristic on XeonPhi for the highest throughput providing tile size. However we show that a smaller tile size which would burden the runtime more with more tasks show improvements when steal half heuristics are employed but not enough to surpass the highest throughput providing tile size for the default settings. We argue these heuristics will allow smaller tile size to be employed which will be more helpful as cores get smaller, simpler and plentiful.

Introducing hierarchical work-stealing cuts L3 misses by 8%-11% compared to the default settings. When we introduce data locality as priority, we see a -2% to 2% reduction for the median values for L2 misses. We also see a -3% to 3% reduction in median execution times for all cases but one. The overhead and contention introduced by priority maintenance could not be amortized by stealing a single task, so any competitive execution time utilizes a steal half heuristic.

If pushing tasks to most local workers, with retrieval cost as priority for task queues, are employed all at once, we see a 4% to 24% reduction in median values for L2 misses compared to the default case. For partially ordered task queues with data retrieval cost metric the median execution times are slowed down between 27% to 49% when compared to the default case. The results where totally sorted priority queues (and hence more precision) are used, fare much better. We observe an increase in execution time between 1% to 4% when compared to the default case.

VII. Conclusions

We propose macro-dataflow models as a general approach to expose parallelism by explicitly declaring dependences between tasks. In our former work on data-driven tasks [15], we built our model on top of work-sharing systems which may suffer from contention because of their centralized approach compared to the OCR-based work-stealing implementations studied in this paper.

In this work, we built our macro-dataflow model on top of a work-stealing runtime which implies decentralized scheduling and load-balancing. First, we show our macro-dataflow approach can declare programs both with simple dependence structures without a performance penalty, and also declare complicated dependence structures and surpass hand-coded parallel libraries in execution time tuned to the specific application.

We observe that the underlying assumptions of work-stealing runtimes on the nature of a program's task graph do not necessarily apply to complex dependence graphs. We address the granularity challenges by employing heuristics on the schedule-dependent descendance relations. We show reductions on number of steals to showcase better load balance and possible reduction in bandwidth for simple micro-benchmarks and also reduction in execution time for cholesky decomposition.

Lastly, we propose heuristics to the default work-stealing runtime to address locality concerns that arises with employing complex dependence structures. For a simple benchmark, we show that L3 cache misses can be reduced by employing a hierarchical work-stealing algorithm leading to a reduction in execution time. For a more complex dependence graph like cholesky decomposition, using all heuristics proposed we observe up to 22% reduction of total L2 cache misses with only a 4% increase in execution time. We argue that these optimizations would translate to energy savings that are becoming more and more important as the energy budget is increasing in importance for extreme scale systems.

REFERENCES

[1] S. Taşırlar, "Optimized event-driven runtime systems for programmability and performance," Ph.D. dissertation, Rice University, 2015.

[2] "The arm cortex-a9 processors," http://www.arm.com/files/pdf/ARMCortexA-9Processors.pdf, accessed: 2014-08-28.

[3] "Intel 64 and ia-32 architectures optimization reference manual," http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf, accessed: 2014-08-28.

[4] M. G. Ricken, "A framework for testing concurrent programs," Ph.D. dissertation, Rice University, 2007.

[5] J. Mellor-Crummey, "On-the-fly detection of data races for programs with nested fork-join parallelism," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 24–33. [Online]. Available: http://doi.acm.org/10.1145/125826.125861

[6] S. Chatterjee, S. Taşırlar, Z. Budimlic, V. Cave, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with mpi," *Parallel and Distributed Processing Symposium, International*, vol. 0, pp. 712–725, 2013.

[7] "Ocr github repository," https://github.com/01org/ocr, accessed: 2014-08-28.

[8] J. Reinders, *Intel threading building blocks*, 1st ed. Sebastopol, CA, USA: O'Reilly & Associates, Inc., 2007.

[9] M. Frigo, C. E. Leiserson, and K. H. Randall, "The implementation of the cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, ser. PLDI '98. New York, NY, USA: ACM, 1998, pp. 212–223. [Online]. Available: http://doi.acm.org/10.1145/277650.277725

[10] E. Mohr, D. A. Kranz, and R. H. Halstead, Jr, "Lazy task creation: a technique for increasing the granularity of parallel programs," in *1990 ACM Conference on LISP and Functional Programming*. New York, New York, USA: ACM Request Permissions, May 1990, pp. 185–197.

[11] J. Dinan, D. B. Larkins, P. Sadayappan, S. Krishnamoorthy, and J. Nieplocha, "Scalable work stealing," in *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, ser. SC '09. New York, NY, USA: ACM, 2009, pp. 53:1–53:11. [Online]. Available: http://doi.acm.org/10.1145/1654059.1654113

[12] D. Hendler and N. Shavit, "Non-blocking steal-half work queues," in *Proceedings of the Twenty-first Annual Symposium on Principles of Distributed Computing*, ser. PODC '02. New York, NY, USA: ACM, 2002, pp. 280–289. [Online]. Available: http://doi.acm.org/10.1145/571825.571876

[13] G. Cong, S. Kodali, S. Krishnamoorthy, D. Lea, V. Saraswat, and T. Wen, "Solving large, irregular graph problems using adaptive work-stealing," in *Parallel Processing, 2008. ICPP '08. 37th International Conference on*, Sept 2008, pp. 536–545.

[14] T. Gautier, J. Lima, N. Maillard, and B. Raffin, "Xkaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *Parallel Distributed Processing (IPDPS), 2013 IEEE 27th International Symposium on*, May 2013, pp. 1299–1308.

[15] S. Taşırlar and V. Sarkar, "Data-Driven Tasks and Their Implementation," in *2011 International Conference on Parallel Processing*. IEEE Computer Society, Sep. 2011.

[16] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," *J. ACM*, vol. 46, pp. 720–748, September 1999. [Online]. Available: http://doi.acm.org/10.1145/324133.324234

[17] J.-N. Quintin and F. Wagner, "Hierarchical work-stealing," in *Proceedings of the 16th International Euro-Par Conference on Parallel Processing: Part I*, ser. EuroPar'10. Berlin, Heidelberg: Springer-Verlag, 2010, pp. 217–229. [Online]. Available: http://dl.acm.org/citation.cfm?id=1887695.1887719