Efficient Primitives for Creating and Scheduling Parallel Computations

Umut A. Acar Arthur Charguéraud Mike Rainey
Max-Planck Institute for Software Systems
{umut,charguer,mrainey}@mpi-sws.org

Abstract

We give a brief overview of our ongoing work on developing efficient and expressive abstractions for programming multicore machines. We propose a programming interface for expressing a parallel computation dynamically, as a directed acyclic graph (DAG). The DAG consists of tasks and dependencies between them. Because our interface lets the DAG take shape as the computation unfolds, the programmer can describe a variety of computations, including those expressible with existing parallel-computing paradigms, such as fork-join, spawn-sync, and parallel futures. In some parallel applications, such as parallel, load-balancing garbage collectors and graph-connectivity algorithms, performance can be improved by reducing the cost of synchronizing parallel tasks. Our interface gives the programmer a few critical building blocks to aid in reducing such synchronization costs. In particular, through the interface, the programmer can specify, on a per-task basis, the strategy that should be employed for detecting when tasks become ready. We have implemented our interface and a number of strategies in a C++ scheduler.

1. Motivation

The fork-join and sync-spawn constructs, as well as futures, are useful abstractions for describing parallelism. They are, however, not always expressive enough. For example, the depth-first traversal of a graph can, in principle, be expressed as a fork-join program. But the resulting program will incur significant loss in efficiency due to the large number of join operations that it involves. An ideal interface for parallel computation would not only be expressive enough for the programmer to describe any parallel computation, but it would also care about efficiency.

DAMP'12

Any parallel computation can be viewed as a directed acyclic graph (DAG). A good interface for parallel computation must enable the programmer to construct arbitrary DAGs (at run time) by creating tasks (nodes in the DAG) and adding dependencies between tasks (edges in the DAG). Tracking dependencies between tasks is needed for determining when and how to execute tasks. However, this tracking may require significant synchronization overhead at run time. To control this overhead, the programmer needs to be able to select, based on the algorithm being implemented, the most appropriate strategy for implementing the joins.

In this paper, we present an interface for expressing arbitrary parallel computations, where the programmer can control the strategies employed for tracking dependencies, on a per-task basis. Our interface consists of only four functions: one to add a node to the DAG, one to add an edge to the DAG, one to notify the scheduler that the incoming edges of a node have been set up, and one to transfer the continuation of the currently-executing task to another task. When creating a node, the user specifies which *in-strategy* to use for counting the number of incoming edges for this node, and specifies which *out-strategy* to use for maintaining the list of outgoing edges from this node.

Our library provides a set of commonly-used in-strategies and out-strategies, but also accommodates user-defined strategies that might be better-suited to particular algorithms. The pre-defined strategies include strategies based on atomic instructions such as fetch-and-add or compare-and-swap, as well as strategies that rely solely on the message-passing communication. In particular, the latter can be used to generalize our previous work on the implementation of a work stealing scheduler that improves efficiency by minimizing use of atomic instructions [4].

The idea of allowing parallel computations to be expressed as computation DAG's is not new. For example, the work-time framework [1] for understanding efficiency of parallel programs critically takes advantage of the model. Use of the DAG model for expressing parallel programs is, however, less common. To the best of our knowledge, Matsakis and Gross's proposal is the only interface that allows expressing arbitrary computation DAGs at run time [3].

```
type task
task* add_task(closure* c, instrategy* i,
                           outstrategy* o)
void add_dependency(task* t1, task* t2)
void init_task(task* t)
outstrategy* capture_outstrategy()
```

Figure 1. Functions for building the computation DAG

```
task* add_ready_task(closure* c, outstrategy* o)
  instrategy* i = new instrategy_ready()
  task* t = add_task(c, i, o)
  init_task(t)
  return t

task* make_join(closure* c, instrategy* i)
  outstrategy* o = capture_outstrategy()
  return add_task(c, i, o)

void fork(closure* c, task* tj)
  outstrategy* o = new outstrategy_unary()
  task* t = add_ready_task(c, o)
  add_dependency(t, tj)

void fork2_join(closure* c1, closure* c2,
                closure* cj, instrategy* i)
  task* tj = make_join(cj, i)
  fork(c1, tj)
  fork(c2, tj)

// per-thread variable
set<task*> spawned = set_empty()

void spawn(closure* c)
  outstrategy* o = new outstrategy_unary()
  task* t = add_ready_task(c, o)
  spawned.add(t)

void sync(closure c, instrategy* i)
  task* t = make_join(c, i)
  spawned.iter(fun ti→ add_dependency(ti,t))
  spawned = set_empty()
  init_task(t)
```

Figure 2. Encoding of particular parallelism constructs

That work, however, does not consider the effect of in- and out-strategies on efficiency. Intel's Thread Building Blocks (TBB) [2] provides an interface for building "flow graphs", which are computation graphs that can accommodate fork join and some pipelined computations. The TBB interface allows customization of "joins", which are analogous to our in-strategies, but TBB provides nothing analogous to our out-strategies. Our interface is the first to support customizable in- and out- strategies. In Section 3, we describe a few examples of where this extra flexibility is useful.

2. Interface

Figure 1 shows the interface offered to the programmer for building DAGs. The type of a task is left abstract. Internally, a task consists of a closure (i.e., objects with a `run` method), an in-strategy and an out-strategy. The interface consists of four functions. The first one allows adding a task (i.e., a node) to the computation DAG. The creation of the task is parameterized by the representation of the in- and out-strategies, which the programmer can pick among a (user-extensible) list of strategies. The second one allows adding a dependency (i.e., an edge).

The third function needs to be called on the newly-created nodes, once their incoming edges have been set up. This function is needed for detecting tasks that have zero incoming edges. It is also exploited by particular in-strategies which make the assumption that all the incoming edges are added before the any of its dependency terminates. For other in-strategies, it is possible to add incoming edges after the call to the initialization function. The fourth function can be used to capture the set of outgoing edges from the currently-executing node. Such capture is useful for dynamically expanding a computation into a sub-DAG while maintaining dependencies correctly. More precisely, the function returns the out-strategy of the currently-executing node, and replaces it with the empty out-strategy, i.e., the out-strategy that corresponds to having zero outgoing edges.

Figure 2 shows the encoding of several standard parallelism constructs our DAG model. The content of the figure begins with three auxiliary functions. The first function, `add_ready_join`, creates a task with a particular in-strategy that indicates that the task has zero incoming edges, and it notifies the scheduler of the existence of this ready task. Function `make_join` helps creating a join task whose continuation is the continuation of the currently-executing task. It takes as argument the closure that corresponds to the join task and the in-strategy to be used for keeping track of the number of edges incoming on this join task. The function `fork` creates a new node, containing the closure passed as first argument, and it sets the join task passed as second argument as target of the single outgoing edge of this new node.

Function `fork2_join` corresponds to the standard binary fork-join construct. It creates two nodes, both of them pointing to a join task, whose continuation corresponds to the current continuation. The function thus takes four arguments: the closures associated with the two nodes, the closure associated with the join task, and the in-strategy for the join task (usually `instrategy_fetch_add`). An example of use of a binary fork-join appears in Figure 3.

Fork-joins of variable arity can be constructed using the functions `make_join` and `fork`, generalizing the pattern that appears in the definition of `fork2_join`. Typically, the `fork` function is called from within a loop. Note that the function `check_readiness` needs to be called on the join task at the end of the loop if this loop may fork zero tasks. Another way of creating fork-joins of variable arity consists in calling the function `spawn` on each task to be forked and then calling `sync` in order to specify the task on which all the forked tasks should join. These slightly more specific functions save the programmer the burden of manipulating a task pointer. Due to lack of space, we do not describe the encoding of

```
void fib(int n, int* r)
  if n < 2 then *r = 1 else
    int* r1 = new int*
    int* r2 = new int*
    fork2_join (fun () → fib(n-1, r1))
               (fun () → fib(n-2, r2))
               (fun () → *r = *r1 + *r2;
                         free r1; free r2)
```

Figure 3. Fibonacci, in explicit destination-passing style

```
class instrategy
  task* t // back-pointer to the task
  void init()
  void delta(int d)
  void msg_delta(int d)

class outstrategy
  task* t // back-pointer to the task
  void add(task* td)
  void finished()
  void msg_add(task* td)
```

Figure 4. Signature for instrategies and outstrategies

futures and lazy futures, which can also be encoded in terms of the four basic functions and with help of particular in- and out-strategies.

We expect the programmer to use the functions from Figure 2 whenever applicable, and to manipulate tasks and dependencies manually only for programs where the these functions are not expressive enough.

3. Strategies

A crucial ingredient to achieving efficient scheduling is the ability to detect when tasks become ready. If we were in a sequential world, it would very simple: each task would be associated with a counter and a list of pointers. The counter gives the number of incoming dependencies on this task, and the list of pointers describes the list of outgoing dependencies. Whenever a task terminates, the target of each of its outgoing edges see their incoming counter decremented by one. In a multithreaded world, however, the implementation of counters and of set of outgoing edges is a lot more subtle.

In this section, we describe several useful in- and out-strategies, covering both strategies using atomic operations such as fetch-and-add, and strategies relying solely on messages for resolving data races. The code for the core of the scheduler, which is responsible for maintaining the set of ready tasks and for executing these ready tasks one by one, appears in Figure 5. The code covers the implementation of the four functions from our interface, plus the main loop that controls the execution of each thread. This code helps understanding how the scheduler interacts with the in- and the out-strategies, which will be explained later on.

The high-level presentation of Figure 5 is as follows. When a task is created, it is added to the set of ready tasks. When a dependency is created between a task A and a task B, the A's out-strategy and B's in-strategy are notified. When

```
type task = { closure* c, instrategy* i,
                          outstrategy* o }
set<task*> ready_tasks // per-thread
task* my_current_task  // per-thread

task* add_task(closure* c, instrategy* i,
                           outstrategy* o)
  task* t = new task(c, i, o)
  i.t = t; o.t = t; return t

void add_dependency(task* t1, task* t2)
  t1.o.add(t2)
  t2.i.delta(+1)

void init_task(task* t)
  t.i.init()

outstrategy* capture_outstrategy()
  outstrategy* o = my_current_task.o
  my_current_task.o = new outstrategy_void()
  return o

void main() // per-thread
  while true do
    while ready_tasks.is_empty() do
      communicate_with_other_threads()
    task* t = ready_tasks.pop()
    my_current_task = t
    t.c.run()
    t.o.finished()
```

Figure 5. Implementation of the interface in a scheduler

```
void schedule(task* t)
  ready_tasks.add(t)

void decr_dependencies(task* t)
  t.i.delta(-1)

type message =
  | MSG_DELTA_IN(instrategy* i, int d)
  | MSG_ADD_OUT(outstrategy* o, task* td)

void handle_message(message m)
  match m with
  | MSG_DELTA_IN(i,d) → i.msg_delta(d)
  | MSG_ADD_OUT(o,td) → o.msg_add(td)

void send(int id_thread, message m);
```

Figure 6. Helper functions for implementing strategies

a task A completes its execution, the out-strategy of a task is responsible for notifying the in-strategy of each task that depends on task A.

The scheduler also provides several functions, shown in Figure 6, that are used to implement in- and out-strategies. A call to `schedule(t)` indicates the to scheduler that the task `t` has just become ready. A call to `decr_dependencies(t)` can be made by an out-strategy to notify the in-strategy of the task `t` that one dependency has just become satisfied. Moreover, for the implementation of message-based strategies, we assume that each thread is polling for messages on a regular basis and invoking the function `handle_message`

```
class outstrategy_void extends outstrategy
  void finished() { free this }
  void add(task* td) { assert false }
  void msg_add(task* td) { assert false }

class outstrategy_unary extends outstrategy
  task* tone = NULL
  void add (task* td) { tone = td }
  void finished() {
    decr_dependencies(tone); free this }
  void msg_add(task* td) { assert false }

class outstrategy_message extends outstrategy
  int master = my_id
  list<task*> ts = nil
  void add(task* td) {
    if my_id == master
      then master_add(td)
      else send(master, MSG_ADD_OUT(this, td)) }
  void msg_add(task* td) { master_add(td) }
  void master_add(task* td) { ts.cons(td) }
  void finished() {
    ts.iter(decr_dependencies); free this }
```

Figure 7. Implementation of particular outstrategies

on each received message. A function `send` is provided by the scheduler for sending messages.

The signature for out-strategies appears in Figure 4. The function `add(td)` is used to add a task `td` to the set of dependencies. The function `finished` is invoked by the scheduler on completion of the task, indicating the out-strategy that it should notify all the dependencies. The function `msg_add` is used by message-passing implementations. Figure 7 shows the implementation of three out-strategies: one for the case where there is no outgoing edges, one for the case where there is exactly one outgoing edge, one that handles the general case using messages. By lack of space, we do not show the strategy based on concurrent lists or per-thread lists, nor the strategies specialized for futures and lazy futures.

The signature for in-strategies appears in Figure 4. The function `init` is called when the user calls `init_task`. The function `delta` can be used to update the number of incoming dependencies. The function `msg_delta` is used by message-passing implementations. Figure 8 starts with a template class for in-strategies. The template class, called `instrategy_def` assumes the existence of a function called `check`, which tests whether there is no remaining dependencies. It offers an auxiliary function `start` for scheduling the task and deleting the current in-strategy.

The rest of Figure 8 contains three particular in-strategies: one specialized for tasks that are always ready, one that handles the general case using fetch-and-add atomic operations, and one called the *optimistic in-strategy*. The latter strategy, introduced in our earlier work [4], relies on a combination of non-atomic decrement operations and messages to avoid expensive fetch-and-add operations while still being able to recover from the (rare) case where a race occurs. Note that the optimistic strategy assumes that no incoming edge is added

```
class instrategy_def extends instrategy
  void check()
  void init() { check() }
  void start() { schedule(t); free this }
  void msg_delta(int d) { assert false }

class instrategy_ready extends instrategy_def
  void check() { start() }
  void delta(int d) { assert false }

class instrategy_fetch_add extends instrategy_def
  int counter = 0
  void check() {
    if counter == 0 then schedule(t) }
  void delta(int d) {
    int old_counter = fetch_add(counter, d)
    if old_counter == 1 then schedule(t) }

class instrategy_optimistic extends instrategy_def
  int master = my_id
  int mcounter = 0
  int scounter = 0
  void init() { scounter = mcounter; check() }
  void check() { if (mcounter == 0) then start() }
  void delta(int d) {
    if (d > 0)
      assert (my_id == master)
      mcounter += d
    else
      if (my_id == master)
        mcounter += d
        scounter += d
        check()
      else
        if (scounter + d == 0)
          scounter = -1
          schedule(t)
        else
          scounter += d;
        send(master, MSG_DELTA_IN(this, d)) }
  void msg_delta(int d) {
    mcounter += d
    if (mcounter == 0)
      if (scounter != -1)
        schedule(t)
      free this }
```

Figure 8. Strategies for detecting readiness of tasks

after the dependencies start executing. Due to lack of space, we do not show the strategy based solely on messages, nor the strategy that uses one counter per thread and assigns one thread the responsibility to check whether the sum of all the counters reaches zero. Such a *distributed in-strategy* is useful in particular for detecting termination of graph traversal.

[1] Richard P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21(2):201–206, 1974.

[2] Intel. Intel threading building blocks. 2011.

[3] Nicholas Matsakis and Thomas Gross. Programming with intervals. In *LCPC*, volume 5898 of *LNCS*, pages 203–217. 2010.

[4] Mike Rainey Umut A. Acar, Arthur Charguéraud. Efficient synchronization-free work stealing. Draft., August 2011.