Report Number:

2016

Locality-Aware Dynamic Task Graph Scheduling

Jordyn Maglalang, Sriram Krishnamoorthy, and Kunal Agrawal

Dynamic task graph schedulers automatically balance work across processor cores by scheduling tasks among available threads while preserving dependences. In this paper, we design NabbitC, a provably efficient dynamic task graph scheduler that accounts for data locality on NUMA systems. NabbitC allows users to assign a color to each task representing the location (e.g., a processor core) that has the most efficient access to data needed during that node's execution. NabbitC then automatically adjusts the scheduling so as to preferentially execute each node at the location that matches its color—leading to better locality because the node is likely to... Read complete abstract on page 2.

Locality-Aware Dynamic Task Graph Scheduling

Jordyn Maglalang, Sriram Krishnamoorthy, and Kunal Agrawal

Complete Abstract:

Dynamic task graph schedulers automatically balance work across processor cores by scheduling tasks among available threads while preserving dependences. In this paper, we design NabbitC, a provably efficient dynamic task graph scheduler that accounts for data locality on NUMA systems. NabbitC allows users to assign a color to each task representing the location (e.g., a processor core) that has the most efficient access to data needed during that node's execution. NabbitC then automatically adjusts the scheduling so as to preferentially execute each node at the location that matches its color—leading to better locality because the node is likely to make local rather than remote accesses. At the same time, NabbitC tries to optimize load balance and not add too much overhead compared to the vanilla Nabbit scheduler that does not consider locality. We provide a theoretical analysis that shows that NabbitC does not asymptotically impact the scalability of Nabbit . We evaluated the performance of NabbitC on a suite of memory intensive benchmarks. Our experiments indicates that adding locality awareness has a considerable performance advantage compared to the vanilla Nabbit scheduler. In addition, we also compared NabbitC to OpenMP programs for both regular and irregular applications. For regular applications, OpenMP achieves perfect locality and perfect load balance statically. For these benchmarks, NabbitC has a small performance penalty compared to OpenMP due to its dynamic scheduling strategy. For irregular applications, where OpenMP can not achieve locality and load balance simultaneously, we find that NabbitC performs better. Therefore, NabbitC combines the benefits of locality- aware scheduling for regular applications (the forte of static schedulers such as those in OpenMP) and dynamically adapting to load imbalance (the forte of dynamic schedulers such as Cilk Plus, TBB, and Nabbit).

Locality-Aware Dynamic Task Graph Scheduling

Jordyn Maglalang*, Sriram Krishnamoorthy†, Kunal Agrawal*
*Washington University in St. Louis †Pacific Northwest National Laboratory
jordyn.maglalang@wustl.edu, sriram@pnnl.gov, kunal@seas.wustl.edu

***Abstract*—Dynamic task graph schedulers automatically balance work across processor cores by scheduling tasks among available threads while preserving dependences. In this paper, we design NABBITC, a provably efficient dynamic task graph scheduler that accounts for data locality on NUMA systems. NABBITC allows users to assign a color to each task representing the location (e.g., a processor core) that has the most efficient access to data needed during that node's execution. NABBITC then automatically adjusts the scheduling so as to preferentially execute each node at the location that matches its color—leading to better locality because the node is likely to make local rather than remote accesses. At the same time, NABBITC tries to optimize load balance and not add too much overhead compared to the vanilla NABBIT scheduler that does not consider locality. We provide a theoretical analysis that shows that NABBITC does not asymptotically impact the scalability of NABBIT.**

We evaluated the performance of NABBITC on a suite of memory intensive benchmarks. Our experiments indicates that adding locality awareness has a considerable performance advantage compared to the vanilla NABBIT scheduler. In addition, we also compared NABBITC to OpenMP programs for both regular and irregular applications. For regular applications, OpenMP achieves perfect locality and perfect load balance statically. For these benchmarks, NABBITC has a small performance penalty compared to OpenMP due to its dynamic scheduling strategy. For irregular applications, where OpenMP can not achieve locality and load balance simultaneously, we find that NABBITC performs better. Therefore, NABBITC combines the benefits of locality-aware scheduling for regular applications (the forte of static schedulers such as those in OpenMP) and dynamically adapting to load imbalance (the forte of dynamic schedulers such as Cilk Plus, TBB, and Nabbit).

I. INTRODUCTION

In recent years, parallel computers have become ubiquitous and many high-level programming languages and libraries, such as OpenMP, Cilk Plus, Nabbit, etc. have emerged. These languages and libraries allow programmers to express the logical parallelism in their programs while the runtime scheduler schedules the work on the available cores automatically. For multicores with few cores and uniform access to the memory hierarchy, these languages and runtime systems provide both good performance and a relatively simple programming model.

On large multicores with non-uniform memory access (NUMA), however, *locality* is an important consideration since a *remote memory access*—access to data reachable from a memory controller that is further away via the on-chip network—can cost much more than a *local memory access*. Regular applications can be structured to implicitly ensure locality between initialization and subsequent use when using static schedulers such as in OpenMP. Irregular applications, on the other hand, require dynamic load balancing which dynamic schedulers, such as those in OpenMP tasks, TBB, and Cilk Plus provide. These systems, however, have no notion of the location of data and often fail to provide good performance for regular memory-intensive applications.

Ideally, one would like to have a high-level and easy-to-use programming model which incorporates dynamic scheduling and locality. We present NABBITC, a locality-aware extension of a task graph library NABBIT. In the NABBIT programming model, the programmer expresses computations as a task graph where each node is a task and edges represent dependences between tasks. NABBIT is a library built on top of a Cilk Plus[1] and therefore, NABBIT programs are scheduled using a provably good work-stealing scheduler. This paper makes NABBIT locality aware by allowing the programmer to give locality hints to the scheduler using a simple coloring scheme. In particular, we make the following contributions.

1. We extend the interface so that the programmer can provide a *color* to each task; if a task is colored a color *c*, then the data used by this task is local to processor with color *c*. Multiple nearby cores can have the same color.
2. We modify both the NABBIT library and the Cilk Plus runtime system to allow processors to preferentially execute tasks that share the color with them. Therefore, if the user provides "correct coloring", then workers preferentially execute tasks that access local data, thereby reducing the expensive remote accesses.
3. NABBITC tries to strike a balance between improving locality and preserving the guarantees of low overhead and good load balance provided by NABBIT. We prove that NABBITC, by and large, preserves the asymptotic guarantees provided by NABBIT. In particular, for reasonable task graphs—those with enough parallelism and where tasks of all colors appear near the root of the graph—NABBITC provides nearly asymptotically optimal speedup.
4. We evaluated the performance of NABBITC on a suite of memory intensive applications and find that it succeeds in providing both good load balance and good locality. It consistently out performs vanilla NABBIT due to improvements in locality. In addition, on PageRank, an exemplar irregular benchmark, NABBITC outperforms OpenMP static and guided scheduling strategies by combining dynamic load balancing and locality awareness.

II. BACKGROUND

In this section, we describe NABBIT, a high-level task-graph scheduling library built on top of the Cilk Plus runtime system. We outline the NABBIT programming model and show how NABBIT recursively executes task graphs in parallel. We also

[1] It was originally designed to be built on top of Cilk++, but it is trivial to port to Cilk Plus. Indeed, it was designed so that it can be ported to any programming language that supports fork-join parallelism.

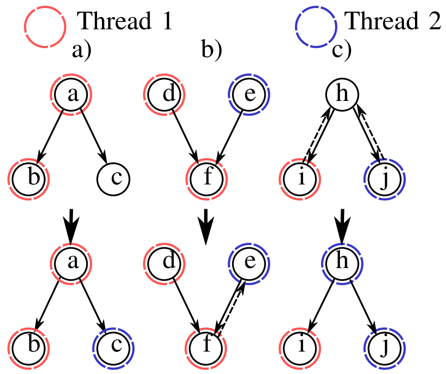

Fig. 1: NABBIT scheduling examples presented in two stages. A thread surrounding a node denotes that thread is processing that node. A solid line from a to b denotes that b is a predecessor of a while a dashed line from b to a denotes that a is in b's successor list.

provide a brief overview of the GCC Cilk Plus implementation upon which NABBIT is built.

NABBIT task-graph scheduling. NABBIT schedules task graphs through static and dynamic exploration of the task graph. A NABBIT task graph is a directed acyclic graph with a set of explicit nodes that represent tasks and edges that represent dependences between tasks. Each node u in the task graph specifies its predecessors—tasks that have edges to u and therefore must be executed before u can be processed. For this paper, we will use the terms node and task interchangeably.

We summarize the key aspects of the NABBIT dynamic task graph scheduler (more details in Agrawal et al. [1]). One interesting property of NABBIT is that it computes nodes *on demand*. The scheduler takes an input specified in the form of a sink node, whose execution completes the execution of the task graph. Upon creation, this node has a list of predecessors and no successors. The sink node together with the predecessor specification transitively identifies all vertices that need to be executed to compute the sink node. The scheduler actions:

1) To process a node, a thread initializes the node and its list of predecessors and proceeds to execute them in a recursive parallel depth-first fashion. Consider the example in Figure 1a. When thread 1 wants to process a and finds that b and c are its predecessors that have not been initialized, it goes ahead and tries to process one (b in this case). While b is being processed, another thread can steal c.
2) When processing a node's predecessor, if a thread finds that some predecessor has already been initialized by some other thread but has not finished executing, the thread adds the current node to the predecessor's successor list and moves on. In Figure1b, thread 1 is processing d while thread 2 is processing e. f is a predecessor of both d and e. Each thread will try to initialize the predecessor f but only one will succeed, in this case thread 1. Thread 2 is attaches e to f's successor list and tries to find other work to do.
3) After a node is computed, the thread checks if there are any enqueued successors and if so, determines if those successors are ready to execute (i.e., have no other predecessors on which they are waiting). In the event that a successor is ready, the thread will recursively execute that node. In Figure 1c, both nodes i and j have h enqueued in their successor lists. Thread 1 computes i and checks if h is ready to execute. Since h still depends on j, thread 1 moves on. Thread 2, after computing j checks h, sees that it is ready and proceeds to execute it.

This procedure ensures that a node is computed only after all its (transitive) predecessors have been computed, ensuring correctness. In addition until an initialized node u is computed, it is (a) either in a thread's stack, (b) in a successor list, or (c) is a predecessor of an initialized node. This guarantees that every node u will be inspected and executed eventually. Also, this ensures that the sink node, and thus whole task graph, is executed to completion.

Atomicity choices ensure the absence of data races. The predecessor and successor lists allow threads to execute without blocking/waiting for any action by another thread. The recursive parallel design allows for the implementation of the NABBIT's scheduler as a Cilk Plus program. All vertices in either the predecessor or successor lists can be executed in parallel. In addition, NABBIT ensures that no ordering constraints other than those implied by the predecessor relationships is imposed on the execution: a node u is ready to execute immediately after all its predecessors have been computed and unless every processor is busy doing other work, some processor will find and execute u. This ensures that NABBIT does not alter the task graph's critical path length, enabling the scheduler to guarantee asymptotic optimality. Essentially, if the task graph itself has a parallelism of at least P, then NABBIT guarantees that it gets $\Omega(P)$ speedup on P processors for most reasonable task graphs. In addition, since it leverages the Cilk Plus work-stealing scheduler and uses distributed processing, NABBIT provides low overheads. These properties of asymptotic optimality and low overheads are not normally achieved by other task graph schedulers, such as scheduler currently used in OpenMP's SMPSs [2], since they do not process nodes on demand.

GCC Cilk Plus We compile NABBIT using the GCC implementation of Cilk Plus, an extension to C++. Cilk Plus is a *processor oblivious* language—the programmer expresses the *logical parallelism* of the program using three keywords without any reference to how many threads must execute the program and how. The **cilk_spawn** keyword indicates that the succeeding function can execute in parallel with its continuation. The **cilk_sync** keyword is a local barrier; all previously spawned functions by current function must complete before the program execution can move this statement. Cilk Plus also provides a parallel_for keyword, which indicates that all iterations can be executes in parallel. This keyword is essentially syntactic sugar and is implemented using spawns and syncs.

The Cilk Plus runtime system uses randomized work stealing to schedule these fork-join programs on P available cores. The program executes on P worker threads, one for each core in the target machine. Each worker has a local deque of work. When a worker p executing function foo spawns a function bar, the frame corresponding to the caller foo is placed at the bottom of the p's deque and p starts executing bar. When p returns from a function, it pops the function at the bottom of its deque and continues executing. (If executing on one thread,

```
class DynamicNabbitNode:
  Key key //this node's key
  List<Key> predecessors //List of predecessors' keys
  List<DynamicNabbitNode *> successors //List of successor nodes
  virtual void init() //initialize this code (user-defined)
  virtual void compute() //compute this task representing this node (user-defined)
  int color() = color(node.key) //helper function: this node's color

int color(Key key) //user-defined function mapping a key to its color
```

Fig. 2: NABBITC abstract class interface

the program follows the normal depth-first execution followed by C or C++.) If worker q's deque is empty, it becomes a thief, picks a random victim worker, say p, steals the top frame from p's deque, and starts executing it. If a steal attempt is unsuccessful, meaning that the victim had an empty deque, then the thief continues to steal until it finds work.The Cilk Plus compiler inserts code at spawns and syncs to ensure that deques are managed correctly. In addition, when a worker's deque is empty, it makes calls into the runtime to make sure that steals occur correctly.

III. NABBITC DESIGN

In this section, we describe our extensions to NABBIT to specify colors, propagate this information through the runtime, and extend the scheduler to be take into account task colors. Throughout this section, we will present the relevant NABBIT interface and the NABBITC extensions for color-aware task graph scheduling.

NABBITC *interface*

Recall that in NABBIT, users model their computation as a task graph, where nodes are tasks to be computed and edges represent data dependencies between computations. Algorithm 2 shows the abstract class interface for defining nodes and their data dependencies. All nodes in a task graph dynamically scheduled by NABBIT inherit from DynamicNabbitNode class and implement the member functions shown in Algorithm 2. The init() and compute() functions serve to initialize node parameters and perform the computation represented by the node, respectively. Each task (node in the task graph) is associated with a unique key. The user also specifies the list of predecessors, identified by their keys, this node depends on in the predecessors array. In addition to the information on tasks and their dependencies needed by original NABBIT, NABBITC requires the user to define a color() function that returns a node's color. This function definition serves as the mechanism for the user to provide locality information to NABBITC and is the only additional piece of information the user must provide.

We now present extensions to NABBIT and the underlying Cilk Plus runtime that together constitute the NABBITC infrastructure to exploit this color information to optimize locality.

Designing a locality-guided task-graph scheduler

NABBITC attempts to achieve multiple goals during scheduling: (1) improve data locality by executing nodes of the same color as the executing processor; (2) achieve good load-balance for the computation as a whole; and (3) introduce minimal overhead into the original NABBIT scheduling pathway. Extending the NABBIT task-graph scheduling library to make use of user-provided locality information involves altering how Cilk Plus workers find and determine what to work on. We introduce two specific changes to NABBIT in order to implement this change in policy: (1) **color-aware scheduling using morphing continuations** allow workers to reorganize work so that they may preferentially execute nodes that have the same color as theirs and (2) **colored steals** allow Cilk Plus workers to find work of their color from the current set of stealable frames. In order to implement these policies, we must also change the Cilk Plus runtime system. We now describe these changes.

```
//Recursively spawn colors using morphing continuations
void spawn_colors(colors):
  if len(colors)==1:
     spawn_nodes(colors[0])
  else
    c_p = /*this worker's color*/
    /*split available colors into two halves*/
    first_half = colors[0:len(colors)/2]
    second_half = colors[len(colors)/2:]
    if c_p in second_half.keys():
      swap(first_half, second_half)
    cilkrts_set_next_colors(second_half.keys())
    cilk_spawn spawn_colors(first_half)
    spawn_colors(second_half)
    cilk_sync

//Recursively spawn nodes of the same color
void spawn_nodes(nodes):
  if len(nodes)==1:
    if nodes[0] is a successor:
      if nodes[0] is ready:
        nodes[0].compute_and_notify()
    else: /*predecessor key*/
      try_init_compute(this,nodes[0])
  else:
    color = nodes[0].color() /*all nodes have same color*/
    first_half = nodes[0:len(nodes)/2]
    second_half = nodes[len(nodes)/2:len(nodes)]
    cilkrts_set_next_colors(color)
    cilk_spawn spawn_nodes(first_half)
    spawn_nodes(second_half)
    cilk_sync
```

Fig. 3: Pseudo-code for color-aware spawning of a set of nodes in NABBITC using morphing continuations. We use a hybrid C++/Python syntax to enhance readability.

Color-aware execution order using morphing continuations. The primary source of concurrency in NABBIT is the concurrent processing of all predecessors or successors[2] of a given node. As explained in Section II, NABBIT enables this by spawning the execution of all predecessors (or successors) in parallel using a parallel for loop. NABBIT is oblivious to the order in which these nodes are processed. NABBITC, however, extracts colors from this list of nodes in order to preferentially process nodes with colors that would improve locality.

The crucial function for this purpose is the function spawn_colors shown in Algorithm 3. This function is called on a list of colors colors (and implicitly, a set of nodes which have these colors). At a high-level, when a processor with color c_p is executing this function, it tries to execute the nodes with color c_p by recursively calling spawn_colors on

[2] As described in Section II, NABBIT implicitly maintains successors array for each node u and NABBIT may push successor nodes into it when they must wait for u to complete.

```
/*Helper functions to obtain colors*/
int color(DynamicNabbitNode node):
  return node.color()
int color(DynamicNabbitNode *node):
 return node−>color()

/*Gather list of spawns based on their color.
  T = Key (for predecessor list) or
  T = DynamicNabbitNode (for successor list)*/
auto gather_colors(T nodes):
  //group nodes based on their colors
  Map<int,List<T>> colors
  for n in nodes:
    colors[color(n)].add(n)
  return colors

/*Initialize this (already created) node and compute*/
void init_node_and_compute():
  this.init()
  colors = gather_colors(this.predecessors)
  spawn_colors(colors)
  if all this.predecessors have been computed:
    this.compute_and_notify()

/*Try to initialize node's predecessor with key pkey */
void try_init_compute(node, pkey):
  //atomically attempt to create a predecessor with key pkey
  pred = /*reference to node for key pkey*/
  if /*creation succeeded*/:
    pred.init_node_and_compute()
  else: /*already created by this or some other thread*/
    atomic pred.successors.add(node) //enqueue

/*compute a node and notify its successors*/
void compute_and_notify():
  this.compute()
  while /*there are new successors in this.successors*/:
    colors = gather_colors(this.successors)
    spawn_colors(colors)
```

Fig. 4: Key routines to spawn predecessors and successors in NABBITC.

the half of the list that contains c_p. Once it reaches the base case (the set colors contains only one color), it then spawns all the nodes of color c_p using the function spawn_nodes. This function spawn_nodes is essentially a parallel-for loop over the nodes of this color.

The function spawn_colors re-organizes the order in which nodes are spawned so that the nodes of the preferred color c_p are spawned first, implementing what we call a *morphing continuation*. The particular strand that is spawned and the continuation of the strand depends on the color of the processor which is doing the spawn. Another important thing to note about this code is that if the preferred color c_p is not present in the list, the function will spawn the nodes in the original ordering of the list — therefore, a worker does not stall even if it can not find the work of its color.

The function spawn_colors is called in three places in the NABBITC library. Algorithm 4 shows the actions to initialize and execute a node. init_and_compute() acquires the colors of the current node's predecessors and invokes spawn_colors() if there exist more than one. Similarly, when spawning the list of successors, compute_and_notify() collects the set of colors for the list of successors and invokes spawn_colors() if there are more than one. Finally, spawn_colors is a recursive function which is also called by itself.

This morphing continuation design allows us to use the same mechanism in two scenarios. First, when a processor spawns the predecessors (or successors) if the node it is currently working on, it uses spawn_nodes to preferentially execute the predecessor(s) (or successor(s)) of its own color. Second, and the more subtle point, is as follows. Note that in Cilk Plus, when a thief worker steals a task after the spawn of a function[3], it executes the function's continuation. Since spawn_colors is recursive, when a worker steals a continuation, the first statement is executes is spawn_colors. Therefore, the thief also preferentially executes the nodes of *its* color using the same mechanism.

Colored Steals: When a worker has no assigned work, either because it has run out of local work or is at the start of execution, we want that worker to acquire work of its preferred color if possible. In order to do so, we change the stealing policy of Cilk Plus to allow *colored steal* where a worker checks a deque and only steals the work (continuation) at the top of the deque if that continuation contains some node of this worker's preferred color. We will describe the implementation below — we first describe our policy details about when we do colored steals vs. random steals.

One of the goals of NABBITC is to strike a reasonable balance between locality — workers preferentially execute work of their color — and load balance — workers are not idle for too long. In order to do so, we make two changes to the standard Cilk Plus policy of random steals. First, when a worker p with color c_p runs out of work, it does a constant number of colored steal attempts before attempting a random steal. That is, it randomly picks a victim worker q and checks if the frame on the top of q's deque has any tasks of color c_p — if so, it steals this frame making this a successful colored steal. If not, it tries again. If it fails on a constant number of colored steal attempt, it makes a random attempt where it steals whatever is on the top of the victim worker's deque regardless of whether it has a task of color c_p or not. This policy makes sure that p tries to find work of its own color, but then also maintains provable load balance guarantees (as shown in Section IV) by greedily doing any work available if it can not easily find work of its color.

There is an exception to this policy, however, at the beginning of the computation. At the beginning of the computation, one worker starts out with executing the root node and all other workers are stealing. At this time, if a worker begins execution in a region of a task graph with no tasks of its preferred color, it will continue executing the available non-preferred tasks until all work is exhausted (as explained in the morphing continuations section). In addition, often, the first steal represents a significant amount of work (conceptually corresponding to nodes higher up in the task graph or computation tree) and a random first steal can potential to lead poor locality. Therefore, we enforce that the first steal a worker performs is a successful colored steal. After the first steal, the worker follows the policy explained above. This enforcement does affect Cilk's time bound, which we explore in Section IV. In our experiments, we found that if all colors are available at the root of the task graph, this time to first work (successful steal) is agnostic to the application, is strictly determined by

[3] A task is represented at runtime by the task's stack or activation frame

```
void cilkrts_set_worker_color(int color)
void cilkrts_set_next_colors(List<int> colors)
```

Fig. 5: Extensions to the Cilk Plus RTS API to inform the runtime of the worker's preferred color and the colors available in a continuation.

the number of processors, and, in general, has a small impact on the overall execution time.

We now describe the changes made to both NABBIT and Cilk Plus in order to implement the colored steal policy.

Color-aware GCC Cilk Plus runtime

We make the GCC Cilk Plus runtime color aware by making the following changes. First, we add two additional functions to the Cilk Plus API, shown in Algorithm 5, that allows NABBITC to provide color information to the runtime system. The first function is straight forward and is simply used by each worker to set the color of this worker. We pin worker threads and assign them a unique color based on their thread id. The second one is used to implement colored steals and requires more explanation. Recall that in order to do colored steals, a thief worker must be able to tell which color nodes are available in the frame that is on the top of victim worker's deque. This API allows NABBITC to pass this information to Cilk Plus runtime. In particular, before every **cilk_spawn**, NABBITC calls cilkrts_set_next_colors() to inform the Cilk Plus runtime about which colors are available in the continuation.

The Cilk Plus runtime is also changed with respect to what it does on spawns. At each spawn, the vanilla Cilk Plus pushes the frame of the currently executing function into a worker's deque — allowing some other worker to steal the continuation of the spawn. To enable colored steals, we maintain a color deque alongside the work deque to hold the colors available in each continuation. When NABBITC calls cilkrts_set_next_colors with a set of colors before the spawn statement, this set of colors is pushed at the bottom of the color deque — therefore, each continuation on the work deque has a corresponding set of colors on the color deque.

Now it is easy to see how one can implement colored steals. When a worker p wants to do a colored steal, it simply checks to see of color c_p (p's preferred color) is in the set of colors on the top of victim's color deque. If so, it pops the top of both the color deque and the work deque and puts them on the top of its corresponding deques making it a successful colored steal. Since the number of colors is determined by the number of workers, we make each entry in the colored deque a fixed length array of boolean flags indicating colors contained in the corresponding continuation. This makes the thief's check a constant time operation.

Setting continuation colors in NABBITC: As mentioned above, NABBITC must set colors of continuation at each spawn using cilkrts_set_next_colors function. This is done on Lines 12, 29, etc within the code in Figure 3. Note that this fits in seamlessly with the design of morphing continuations. At each spawn, we know exactly which colors are available within the spawn and which are available within the continuation. Therefore, NABBITC can easily notify Cilk Plus of the colors available in the continuation by simply telling it which colors are available in the second call to spawn_colors.

Optimizing locality through coloring

NABBITC requires that the user intentionally distribute data across their system and provides a coloring that captures computation locality. We rely on the user knowing how best to distribute data (but not partitioning work among threads), although in many cases an even distribution is sufficient. The coloring the user provides to NABBITC is intended to capture the locality of work performed, based on their data initialization. For this we make two assumptions about color: (1) that data initialized by each individual worker thread is given a unique color and (2) that each node of the computation task-graph is assigned a single color. Requiring the user to describe each node with a single color can lead to some information loss about a node's locality. For example, a node (corresponding to a task) can require data from multiple colored regions and a single color cannot comprehensively describe the node's locality. In these scenarios, the user specifies the node's color to be the one that maximizes locality for that node.

IV. ANALYSIS OF COMPLETION TIME

We now present a simple analysis showing that the modifications made to NABBIT do not negatively effect the asymptotic runtime—this implies that NABBITC also provides almost asymptotically optimal load balancing for programs that have enough parallelism.

Just as in the NABBIT paper [1], say, we are given a task graph $G = (V, E)$, where each node u has work $W(u)$. Also, say that s is the unique node with zero in-degree and t is the unique node with zero out-degree. If these nodes are not unique, we can trivially add dummy root and final nodes. Define M as the number of nodes on the longest path in V from the source s to the sink t.

We can define the work T_1 as the time it takes to execute the task graph on a single processor and span T_∞ as the time it takes to execute it on an infinite number of processors. Therefore, the work is $T_1 = \sum_{u \in V} W(u) + O(|E|)$. The second term is due to the fact that each edge needs to be checked to make sure that it is satisfied. Similarly, we the span is $T_\infty = \max_{p \in paths(s,t)} \{\sum_{u \in p} W(u) + O(M)\}$ since nodes along any path through V can not execute in parallel. By the work and span laws [3, p. 780], the completion time on P processors for a task graph is at least $\max\{T_1/P, T_\infty\}$.

We will prove the following theorem—the analysis is a small extension to the analysis of runtime for NABBIT.

Theorem 1. *For task graph $G = (V, E)$ with maximum degree d, NABBITC executes G in time $O(T_1/P + T_\infty + M \lg d + \lg(P/\epsilon) + C)$ time on P processors with probability at least $1 - \epsilon$ where C is the amount of time each worker spends at startup trying to find a node of its own color.*

This theorem is similar to the theorem proved for NABBIT [1] apart from the last term C. The main difference between NABBIT and NABBITC is the fact of colored steals. In particular, when a worker runs out of work in NABBIT, it performs a random steal. On the other hand, when a worker runs out of work in NABBITC, it first checks a constant number of deques to see if it can find work of its own color and only performs a random steal if all these checks fail. In addition, at the start of the computation, NABBITC forces a colored steal and each processor may make C checks to find a node of its own color where C may not be a constant.

Lemma 1. *The total number of colored steals performed by* NABBITC *is* $O(W+S+PC)$ *where* S *is the number of random steal attempts,* W *is the number of steps the processors spend working on computation nodes, and* C *is the number of checks each processor performs at the beginning of the computation. Consequently, the total number of colored steals is bounded by* $O(T_1 + PT_\infty + PM \lg d + P \lg(P/\epsilon) + PC)$

Proof: Trivially, the number of checks at the beginning of the computation is PC since each processor performs at most C of them. After this, after a constant number of checks, a processor has either found work (therefore, these checks are bounded by $O(W)$) or the processor performs a random steal (these checks are bounded by $O(S)$). Summing these up gives us the result. The NABBIT analysis proves that the total number of work steps in the computation is at most T_1 and the total number of steal steps is at most $O(PT_\infty + PM \lg d + P \lg(P/\epsilon))$. This gives us the desired bound. ■

At any step, a worker is either working, doing a random steal, or doing a colored steal. Therefore, the total number of processor steps is bounded by $O(T_1 + PT_\infty + PM \lg d + P \lg(P/\epsilon) + PC)$ Since there are a total of P workers, we can divide by P to get the desired running time.

V. EXPERIMENTAL EVALUATION

In this section, we evaluate NABBITC by comparing it's performance against original NABBIT and OPENMP. In particular, we try to answer the following questions:

- How well does NABBITC address locality deficiencies in NABBIT? We answer this question using benchmarks in which locality-optimized and load balanced schedules can be created using static scheduling of OPENMP and find that NABBITC provides much better performance than NABBIT and performance comparable to OPENMP.
- How well can NABBITC improve data locality while preserving the dynamic load balancing benefits from NABBIT? We answer this question using the PageRank benchmark, which cannot be easily statically scheduled, using different data sets. In this case, NABBITC really shines and performs better than both OPENMP and NABBIT.
- To what extent does NABBITC improve data locality? We find that NABBIT has significantly fewer remote accesses compared to NABBIT.
- Does the use of colored steals increase the overall cost to find work as compared to random stealing? We find that while the cost of enforcing the first colored steal is significant, NABBITC makes up for this overhead by having fewer steal attempts later.
- What is the impact of the choice of colors by the user? We consider the behavior of NABBITC using two particularly bad color choices and compare its behavior with NABBIT.

In general, NABBITC shines on benchmarks with irregular memory access patterns, remains competitive with OPENMP when memory accesses are more regular, and almost always outperforms original NABBIT. We observe that our modifications to NABBIT and Cilk Plus introduce minimal overheads, affording performance gains due to a reduction in remote memory accesses when a good coloring is provided.

Experimental Setup: All our experiments were performed on an 80-core NUMA machine with 8 Intel Xeon E7-8860 2.27GHz 10-core processors and 1TB of collective DRAM. The machine uses Red Hat Linux 4.4.7-9 configured with 4KB pages. We use a stable GCC 4.9.0 build from the gcc-cilkplus branch for compiling our OPENMP and NABBIT benchmarks and extend this build for NABBITC.

Benchmarks and Baselines: We will compare NABBITC performance to NABBIT and OPENMP. OPENMP offers multiple scheduling strategies for parallel for loops. The OPENMPSTATIC policy simply divides up the iteration space evenly among workers while OPENMPGUIDED dynamically load balances using adaptive block sizes.

Table I details the benchmarks and input configurations used. We selected various memory-bound applications to demonstrate the importance of achieving good locality when scheduling task-graph computations. The first five benchmarks exhibit regular memory access patterns. We consider these benchmarks to demonstrate the limitations of a dynamic task graph scheduler such as NABBIT that does not account for locality, and evaluate the potential for NABBITC to address these limitations. For these benchmarks, OPENMPSTATIC performs very well if we match the initialization and computation loops; as explained later, this strategy provides optimal locality to regular applications even without locality hints. Therefore, we only compare against this OPENMP strategy since it always performs better than OPENMPGUIDED.

PageRank iteratively computes the PageRank using the power method [7]. This benchmark exhibits access patterns dependent on the graph structure, with varying amounts of work per vertex. We consider three data sets from web crawls [4] that vary in size and graph structure. Specifically, twitter-2010 shows wider variation in its connectivity (e.g., much larger maximum out-degree) than the other data sets considered. On this benchmark, we compare against both OPENMPSTATIC and OPENMPGUIDED strategies for this benchmark.

The Smith-Waterman dynamic program [8] benchmarks exhibit highly regular memory access patterns. We have implemented the wavefront computation in OPENMP, which must synchronize at each diagonal step. In NABBIT and NABBITC, we model the entire computation as a task-graph, exposing more parallelism.

Coloring strategy: In all benchmarks, we used OPENMP to distribute data evenly across the machine, with each processor core initializing a unique region of the data. Each thread is pinned to a processor core and given a unique color. During initialization, each data region is colored based on the color of the thread that initializes it. For regular benchmarks, we group the data accessed by each node based on their color, and pick the color corresponding to the largest fraction of data as the node's color. This color function, provided by the user, can be implemented efficiently for regular benchmarks. Computing the largest color is expensive for irregular benchmarks such as PageRank, where the accesses are data-dependent and involve

Benchmark	Description	Problem size	Iterations	Task graph nodes	Serial OpenMPstatic time (seconds)
cg	NAS conjugate gradient	$NA = 900000$, $NNZ = 26$	1	300	309
mg	NAS multigrid	$n\{x, y, z\} = 2048$, $LM = 11$	1	16384	690
heat	Heat diffusion stencil	$n = 16384$, $m = 655360$	5	102400	377
fdtd	Finite difference time domain	$n = 16384$, $m = 655360$	5	102400	970
life	Conway's game of life	$n = 16384$, $m = 655360$	5	102400	275
page-uk-2002	PageRank (power method) uk-2002 dataset	$nv = 18\text{M}$, $ne = 298\text{M}$	10	1800	198
page-twitter-2010	PageRank (power method) twitter-2010 dataset	$nv = 41\text{M}$, $ne = 1468\text{M}$	10	4100	1025
page-uk-2007-05	PageRank (power method) uk-2007-05 dataset	$nv = 105\text{M}$, $ne = 3738\text{M}$	10	10500	900
sw	Smith-Waterman (n^3)	$\{n, m\} = 5120$, $B = 32x32$	1	25600	450
swn2	Smith-Waterman (n^2)	$\{n, m\} = 131072$, $B = 1024x1024$	1	16384	179

TABLE I: Benchmark configurations and serial OpenMPstatic execution time. The PageRank benchmarks use the same code with three different web crawl datasets [4], [5], [6].

a large number of irregular accesses. In PageRank, each task takes a block of pages as input, which are accessed regularly, and updates the ranks of pages linked to them, which are accessed in an irregular fashion. The irregular accesses while traversing the links are not avoidable. Therefore, we color each task based on the block of pages it takes as input.

A. Overall performance

We now demonstrate the effect of locality-guided scheduling on the overall performance. In Figure 6, we present the speedup achieved by OpenMP, Nabbit, and NabbitC over serial execution. Error bars show standard deviation across five runs. In general, NabbitC outperforms Nabbit when the problem is sufficiently large. NabbitC shines best with larger irregular PageRank benchmarks, where the impact of locality is more prominent, while remaining competitive with OpenMP on the stencils and NAS benchmarks and outperforming OpenMP for the Smith-Waterman dynamic programs.

We see that in cg, when there are very few nodes in the task graph, NabbitC's benefit over original Nabbit becomes negligible because processor cores have few nodes to work with. With mg, heat, fdtd, and life, when there are many nodes in the task graph, NabbitC is able to continue getting good performance while original Nabbit suffers due to its locality-obliviousness. For these benchmarks, we see that OpenMP consistently performs best. When threads are pinned and the computation loops are scheduled in the same way as the data initialization loops, OpenMP achieves the maximum locality possible despite not having received any explicit locality hints from the programmer. In addition, it also achieves good load balance, since each iteration does approximately equal amount of work. For these benchmarks, NabbitC's performance approaches that of OpenMP, whereas Nabbit's scalability suffers with increase in core count. For PageRank, OpenMP is not able to maintain its consistency in performance because it is no longer able to achieve locality and load balance simultaneously due to the irregular nature of this application. We see that for larger problems (indicated by the problem size and the larger serial execution time), NabbitC scales better than original Nabbit, OpenMPstatic, or OpenMPguided. For SmithWaterman we see that with the unavoidable remote accesses inherent in the algorithms, NabbitC and Nabbit perform comparably. Both, however, are able to exploit more parallelism than the wavefront OpenMP implementation and edge out ahead.

B. Locality impact of NabbitC's scheduling strategy

We now look closer at the locality achieved by NabbitC during the execution of these benchmarks. Because counting each memory reference might be expensive[4], we perform this check at the node level in the task graph. This consists of two parts. Note that each of our evaluation system consists of eight NUMA domains, each with 10 cores. First, for each thread, we count the number of nodes it executes that are not the same color as any thread in the same NUMA node. Second, for each thread, we check all predecessors of executed nodes, and count those that are not the same color as any thread in the same NUMA node. Sum of these counts across all threads is reported as the number of remote accesses. For the regular benchmarks, we can compute this as the benchmarks execute without perturbing the execution. For PageRank, this instrumentation can significantly perturb the execution time. Therefore, we track the nodes executed by each thread to record the schedule used in the timing runs. This schedule is replayed to compute the percentage of remote accesses.

Figure 7 shows the percentage of accesses that are remote for Nabbit, NabbitC, and OpenMPstatic, on 20 or more processor cores (smaller core counts fit in one NUMA domain and do not incur remote accesses). Because Nabbit relies on the random steals in Cilk Plus to disseminate work, the percentage of remote accesses increases with scale, ranging from 45% to 88%, exhibiting a consistent trend across all benchmarks. The introduction of colored steals significantly decreases the percentage of remote accesses. For all benchmarks except twitter-2010 and the Smith-Waterman benchmarks, NabbitC incurs 0% to 9% remote accesses. Importantly, unlike in the case of Nabbit, this percentage does not strictly increase with scale for the regular benchmarks. All strategies incur a high percentage of remote accesses for twitter-2010 and Smith-Waterman.

For regular applications, OpenMPstatic incurs almost no remote accesses, as we expect from how the data is initialized. For PageRank, OpenMPstatic still has fewer remote accesses than NabbitC; however, as we saw above, it does not have good performance since it is unable to provide good load balance. This result indicates the importance of both locality and load balance—while Nabbit provides great load

[4]We were limited by OS version and available hardware counters and were unable to measure remote accesses, stall cycles, etc.

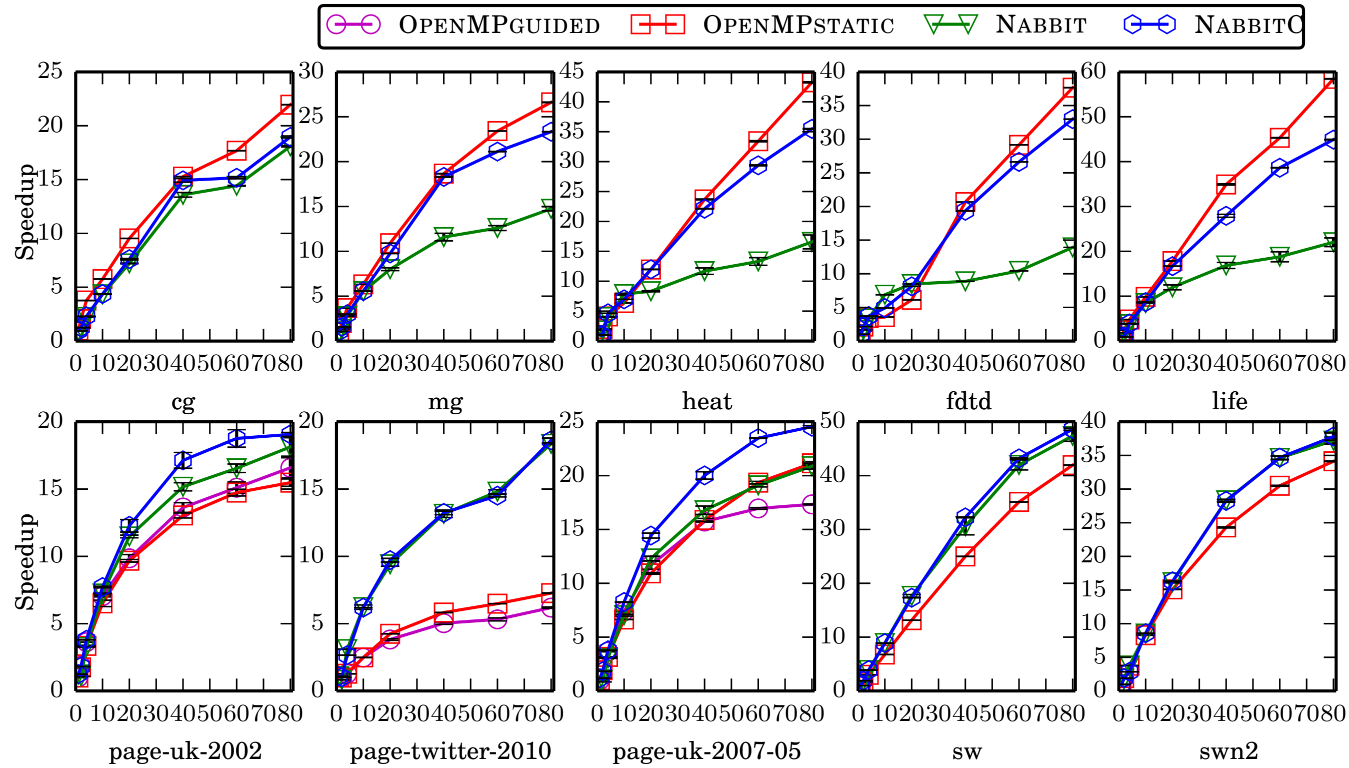

Fig. 6: Speedup for all benchmarks. x-axis: number of threads (processor cores); y-axis: speedup over serial OpenMPStatic.

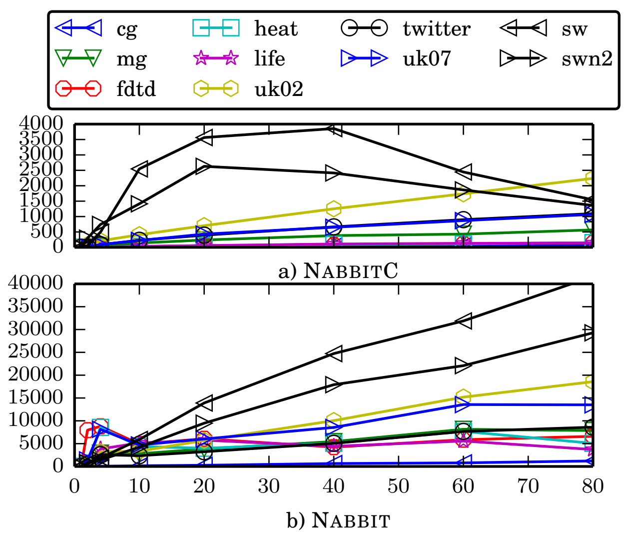

Fig. 8: Average number of successful steals for (a) NabbitC and (b) Nabbit. x-axis: number of cores; y-axis: Average number of successful steals

balance and OpenMPStatic provides great locality, NabbitC performs better than both on this irregular benchmark since it simultaneously considers both metrics.

C. Overheads due to colored steals

The two sources of overhead for NabbitC arise from requiring a constant number of colored steals before performing random steals and forcing the first steal to be a colored steal.

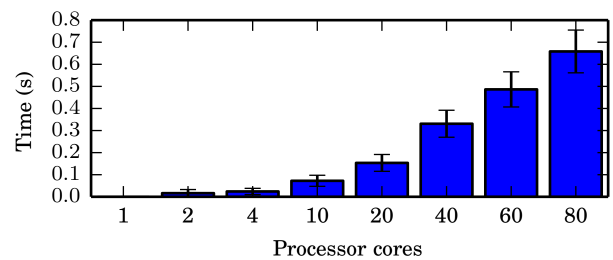

Fig. 9: Average idle time per processor core (across all processor cores and runs) due to forcing the first colored steal for the heat benchmark. Error bars show standard deviation across five runs among all processor cores. We observed this time was the same for all benchmarks.

Effect on total steals: We now look at the comparison of NabbitC and Nabbit at a more fine-grained level. In Figure 8 we see that NabbitC, perhaps counter-intuitively, performs far fewer total successful steals than Nabbit. The introduction of colored steals, and specifically enforcing the first colored steal, helps to significantly reduce the total number steals by ensuring that thieves acquire nodes higher up in the task graph to start with. Due to the depth-first nature of the scheduler, nodes higher up in the task graph have more potential work. Therefore, by ensuring thieves begin with nodes connected to the root, NabbitC is able to effectively increase the amount of work each worker begins with, reducing the total number of steals required.

Overhead due to enforcing first colored steal: To calculate the overhead of ensuring that the first steal is a colored steal, Fig. 9 shows the average amount of time processor cores

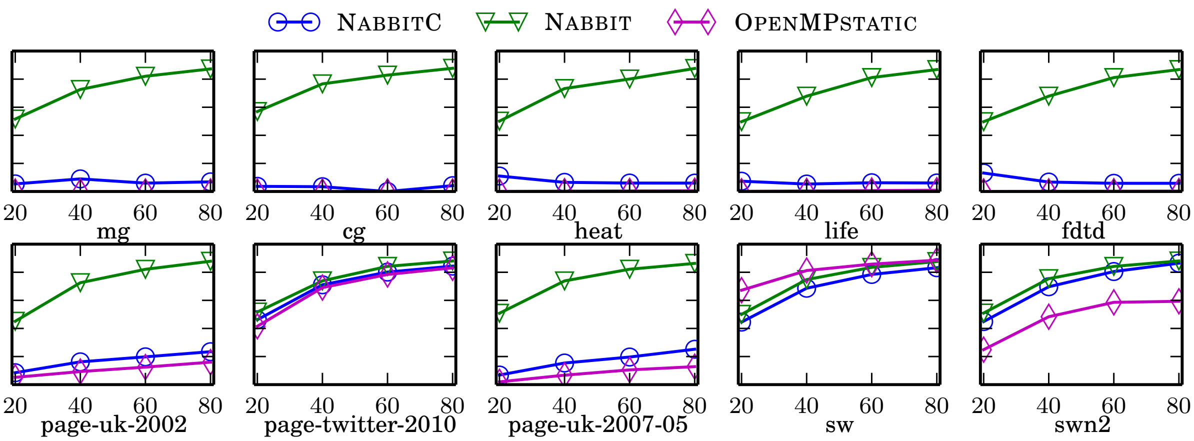

Fig. 7: Percentage of accesses that are to data in remote NUMA domains. We show percentages for 20–80 cores (1–10 cores fit in one NUMA domain and do not incur remote accesses). x-axis: core count; y-axis: Percentage of accesses that are remote.

P	cg	mg	heat	fdtd	life	uk-02	twitter	uk-07	sw	swn2
20	0.96	0.76	0.66	1.08	0.65	0.75	0.83	0.76	0.76	0.66
40	0.97	0.88	0.78	1.10	0.77	0.83	0.55	0.94	0.88	0.78
60	0.96	0.97	0.95	1.08	0.94	0.89	0.49	1.02	0.97	0.95
80	1.02	0.96	0.93	0.94	0.97	0.91	0.41	1.02	0.96	0.93
	0.98	0.89	0.83	1.05	0.83	0.85	0.57	0.93	0.89	0.83
	0.02	0.08	0.12	0.06	0.13	0.06	0.16	0.11	0.08	0.12

TABLE II: Speedup of NabbitC over Nabbit when all tasks are assigned bad colors resulting in preferential execution of non-local tasks. S.D. denotes standard deviation.

P	cg	mg	heat	fdtd	life	uk02	twitter	uk07	sw	swn2
20	1.03	0.99	0.94	1.06	0.93	1.03	1.12	1.12	0.99	0.94
40	1.02	0.99	0.99	1.04	0.92	0.97	1.09	1.10	0.99	0.99
60	0.99	0.98	0.94	1.03	0.92	0.98	1.01	1.08	0.98	0.94
80	1.06	0.97	0.88	0.91	0.98	0.94	1.07	1.07	0.97	0.88
	1.03	0.99	0.94	1.01	0.94	0.98	1.07	1.09	0.99	0.94
	0.02	0.01	0.04	0.06	0.03	0.03	0.04	0.02	0.01	0.04

TABLE III: Speedup of NabbitC over Nabbit when all tasks are assigned invalid colors resulting in failure of all colored steal attempts. S.D. denotes standard deviation.

spent waiting to acquire work for the heat benchmark. We observed that the times were very similar for all other benchmarks and do not present them here due to space limitations. While this overhead can be substantial, it is agnostic to the application, provided there is at least one node from each color connected to the root. This startup cost can be amortized out with larger, longer running benchmarks. Additionally, recall that we observed that, in practice, enforcing the first colored steal results in far fewer total number of steals which makes up for this overhead.

D. Importance of good coloring

Overheads with invalid coloring: To evaluate this worst case overhead from attempted colored steals, we assigned all nodes an invalid color (no worker has this color), ensuring that all colored steals fail. Therefore, this version of NabbitC behaves like original Nabbit apart from incurring the overheads of colored steals. In Table. III, we see that NabbitC with this alternative coloring performs comparable to original Nabbit indicating that the additional work performed by colored steals introduces minimal overhead. Specifically, we observe that the mean speedups are within one or two standard deviations, indicating that, for the benchmarks considered, colored steals incur no statistically significant overhead.

Behavior under bad coloring: The performance of NabbitC is directly tied to the coloring provided by the user. NabbitC assumes the user has constructed a "good" coloring and makes decisions based on this assumption. In the event that the user has provided a "bad" coloring, NabbitC can perform as badly, or worse, than original Nabbit. To test this, we create a coloring where all nodes are given valid incorrect colors. Therefore, in this implementation, all workers will preferentially do non-local work. In Table. II, we see that NabbitC with a bad coloring loses all the performance benefits achieved due to coloring and performs similar to Nabbit. Interestingly, we observe that the mean speedups are within two standard deviations, indicating that Nabbit's locality behavior under random stealing is statistically no better than that of NabbitC under an intentionally bad coloring??.

VI. Related Work

Static task graph schedulers [9], [10], [11] minimize completion time while maximizing locality [12] by completing expanding and analyzing a task graph, together with accurate information on computation and communication costs associated with each task. We consider task graphs that are dynamically explored and do not require prior knowledge of task and communication times.

Cilk's random work stealing is agnostic of locality considerations [13]. Several efforts have incorporated locality considerations by altering the work stealing strategy [14], [15], [16], [17]. These approaches do not naturally extend to scheduling data-flow graphs while preserving provably efficiency in terms of scheduling overheads and effectiveness of load balancing.

Event-driven scheduling strategies map tasks to locality domains together with efficient identification and tracking of ready tasks that can be scheduled [18], [19]. In these systems, data distribution implies a computation partitioning with no further migration of tasks to tackle load imbalance.

SuperMatrix [20], a runtime scheduling system for algorithms operating on blocks as observed in linear algebra programs, mimics a superscalar microarchitecture's scheduling strategy in software. StarPU [21] is a task-graph scheduler for heterogeneous multi-core systems. Neither approach accounts for data locality. Dague [22], a distributed DAG engine, improves locality by working on the local queue when possible. XKaapi [23] is a work-stealing-based scheduler for task graphs that pushes tasks to processors that have better locality for those tasks. It does not preserve the critical path length or provide provable parallel efficiency.

SMPSs [2] schedules dependent tasks together to improve locality. Legion [24] exploits user-specified locality information and coherence properties to perform locality-aware scheduling using a software out-of-order processor. CnC [25] allows the specification of task graphs that are scheduling using a variety of strategies. Legion and CnC also allow user-specification to control task mapping and scheduling (using mappers in Legion and tuners in CnC). Olivier et al. developed various strategies to schedule OpenMP tasks including hierarchical scheduling, and work stealing by one thread on behalf of others in the same chip [26]. None of these schedulers in these systems attempt to preserve optimality guarantees. However, the scheduling strategy developed in this paper can be used to develop provably efficient and locality-aware scheduling algorithms for these task-graph frameworks.

Bugnion et al. [27] developed compiler-directed page coloring techniques to minimize conflict misses. Chilimbi and Shaham [28] identified *hot* data streams and then colocated them to improve spatial locality. Chen at al. studied scheduling threads for constructive cache sharing [29]. Various approaches have studied the partitioning of shared caches among threads (e.g., [30], [31], [32]). These approaches cannot be applied to optimize NUMA locality considered in this paper.

VII. Conclusions

In this paper, we have presented NabbitC, a flexible and easy-to-use task graph library that allows the user to provide locality hints via the use of coloring and provides good load balance via dynamic scheduling. NabbitC is geared towards scheduling on NUMA hardware, where remote accesses may be considerably more expensive than local accesses, but one must strike a balance between locality and load balance to get good performance. Experimental results indicate that this approach is promising, especially for memory intensive irregular applications running on NUMA machines, where static scheduling can compromise load balancing and locality-unaware dynamic scheduling has too many remote accesses. While NabbitC uses Cilk Plus as the underlying language and runtime, we believe this approach can be implemented on other systems such as Intel's Threading Building Blocks.

References

[1] K. Agrawal, C. E. Leiserson, and J. Sukha, "Executing task graphs using work-stealing," in *IPDPS*, 2010, pp. 1–12.

[2] J. M. Perez, R. M. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *CLUSTER*, 2008, pp. 142–151.

[3] T. Cormen, C. Leiserson, R. Rivest, and C. Stein, *Introduction to Algorithms*. MIT Press, 2009.

[4] "Laboratory for web algorithmics," http://law.di.unimi.it/datasets.php.

[5] P. Boldi and S. Vigna, "The WebGraph framework I: Compression techniques," in *WWW*, 2004, pp. 595–601.

[6] P. Boldi, M. Rosa, M. Santini, and S. Vigna, "Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks," in *WWW*, 2011, pp. 587–596.

[7] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: bringing order to the web," 1999.

[8] T. F. Smith and M. S. Waterman, "Identification of common molecular subsequences," *Journal of molecular biology*, vol. 147, no. 1, pp. 195–197, 1981.

[9] Y. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *ACM Comput. Surv.*, vol. 31, no. 4, pp. 406–471, Dec. 1999.

[10] G. Liu, K.-L. Poh, and M. Xie, "Iterative list scheduling for heterogeneous computing," *JPDC*, vol. 65, no. 5, pp. 654–665, 2005.

[11] T. Yang and A. Gerasoulis, "PYRROS: static task scheduling and code generation for message passing multiprocessors," in *SC*, 1992, pp. 428–437.

[12] N. Vydyanathan, S. Krishnamoorthy, G. M. Sabin, Ü. V. Çatalyürek, T. M. Kurç, P. Sadayappan, and J. H. Saltz, "An integrated approach to locality-conscious processor allocation and scheduling of mixed-parallel applications," *TPDS*, vol. 20, no. 8, pp. 1158–1172, 2009.

[13] M. Frigo, C. Leiserson, and K. Randall, "The implementation of the Cilk-5 multithreaded language," in *PLDI*, 1998, pp. 212–223.

[14] Y. Guo, J. Zhao, V. Cave, and V. Sarkar, "SLAW: a scalable locality-aware adaptive work-stealing scheduler for multi-core systems," in *IPDPS*, 2010, pp. 341–342.

[15] S.-J. Min, C. Iancu, and K. Yelick, "Hierarchical work stealing on manycore clusters," in *PGAS*, 2011.

[16] J.-N. Quintin and F. Wagner, "Hierarchical work-stealing," in *Euro-Par 2010-Parallel Processing*, 2010, pp. 217–229.

[17] J. Lifflander, S. Krishnamoorthy, and L. V. Kale, "Optimizing data locality for fork/join programs using constrained work stealing," in *SC*, 2014, pp. 857–868.

[18] S. Chatterjee, S. Tasirlar, Z. Budimlic, V. Cavé, M. Chabbi, M. Grossman, V. Sarkar, and Y. Yan, "Integrating asynchronous task parallelism with MPI," in *IPDPS*, 2013, pp. 712–725.

[19] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, T. Hérault, and J. J. Dongarra, "PaRSEC: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.

[20] E. Chan, E. S. Quintana-Orti, G. Quintana-Orti, and R. Van De Geijn, "SuperMatrix out-of-order scheduling of matrix operations for smp and multi-core architectures," in *SPAA*, 2007, pp. 116–125.

[21] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *CCPE*, vol. 23, no. 2, pp. 187–198, 2011.

[22] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "Dague: A generic distributed dag engine for high performance computing," *ParCo*, vol. 38, no. 1, pp. 37–51, 2012.

[23] T. Gautier, J. V. Lima, N. Maillard, and B. Raffin, "XKaapi: A runtime system for data-flow task programming on heterogeneous architectures," in *IPDPS*, 2013, pp. 1299–1308.

[24] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken, "Legion: expressing locality and independence with logical regions," in *SC*, 2012, p. 66.

[25] Z. Budimlić, M. Burke, V. Cavé, K. Knobe *et al.*, "Concurrent collections," *Scientific Programming*, vol. 18, no. 3-4, pp. 203–217, 2010.

[26] S. L. Olivier, A. K. Porterfield, K. B. Wheeler, and J. F. Prins, "Scheduling task parallelism on multi-socket multicore systems," in *ROSS*, 2011, pp. 49–56.

[27] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam, "Compiler-directed page coloring for multiprocessors," in *ACM SIGPLAN Notices*, vol. 31, no. 9. ACM, 1996, pp. 244–255.

[28] T. M. Chilimbi and R. Shaham, "Cache-conscious coallocation of hot data streams," in *PLDI*, 2006, pp. 252–262.

[29] S. Chen, P. B. Gibbons, M. Kozuch, V. Liaskovitis, A. Ailamaki, G. E. Blelloch *et al.*, "Scheduling threads for constructive cache sharing on cmps," in *SPAA*, 2007, pp. 105–115.

[30] M. K. Qureshi and Y. N. Patt, "Utility-based cache partitioning: A low-overhead, high-performance, runtime mechanism to partition shared caches," in *MICRO*, 2006, pp. 423–432.

[31] G. Suh, L. Rudolph, and S. Devadas, "Dynamic partitioning of shared cache memory," *Journal of Supercomp.*, vol. 28, no. 1, pp. 7–26, 2004.

[32] F. Mueller, "Compiler support for software-based cache partitioning," in *ACM Sigplan Notices*, vol. 30, no. 11. ACM, 1995, pp. 125–133.