Design and Implementation of the OpenMP 4.0 Task Dataflow Model for Cache-Coherent Shared-Memory Parallel Systems in the Runtime of the OMPi OpenMP/C Compiler

Technical University of Crete

School of Electronic and Computer Engineering

Anastasios Souris

September 13, 2015

Contents

List of Figures

List of Tables

Chapter 1

Introduction

OpenMP provides an application programming interface (API) for shared-memory parallel programming. It extends the C/C++ and Fortran programming languages with *directives* that the programmer can use to express *fork-join* and *task* parallelism inherent in the algorithm to the OpenMP compiler. In C, the programming language used for this thesis, OpenMP directives are specified with the **#pragma** preprocessor directive mechanism that is used to access compiler-specific preprocessor extensions for the C compiler. The syntax of an OpenMP directive informally is as follows:

#pragma omp *directive-name [clause[[,] clause ...] new-line*

Each OpenMP directive has specific clauses that can be specified in order to adjust the behaviour of the directive. In addition, OpenMP provides a collection of library routines collectively knows as the OpenMP API runtime library, as well as environment variables that affect the execution environment of an OpenMP program.

1.1 Fork-Join Parallelism in OpenMP

In the fork-join model of parallel computing, a *master thread* creates a *parallel team* comprised of itself and zero or more other threads at a *fork-point* to collectively execute independent units of work. After the completion of the parallel computation, the members of the parallel team rendezvous at a *join-point*. Thereafter, the master thread continues execution, whereas the rest are destroyed. Figure 1.1 illustrates the concept of fork-join parallelism.

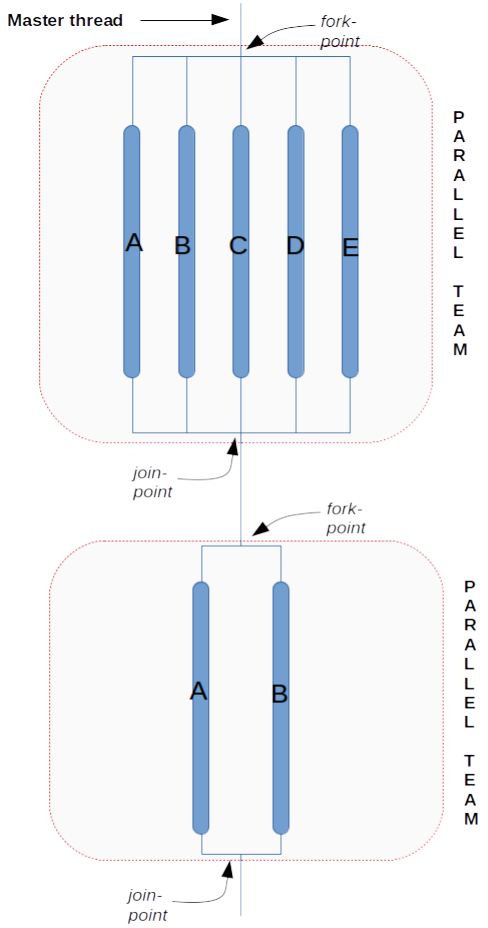

Figure 1.1: The master thread first forks a parallel team of 5 threads *A,B,...,E*. Then, it forks another parallel team of 2 threads *A,B*. In each of the two parallel teams, the threads perform independent units of work.

A OpenMP program starts execution with a single OpenMP thread called the *initial thread*. A OpenMP thread is an execution entity with a stack and associated static memory called *threadprivate memory* that is managed by the OpenMP runtime system. A fork-point is expressed with the **parallel** construct. There is an implied barrier at the end of the parallel construct which serves as the join-point.

The syntax of the parallel construct is as follows:

#pragma omp parallel *[clause[[,] clause ...] new-line*
structured-block
where *clause* is one of the following:

- **if**(*scalar-expression*)
- **num_threads**(*integer-expression)*
- **default**(*shared—none*)
- **private**(*list*)
- **firstprivate**(*list*)
- **shared**(*list*)
- **proc_bind**(*master—close—spread*)

Whenever a OpenMP thread encounters a parallel construct, it creates a parallel team, whose size is determined by the *num_threads* clause and *if* clauses, and becomes the master of the new team. The threads comprising the parallel team execute the *structured-block* and are assigned consecutive identifiers ranging from 0 for the master thread up to one less than the number of threads in the parallel team for the rest. The runtime library routine *omp_get_num_threads()* returns the number of threads in a parallel team and the runtime library routine *omp_get_thread_num()* returns the identifier within the current parallel team of the calling thread.

All code encountered during execution of the structured-block is referred to as a *parallel region*. The data-sharing attributes of variables referenced in the parallel region are determined with the *default*, *private*, *firstprivate* and *shared* clauses. Briefly, a variable referenced in the parallel region is either shared or private to the members of the parallel team. If a variable is listed in a *shared* clause it is shared, whereas if it is listed in a *private* or *firstprivate* clause it is private. The value of the variable when the parallel construct is encountered is irrelevant in the case of the *private* clause, but when the *firstprivate* clause is used the current value of the variable is used to initialize the private copies of the variable for the members of the parallel team. The *default* clause may be used to specify all variables referenced in the parallel region and not listed in a *private*, *firstprivate* or *shared* clause to be shared. The purpose of the *proc_bind* clause is to control the binding of the OpenMP threads comprising the parallel team to the available hardware processing units. Figure 1.2 shows an example of the parallel construct. Function *sub* creates a parallel team that share the variables x and *npoints*, and have private copies of the variables *iam*, *nt*, *ipoints* and *istart*. The members of the parallel team execute the same structured-block. Each member of the parallel team, depending on its identifier *iam*, selects a subset of the array pointed to by x starting at index *istart* with length *ipoints* to process with the *subdomain* function.

```
1  #include <omp.h>
2
3  void subdomain(float *x, int istart, int ipoints)
4  {
5        int i;
6
7        for (i = 0; i < ipoints; i++)
8              x[istart+i] = 123.456;
9  }
10
11 void sub(float *x, int npoints)
12 {
13       int iam, nt, ipoints, istart;
14
15       #pragma omp parallel default(shared) private(iam, nt, ipoints, istart)
16       {
17             iam = omp_get_thread_num();
18             nt = omp_get_num_threads();
19             ipoints = npoints/nt; /* size of partition */
20             istart = iam*ipoints; /* starting array index */
21             if (iam == nt - 1) /* last thread may do more */
22                   ipoints = npoints - istart;
23             subdomain(x, istart, ipoints);
24       }
25 }
26
27 int main()
28 {
29       float array[10000];
30
31       sub(array, 10000);
32
33       return (0);
34 }
```

Figure 1.2: OpenMP parallel-construct example

In order to support work-sharing between the members of the parallel team, OpenMP provides work-sharing constructs (refer to figures 1.3, 1.4 and 1.5 for examples):

Loop Construct The loop construct specifies that the iterations of one or more associated loops may be distributed to the members of the parallel team for parallel execution.

Sections Construct The sections construct allows for the distribution of a set of structured-blocks for parallel execution among the members of the parallel team.

Single Construct The single construct specifies that the associated structured-block must be executed exactly once by some member of the parallel team.

```
1  void parallel_loop_example(int n, float *a, float *b)
2  {
3       int i;
4
5       #pragma omp parallel for
6       for (i = 1; i < n; ++i)
7       {
8            b[i] = (a[i] + a[i-1])/2.0;
9       }
10 }
```

Figure 1.3: The parallel loop construct, which has semantics equivalent to a parallel directive immediately followed by a loop construct, distributes the iterations of the loop 1 to n-1 to the members of the parallel team.

```
1  void parallel_sections_example()
2  {
3       #pragma omp parallel sections
4       {
5            #pragma omp section
6            A();
7
8            #pragma omp section
9            B();
10
11           #pragma omp section
12           C();
13      }
14 }
```

Figure 1.4: The parallel sections construct, which has semantics equivalent to a parallel directive immediately followed by a sections construct in the same way as the parallel loop construct, distributes the function invocations *A()*, *B()* and *C()* to the members of the parallel team.

```
1  void single_example()
2  {
3       #pragma omp parallel
4       {
5            #pragma omp single
6            {
7                 printf("Hello World!“n");
8            }
9       }
10 }
```

Figure 1.5: The *printf()* statement associated with the single construct will be executed exactly once and, thus, the program will output the string *Hello World!* once.

1.2 Task Parallelism in OpenMP

The task parallel programming model is well-suited for irregular, dynamic and unstructured parallelism. To enhance programmer productivity, task parallelism requires the programmer only to identify independent units of work called *tasks* that are dynamically generated and can be executed asynchronously, and not to concern themselves with scheduling those tasks. OpenMP provides support for task parallelism through the usage of the **task** construct that defines an explicit task which essentially is a unit of parallel work that may be executed by any member of the parallel team. The syntax of the task construct for OpenMP 3.1 is:

#pragma omp task *[clause[[,] clause ...] new-line*
structured-block
where *clause* is one of the following:

- **final**(*scalar-expression*)
- **untied**
- **default**(*shared—none*)
- **mergeable**
- **private**(*list*)
- **firstprivate**(*list*)
- **shared**(*list*)

The structured-block associated with the task construct defines the unit of parallel work that is generated. The OpenMP tasking model allows the execution of the generated task to be suspended in *task scheduling points*, in which case by default the suspended task can only be resumed by the thread that started its execution, unless the *untied* clause is specified that allows any thread in the parallel team to resume the task after its suspension. The data-environment of the task can be controlled with the *default*, *private*, *firstprivate* and *shared* clauses similarly to the parallel construct. When an *if* clause is present and its associated *scalar-expression* evaluate to true, then the task is immediately executed. The *final* clause with a true *scalar-expression* indicates that all tasks generated by the newly created task must be executed sequentially. The *mergeable* clause indicates that the generated task may share its data-environment with its parent task (the task that generated the new task). Tasks that are children of the same task are called *sibling* tasks.

1.2.1 Task Synchronization

The OpenMP 3.1 version of the OpenMP standard provides the following means for task synchronization:

Explicit and Implicit Barriers There exists an implicit barrier at the end of the parallel, for, single and sections construct unless a *nowait* clause is specified for that construct. An explicit barrier is specified by the **barrier** directive which is a stand-alone OpenMP directive with no associated structured-block. The syntax of the *barrier* construct is: **#pragma omp barrier** new-line

The taskwait construct The **taskwait** construct suspends the execution of the task until all tasks that it has generated up to the point where the *taskwait* construct is placed are completed. The syntax of the *taskwait* construct is: **#pragma omp taskwait** new-line

Figure 1.6 shows an example of a C function that generates explicit tasks using the OpenMP *task* construct to perform some operation on all elements of a list and relying on implicit barriers for task synchronization Figure 1.7 shows a simple implementation of a function that computes the n-th fibonacci number using tasks and the *taskwait* construct for task synchronization.

```
1  typedef struct node node;
2
3  struct node
4  {
5        int data;
6        node *next;
7  };
8
9  void process(node *p)
10 {
11       /* do work here */
12 }
13
14 void increment_list_items(node *head)
15 {
16       #pragma omp parallel
17       {
18             #pragma omp single
19             {
20                   node *p = head;
21
22                   while (p)
23                   {
24                         #pragma omp task
25                         process(p);
26
27                         p = p->next;
28                   }
29             }
30       }
31 }
```

Figure 1.6: Due to the single construct some member of the parallel team will traverse the list generating an explicit task that calls the function *process()* for all nodes of the list. The tasks shall be executed by the members of the parallel team.The implicit barrier at the end of the single construct guarantees that all task will be executed and thus the function *process()* applied to all nodes of the list before the function returns.

```
1  int fib(int n)
2  {
3      int i, j;
4
5      if (n<2)
6          return n;
7      else
8      {
9          #pragma omp task shared(i)
10         i = fib(n-1);
11
12         #pragma omp task shared(j)
13         j = fib(n-2);
14
15         #pragma omp taskwait
16         return i+j;
17     }
18 }
```

Figure 1.7: A call to *fib(n)* generates two tasks: the first task computes *fib(n-1)* and the second task computes *fib(n-2)*. The parent task has to wait for those two tasks to complete using the *taskwait* construct before it can sum the partial results and return the final answer.

1.2.2 Motivation for Dataflow

OpenMP 3.1 lacks the ability to specify *point-to-point* synchronization, that is synchronization between specific tasks that constraints the order of execution of those tasks. The advantages of *point-to-point* synchronization over *global* synchronization that is currently available by means of implicit and explicit barriers, as well as the *taskwait* construct, are:

Parallelism Global synchronization often inhibits the parallelism inherent in the algorithm because even though a task may have a dependency on only some of the previously generated tasks, the OpenMP programmer is enforced to insert either a *barrier* or a *taskwait* prior to the generation of that task which has the effect of waiting on the completion of *all* previously generated tasks. Figure 1.8 shows the dependencies between the tasks of a realistic application that we shall revisit later. Each rectangle represents a task and an arrow from task A to task B represents a dependency of B on A, that is that B cannot start execution before A terminates. A task-based implementation using the *taskwait* construct for synchronization would first execute the upper left task (shown in red). Then, all the tasks on the first row and on the first column can execute (shown in green). Before any of the blue tasks can be generated the programmer is enforced to insert a *taskwait* thereby waiting on the completion of all greek tasks before it can generate the blue tasks. This solution doesn't exploit all the parallelism in this task graph, though, because observe that any blue task is ready to execute when the green tasks on the same row and column have terminated and not after the completion of all green tasks.

Synchronization Cost Especially compared to *barrier* synchronization using either implicit or explicit barriers, *point-to-point* synchronization incurs significantly less synchronization costs, due to the fact that the former requires synchronization between all members comprising the parallel team, whereas the latter only between the participant tasks.

The OpenMP 4.0 standard provides support for *point-to-point* synchronization by means of the **task dataflow** parallel programming model which will be explained thoroughly in chapter 3.

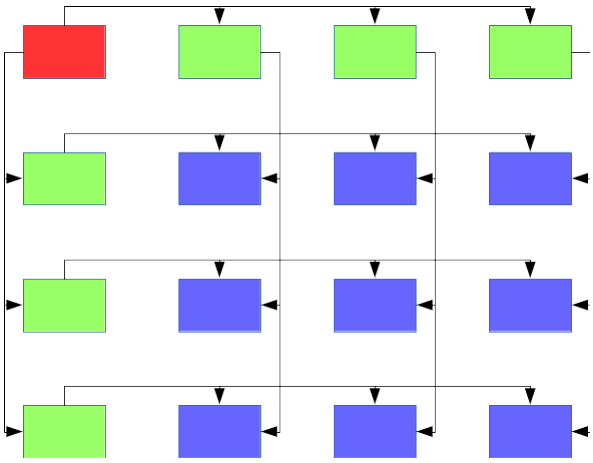

Figure 1.8: Each rectangle represents a task and an arrow from task A to task B means that task A must execute before task B. A task-based implementation of this task graph using the *taskwait* construct would first execute the red task, then the green tasks and, at last, the blue tasks, thereby loosing parallelism because each blue task can be executed after the two green tasks on the same row and column have terminated.

1.3 Thesis Contribution and Organization

The purpose of this thesis is the implementation of the OpenMP 4.0 task dataflow model for shared-memory parallel systems in the runtime of the open source OMPi OpenMP/C compiler. This thesis is organized as follows:

Chapter 2 : Architectural Background This chapter provides necessary background on shared-memory parallel systems that is needed for the design and implementation of the task dataflow model in the runtime of the OMPi OpenMP/C Compiler.

Chapter 3 : The OpenMP 4.0 Task Dataflow Model This chapter explains in detail how the OpenMP 4.0 standard supports *point-to-point* synchronization by means of the task dataflow programming model.

Chapter 4 : A Lock-Free List Scheme In this chapter i describe a variant of the scheme that is presented in Chapter 3 which is lock-free.

Chapter 5 : Performance Evaluation The aim of this chapter is to evaluate the effectiveness of the implementation through experiments on both micro benchmarks and realistic applications (lu factorization, strassen multiplication and a 2D recurrence application).

Chapter 6 : Conclusion This final chapter concludes this thesis.

Chapter 2

Architectural Background

This chapter provides the necessary background on cache-coherent shared-memory parallel systems.

2.1 Components of a Cache-Coherent Shared-Memory Parallel System

A generic cache-coherent shared-memory multiprocessor architecture consists of processor nodes that are connected through an interconnection network. Each processor node has a processor (P) and a cache hierarchy (C). The processor (P) is equipped with one or more cores which may be multi- or single- threaded and are either homogeneous or heterogeneous in their capabilities. A multi-core processor is one with multiple cores on the same chip and is often referred to as a Chip Multiprocessor (CMP). The cores on a CMP are connected through on-die interconnects. The processors share the same global address space and can load and store to any memory location via the interconnection network. In a Uniform Memory Architecture (UMA) (see 2.1) all memory modules are equally distant from every processor node. On the other hand, in a Non-Uniform Memory Architecture (NUMA) (see 2.2), each processor node contains a local memory module which it can access faster than the memory modules local to other processor nodes. Consequently, memory access time in a NUMA machine depends on the location of the memory relative to the processor node issuing the memory access. Typically, the topology of the interconnection network in a UMA machine is a broadcast-based interconnection medium such as a *bus*. On the contrary, NUMA machines often comprise of a large number of processor nodes that need to be connected to the interconnection network and a bus is ill-suited for this scenario. To increase memory bandwidth and decrease memory latency NUMA machines employ a scalable interconnection network that ideally provides bandwidth that scales linearly with the number of processor nodes and memory access latency that grows sub-linearly with the number of processor nodes.

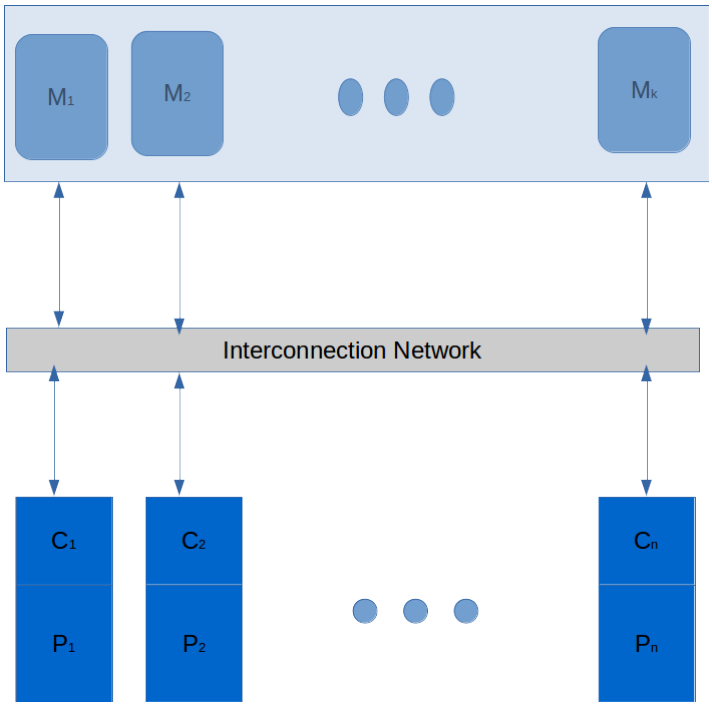

Figure 2.1: A Uniform Memory Access machine consisting of k memory modules and n processor nodes

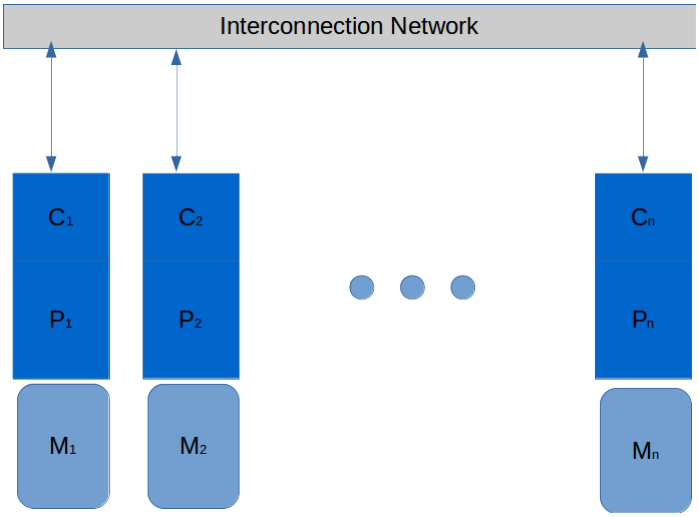

Figure 2.2: A Non-Uniform Memory Access machine consisting of n NUMA nodes.

2.2 Shared Memory Correctness

In a shared memory architecture all cores have access to the same single shared address space. Correctness in a shared memory architecture is defined by two properties, namely *coherence* and *consistency*.

2.2.1 Cache Coherency

A cache is a fast local memory storage used by the processor to reduce memory access times, by storing copies of the recently and most frequently used instructions and data from main memory. The processor cache is based on the principles of *spatial* and *temporal locality*. Spatial locality means that if the algorithm accesses memory location A, most probably it will access a location nearby shortly after. Temporal locality is a property of algorithms that tend to access the same working set repeatedly in a short period of time.

By storing the recently accessed data from main memory into the cache, the processor takes advantage of the temporal locality of the algorithm. To provision for spatial locality, the processor increases the granularity of data accessed from main memory. Instead of fetching only the requested data from main memory, the processor rather fetches a *cache-line* consisting typically of 64 bytes into the cache. Moreover, the processor contains sophisticated *prefetching* mechanisms that fetch memory locations from main memory to local cache before they are requested thus minimizing memory access latency.

Every memory access to a memory location by the processor first consults the cache and if that memory location resided in the cache then we have a *cache-hit*. Otherwise, the processor incurs a *cache-miss* and must reach out to main memory to fetch that memory location into the cache.

Coherency and Correctness Cache coherency deals with the issue of multiple copies of the same memory location in the local caches of different cores. The problem is that if two different cores A and B load a memory location into their local caches and one of them updates its own local copy, the local copy of the other core is *invalid*, a undoubtedly undesirable situation.
To begin with, one should define precisely the requirements for coherency. In order to define coherence, the lifetime of a memory location is divided into *epochs*. In each epoch, either a single core may read and write to that memory location or any core may read that memory location. Formally, coherence may be defined by the following two invariants:

Single-Writer, Multipler-Reader (SWMR) Invariant For any memory location A, at any given (logical) time, there exists only a single core that may write to A (and can also read it) or some number of cores that may only read A.

Data-Value Invariant The value of the memory location at the start of an epoch is the same as the value of the memory location at the end of its last read-write epoch.

Intuitively, the coherence invariants state that a read from a memory location obtains the value of the immediately preceding write to that memory location in logical time, and that writes are performed sequenced as they are issued in logical time.

To maintain the coherence invariants, the system implements coherence protocols that are based on the idea of *invalidation*. In an invalidation-based cache-coherence protocol, a store to a memory location A by some core causes the local copies of that memory location in the other cores to be invalidated. Any access to memory location A by a core is a cache-miss if either that memory location is not cached in the core's local cache or the local copy has been invalidated. Otherwise, the access is a cache-hit. A non-scalable implementation of the invalidation protocol is the *snooping protocol* for bus interconnection networks, whereby each core monitors the bus for any request and takes proper action to either invalidate or update the local cache if necessary. For non-bus interconnection networks, a *directory-based scheme* is often employed for scalability. In a directory-based scheme, there is a logical directory that contains information on the local copies of each memory location. Typically, the logical directory is partitioned in the memory modules and the state of a memory location is located in the directory belonging to the same memory module as the memory location. The directory keeps track, for each memory location, the core that maintains the most recent copy of that memory location, called the owner core. A read request consults the directory to find the owner core and then fetches the value of the memory location from the owner core in case of a cache-miss. A write request has to also update the directory to reflect the new owner of the memory location.

Performance Implications of Cache Coherency To efficiently utilize the cache hierarchy of the architecture, the algorithm must contain high degrees of spatial and temporal locality. As fas as concurrency is concerned, the following can be observed:

True Sharing True sharing occurs when different cores access the same data. Performance, then, degrades for an algorithm performing lots of store operations because these operations lead to cache invalidations and cache-misses for later accesses by other cores. If true sharing cannot be avoided it should be constrained within processor nodes since communication between cores inside a processor node is fast due to the shared last-level cache, whereas communication between cores from different processor nodes involves the interconnection network thus increasing the latency of the access.

False Sharing The granularity of the cache coherency protocol is typically the cache line size and not each individual memory location. As a result, if core A accesses memory location X and core B accesses a different memory location Y, but X and Y happen to coexist in the same cache-line, any update from core A to X invalidates the local copy of Y for core B (and vice-versa), thereby causing a cache-miss on the accesses of core B to Y even though those cores do not access the same data. To avoid false-sharing the data accessed by a core can be padded to fit a cahce-line.

2.2.2 Memory Consistency

A memory consistency model is a contract between the shared memory architecture and the programmer that defines how load and store operations relate to each other.

The Need for a Memory Consistency Model. In order to understand why a memory consistency model is necessary, we look at table 2.1 which is a classic example of the popular *producer/consumer idiom*. The purpose of the producer/consumer idiom is for one core, in this case core A, to pass some data to another core, Core B in this example. This is done, by having the producer core B first writing the data (S1) and then publishing the data by setting a shared flag to true (S2). On the other hand, the consumer core B waits until the shared flag becomes true (L1 and B1) and then reads the data (L2). Normally, a programmer would expect that core B reads the value 1 from data, because S1 precedes S2 and L2 is executed only after core B reads true from flag at L1, meaning that at that time S1 precedes L1 which precedes L2 and, thus, S1 precedes L2 and, consequently, core B must read the value stored to variable data by S1. However, the shared memory hardware may *reorder memory accesses*, thus violating the aforementioned reasoning. As an example, if the hardware reorders the store operations by core A, S1 and S2, it would no longer be the case that S1 precedes S2 and core B could indeed read the old value of the variable data.

Core A	Core B
//Initially data = 0 and flag = false	L1: local_flag = flag;
S1: data = 1;	B1: if (!local_flag){ goto L1; }
S2: flag = true;	L2: local_data = data;

Table 2.1: A procuder/consumer idiom. Core A first populates some data using a store operation (S1) and then publishes the data by setting a global flag to true with a second store operation (S2). Core B, awaits Core A to publish the data by continuously reading the value of the shared flag (L1) until it becomes true (B1). Then, Core B proceeds to read the shared data (L2). The question is whether the data read by Core B (L2) contains the value 1 or 0.

Sources of Inconsistency First of all, accesses by some core to the same memory location are required to be executed by the hardware in a manner that coincides with the order they are issued by the program to preserve sequential execution. Hence, reordering may occur only between memory accesses by the same core to *different* memory locations. Considering only *load* and *store* operations as memory accesses by some core, the possible ways that the hardware may reorder these memory accesses are the following:

Store-Store Reordering Two store operations to separate memory locations may be reordered if the core uses a non-FIFO write buffer and, as a consequence, write operations depart from the write buffer in a order different from the order in which they arrived in the first place. The write buffer, which is also named store buffer, is explained later.

Load-Load Reordering An out-of-order core may execute two load operations on separate memory locations in a different order from the one they were issued by the program, since from the core's perspective there is no data dependency between these two instructions. In the producer/consumer example of table 2.1, reordering the load operations from core B, L1 and L2, as in the example execution of L2, S1, S2 and L1, would result again in core B loading the old value of data.

Load-Store and Store-Load Reordering These reorderings could arise due to an out-of-order execution. Table 2.2 illustrates the effect of a store-load reordering.

Last but not least, compiler optimizations can also result in memory operations being reordered.

Core A	Core B
//Initially x = 0	//Initially y = 0
S1: x = 1	S2: y = 1
L1: local_y = y;	L2: local_x = x;

Table 2.2: Core A first writes variable x and then reads variable y, whereas Core B first writes variable y and then reads variable x. On the absence of memory reorderings, one would expect that at the end at least one of the local_y and local_x variables is 1, because if, for example, local_y would be 0, that would mean that Core B hasn't yet executed S2 and, as a result, when Core B arrives and executes S2 and then L2, Core A would have already executed S1 and this means that Core B would read 1 from x. On the other hand, if the hardware reorders the operations S1-L1 and S2-L2, then both cores could first load the initial values of x and y and then write the new values to them. As a result, store-load reordering in this example permits an outcome of (local_y, local_x) = (0,0). Such a reordering is possible for Total Store Order because the store operations S1 and S2 could reside in the store buffer when the load operations L1 and L2 are performed.

Specification of Memory Consistency Models To reason about memory consistency models we define the following orders on the memory accesses performed by a single core. We use L(a) and S(a) to denote a load operation and a store operation, respectively, on memory location a.

Program Order A total order on the memory operations issued by some core that reflects the order in which that core issues these memory operations. Program order is denoted by $<_{\mathrm{p}}$: L(a) $<_{\mathrm{p}}$ L(b) means that some core issued a load operation on memory location a, prior to issuing a load operation on memory location b.

Global Memory Order A total order on the memory operations of all cores as executed by the hardware. Global memory order is denoted by $<_{\mathrm{m}}$: L(a) $<_{\mathrm{m}}$ L(b) means that the load operation on memory location a by some core is executed before the load operation on memory location b by the same core.

Then, to define a memory consistency model it suffices to specify whether the program order is respected by the global memory order for each pair of memory operations possible (load-load, load-store, store-load, store-store).

Memory Fences Considering the possibility of memory reorderings, the architecture must provide the programmer with primitives that restrict the amount of memory reordering performed by the hardware, so that the programmer can correctly implement programs like the producer/consumer idiom we illustrated before. These primitives are called *memory fences* and, in principle, they enforce ordering between a memory operation and that memory fence that is respected by the global memory order. Henceforth, we assume that the architecture provides a memory fence operation named *FENCE* that preserves program order for the following set of pairs of operations:

- $Load \rightarrow FENCE$
- $Store \rightarrow FENCE$
- $FENCE \rightarrow FENCE$
- $FENCE \rightarrow Load$
- $FENCE \rightarrow Store$

2.2.3 Examples of Memory Consistency Models

In this section we take a closer look to three popular memory consistency models: *sequential consistency*, *total store order* and *relaxed memory consistency*.

Sequential Consistency Sequentially consistency allows no memory reorderings and is the most intuitive model that is assumed by most programmers. Informally, a shared memory system is sequentially consistent if the results of any execution is the same as if the operations of all cores were executed in some sequential order, and the operations of each individual core appear in this sequence in the order specified by its program. However, despite its ease of understanding and programming, no architecture implements sequential consistency because it precludes various architecture optimizations, such as store buffers.

Total Store Order Total Store Order is the memody model used by the x86 and SPARC architectures and, hence, is of practical interest. Total Store Order exists solely as a result of FIFO store buffers in processor architectures. A store buffer is used to hide the latency of store operations by temporary keeping them inside the store buffer before they are ready to be applied to the memory subsystem. Since the store buffer is FIFO, it follows that the only possible reordering is a store-load reordering, and, thus, the program orders respected by Total Store Order are:

- $Load \rightarrow Load$
- $Load \rightarrow Store$
- $Load \rightarrow FENCE$
- $Store \rightarrow Store$
- $Store \rightarrow FEnce$
- $FENCE \rightarrow Load$
- $FENCE \rightarrow Store$
- $FENCE \rightarrow FENCE$

Table 2.2 illustrates an example where a store-load reordering due to the presence of a store buffer results in a non-sequentially consistent execution under a Total Store Order memory consistency model. To prevent the store-load ordering a FENCE instruction must be placed between instructions S1 and L1, as well as between instructions S2 and L2. Note that the producer/consumer example of table 2.1 always produces sequentially consistent executions under a total store order memory model because no store-load reordering is possible.

Relaxed Memory Consistency The relaxed memory consistency model is a *weak* memory model, in the sense that it permits all memory reorderings so that more hardware and software (compiler and runtime system) optimizations are possible (non-FIFO, coalescing write buffer, core speculation). Thus, this model allows for maximum hardware performance but poor programmability because a FENCE instruction is needed whenever ordering is required by the program. The program orders respected by relaxed memory consistency are:

- $Load \rightarrow FENCE$
- $Store \rightarrow FENCE$
- $FENCE \rightarrow Load$
- $FENCE \rightarrow Store$
- $FENCE \rightarrow FENCE$

From the programmer's perspective, the performance advantages of a relaxed memory consistency model can be understood by the example shown in table 2.3. The requirements is that the following three conditions in order for the local_x, local_y and local_z local variables to contain the correct values.

- $S1 \rightarrow S4 \rightarrow L1 \rightarrow L2$, establishes that local_x = 1
- $S2 \rightarrow S4 \rightarrow L1 \rightarrow L3$, establishes that local_y = 1
- $S3 \rightarrow S4 \rightarrow L1 \rightarrow L4$, establishes that local_z = 1

Note that the FENCE instructions F1 and F2 ensure that the aforementioned three conditions hold. In particular, due to F1, $S1 \rightarrow S4$, since $S1 \rightarrow F1$ and $F1 \rightarrow S4$, and, similarly, $S2 \rightarrow S4$ and $S3 \rightarrow S4$. Morevoer, the F2 FENCE ensures that $L1 \rightarrow L2$, since $L1 \rightarrow F2$ and $F2 \rightarrow L2$ and, similarly, $L1 \rightarrow L3$ and $L1 \rightarrow L4$.

However, it is not necessary for $S1 \rightarrow S2 \rightarrow S3$ to be true as required by the Total Store Order memory model, nor for $L2 \rightarrow L3 \rightarrow L4$ to hold. Put another way, the store operations by Core A S1, S2 and S3 may be performed in any order prior to the memory fence instruction F1. In addition, the load operations by Core B L2, L3 and L4 may also be performed in any order.

Core A	Core B
S1: x = 1;	L1: local_flag = flag;
S2: y = 1;	B1: if (!local_flag) goto L1;
S3 : z = 1;	**F2: FENCE**
F1: FENCE	L2: local_x = x;
S4: flag = true;	L3: local_y = y;
	L4: local_z = z;

Table 2.3: Modified version of the producer/consumer idiom.

Performance Implications of Memory Fences A FENCE instruction is required to preserve program order with respect to a preceding load or store operation. Typically, this is accomplished by architectures by providing three kinds of memory fences: A *write fence* that guarantees that all store operations that are issued in program order prior to the write fence are committed to the memory subsystem, a *read fence* that does the same for load operations and, finally, a *full fence* that essentially is a write fence and a read fence simultaneously. For example, in the Total Store Order memory model only a write fence is needed to prevent a store-load reordering, because before performing the load operation we must ensure that all prior store operations have been committed and a write fence does just that. Since a shared memory architecture adhering to the Total Store Order memory model utilizes store buffers, a write fence in essence involves *flushing* the core's store buffer, that is waiting for all store operations currently in the store buffer to complete. As a consequence, the cost of a write fence can be equivalent to thousands of machine instructions thus reducing the performance of the program. The same can be said also for read and full fences. The programmer must, thus, write a program with the least number of memory fences required to avoid the extra cost incurred by each memory fence.

2.3 Architectural Primitives for Concurrency

Concurrent Algorithm A concurrent algorithm is an algorithm that is executed by a set of sequential processes. A sequential process represents a single thread of control. These sequential processes execute on the available processing units (cores in a cache-coherent shared-memory architecture) and may or may not be active at the same time depending on the scheduling of those sequential processes onto the available processing units.

Synchronization For any concurrent algorithm, the participating sequential processes require some form of *synchronization*. Synchronization means that the actions of one sequential process depends on the actions of the other participating sequential process. One can identify two types of synchronization:

Competition This form of synchronization occurs when the participating sequential processes compete to execute in isolation a block of statements which is called a *critical section*.

Cooperation In this form of synchronization the participating processes coordinate their actions in order to achieve a common goal. Cooperation occurs, for example, in the producer/consumer idiom since the producer and the consumer must coordinate so that they ensure that the consumer consumes only the data produced by the producer exactly once and in the correct order. Another popular form of cooperation is *barrier synchronization*, whereby the processes wait for each other before they can proceed.

The design process of a concurrent algorithm involves two important properties: (1) *safety* and (2) *liveness*. Safety properties state that nothing bad happens. This property alone, however, is not sufficient for the correctness of a concurrent algorithm because any concurrent algorithm that does nothing trivially satisfies the safety property but, indisputably, such a concurrent algorithm would be of negligible significance. That being the case, concurrent algorithms need to satisfy some *liveness* property that characterizes the progress of each participating sequential process.

To begin with, though, we classify any concurrent algorithm into two major classes:

Blocking Concurrent Algorithms If the process of some participating sequential process depends on the presence of another participating sequential process, then the concurrent algorithm is called *blocking* because that process would have to block waiting on the arrival of some other process.

Non-Blocking Concurrent Algorithms On the other hand, if any participating sequential process executes independently of the rest of the processes then the concurrent algorithm is called *non-blocking* because that process never has to block waiting on the arrival of some other process.

2.3.1 Blocking Concurrent Algorithms

A blocking concurrent algorithm is typically based on *mutual exclusion*. Mutual exclusion states that a block of statements called the *critical section* can be executed by at most one process at a time. The convention for critical sections is to provide two operations named *acquire_mutex* and *release_mutex*. The operation *acquire_mutex* provides access to the critical section and the operation *release_mutex* makes the critical section available for execution by some other process. The properties of blocking concurrent algorithms are:

Safety Property The mutual exclusion property is the safety property for any blocking concurrent algorithm.

Liveness Properties Depending on the number of steps some participating sequential process has to take in order to execute the critical section two common liveness properties are defined:

Starvation-Freedom Any invocation to the *acquire_mutex* operation eventually terminates

Deadlock-Freedom If any number of processes concurrently invoke the *acquire_mutex* operation then that operation succeeds for at least one of those processes.

The difference between starvation-freedom and deadlock-freedom is that the *acquire_mutex* operation always terminates if it is starvation-free but may not terminate if it is deadlock-free because it could happen that some process p issues a *acquire_mutex* operation and at the same time another process q repeatedly issues *acquire_mutex* operations with process q always succeeding in its operations and, as a result, process p failing to terminate its own operation.

The *acquire_mutex* and *release_mutex* operations are often provided by a object in programming environments that is called a *lock*. Moreover, programming environments typically provide higher level abstractions for the implementation of blocking concurrent algorithms which, among others, include *semaphores*, *monitors*, *readers-writers locks*, *barriers* and *condition variables*.

2.3.2 Non-Blocking Concurrent Algorithms

Since a non-blocking concurrent algorithm cannot use lock objects it has to rely on other forms of architectural primitives to implement synchronization. To start with, we are concerned with the safety property of non-blocking concurrent algorithms, for which we use the notion of *atomicity*.

Computation Model A non-blocking concurrent algorithm is executed by a finite set of n sequential processes, denoted $p_1, \ldots, p_n$. Those processes invoke operations on shared concurrent objects. Each object is defined by a sequential specification that specifies for each operation exposed by that object, its behaviour when invoked in isolation, that is when the object is accessed only sequentially. We model an invocation of some operation op() with arguments arg on an object X by process p_i by two events, the first being the invocation event that occurs when process p_i invokes the operation op() on the object X and is denoted as inv[X.op(arg by p_i)], and the second being the response event that occurs when the operation terminates with a result value res and is denoted as resp[X.op(res) by p_i]. We call an *event* a invocation event or a response event. An execution of a non-blocking concurrent algorithm involves the collection of invocation and response events related to the operation invocations on concurrent objects by the processes. A sequence of events is a history.

To specify the outcome of a history, the events in that history must be ordered in a way that the order respects the real-time order in which the events actually occur, is sequential and that order respects the sequential specification of each object involved in that history. If such a total order on the events exist then the history is said to be *linearizable* and the execution is *atomic*. This definition suggests that each operation executes at some indivisible instant between its invocation and response events which is called a *linearization point*.

Atomicity is the safety property for non-blocking concurrent algorithms. Liveness properties, also called *progress conditions*, for non-blocking concurrent algorithms include:

Obstruction-Freedom Obstruction-Freedom relates with each operation op() on some concurrent object X. An implementation of the operation op() is obstruction-free if it is guaranteed to terminate whenever it is executed in isolation.

Non-Blocking Non-blocking is similar to the deadlock-freedom property for blocking concurrent algorithms. A non-blocking implementation relates with the concurrent object X and not with its operations individually. An implementation of a concurrent object X is non-blocking if at least one operation invocation terminates whenever there are concurrent operation invocations pending on X. It follows that some operation invocation may fail to terminate if it always looses from other operation invocations that occur concurrently.

Wait-Freedom This is the strongest progress condition for non-blocking concurrent algorithms similarly with the starvation-freedom liveness property for blocking concurrent algorithms.Wait-freedom relates with each operation op() on some concurrent object X. An implementation of the operation op() is wait-free if it always terminates. An implementation of a concurrent object X is wait-free if the implementation of each of its operations is wait-free.

Architectural Support for Non-Blocking Synchronization

The basic building blocks for the implementation of non-blocking synchronization are *atomic variables*. An atomic variable can be accessed concurrently by multiple processes with load and store operations. All load and store operations on some atomic variable are totally ordered in such a way that (1) each operation appears to execute instantaneously between its invocation and response events, and (2) any read invocation returns the value written by the closest preceding write invocation. Nevertheless, atomic variables alone are not sufficient to implement any form of non-blocking synchronization possible.

The Computability Power of Concurrent Objects The synchronization power of concurrent objects is based on the notion of a *universal construction*. A concurrent object is a *universal object* if it can be used together with any number of atomic registers to wait-free implement any other concurrent object. Any wait-free algorithm implementing such a construction is called a *universal construction*. It has been shown that a *consensus object* is a universal object. A consensus object is a concurrent object that provides the operation *propose(v)*. A participating sequential process can invoke the propose(v) operation at most once. The processes invoke the operation propose(v) for the purpose of deciding a single value that has been proposed by one of the participating processes. More formally, a consensus object is defined by the following properties:

Validity A decided value is a proposed value

Integity A process decides at most once

Agreement No two processes decide different values

Termination An invocation of propose() by a correct process terminates.

In consequence, it suffices to question whether a concurrent object can wait-free implement a consensus object. To that end, with each concurrent object a *consensus number* is associated that gives the largest n such that, that concurrent object together with atomic variables can wait-free implement a consensus object for n processes. If there is no largest n then the consensus number is infinite. The consensus numbers for concurrent objects are used to compare the synchronization power of concurrent objects and a concurrent object X with a consensus number m is said to be less powerful than a concurrent object Y with a consensus number n with $m < n$ in the sense that concurrent objects of type X cannot be used together with atomic variables to wait-free implement a concurrent object of type Y but the opposite is trivially true. Using the consensus numbers a *consensus hierarchy* is defined with some concurrent atomic objects as shown in table 2.4.

Consensus number	Concurrent atomic objects
1	atomic variables, snapshot objects, ...
2	test&set, swap, fetch&add, FIFO queue, stack, ...
...	...
2m-2	m-register assignment (m>1)
...	...
∞	compare&swap, LL/SC, ...

Table 2.4: The consensus hierarchy

Providing that atomic variables have a consensus number equal to 1 and, hence, cannot be used to implement all non-blocking concurrent objects, the architecture must provide hardware primitives with a consensus number ≥ 2 and almost certainly a primitive with an infinite consensus number so that any number of processes can be supported. The most common such hardware primitives available in modern architectures are: *test&set*, *swap*, *compare&swap* and *LL/SC*. The sequential specification of these hardware primitives for a object X is specified in figure 2.3.

test&set	swap	fetch&add	compare&swap	LL/SC

- X is initialized to 1
- X.test&set() sets the value of X to 0 and returns its previous value
- X.reset() sets the value of X to 1.
	- X.swap(v) atomically stores v to X and returns the previous value of X	- X.fetch&add(v) atomically adds v to X and returns the previous value	- X.compare&swap(old new) is
			if (X == old) then
			X = new;
			return (true);
			else
			return (false);
			end if
				- X.LL(i) is a *linked load* operation by process i. This operation returns the current value of X and marks the process i as the current reader.
				- X.SC(i, v) writes v to X if no process has written X since process i's last invocation of X.LL()

Figure 2.3: Specialized hardware primitives to support non-blocking synchronization for n $\geq$ 2 processes.

In this thesis the fetch&add() and compare&swap() specialized hardware primitives are used. Hence, to illustrate the usage of the primitives figure 2.4 shows how to use the fetch&add() primitive to implement a consensus object for n = 2 processes and figure 2.5 shows how to use the compare&swap() primitive to implement a consensus object for any number of processes n. Note that the code is not formal C code. Also, the examples assume a sequential consistent memory model.

```
1  int REG[0...1];
2  int X = 0;
3
4  // Process pi has index 0 and process pj has index 1.
5  int propose(int i, int v)
6  {
7        REG[i] = v;
8        int prev = fetch&add(X, 1);
9        if (prev == 0)
10       {
11             // first to arrive
12             return REG[i]
13       }
14       else
15       {
16             // second to arrive
17             return REG[1−i]
18       }
19 }
```

Figure 2.4: A construction of a consensus object for n = 2 processes (pi and pj) using a fetch&add() object X. The first fetch&add() operation returns 0 and assigns 1 to X, whereas the second fetch&add() operation returns 1 and assigns 2 to X. The process that receives a return value of 0 knows that it is the first process to arrive. The value that is returned by the propose() operation is the value proposed by the process that arrived first.

```
1  int REG[1...n];
2  int X = −1;
3
4  // Processes have indices in the range [1...n]
5  int propose(int i, int v)
6  {
7        REG[i] = v;
8        bool succ = X.compare&swap(−1, i);
9
10       if (succ)
11       {
12             // first to arrive
13             return REG[i];
14       }
15       else
16       {
17             // X is the identity of the process that arrived first
18             return REG[X];
19       }
20 }
```

Figure 2.5: A construction of a consensus object for n processes using a compre&swap object X. The object X together with the compare&swap operation is used to determine which process arrives first. That process assigns its identity in X (the compare&swap() operations for the rest of the processes fail because the value of the variable X is no longer -1).

The ABA Problem A problem that is associated with the usage of the compare&swap() primitive is the so called ABA problem and any non-blocking concurrent algorithm must be designed with care so that the ABA problem is avoided whenever the compare&swap() primitive is used. The compare&swap() operation is often used in the following manner: first the process reads the value of the variable X (assume that it is A) , then it performs some operations updating some state and, lastly, publishes that new state by changing the value of X to some new value if X still equals A by performing a compare&swap operation. The intention of the process however is to publish the new state it has updated only if the state as existed when it first read the variable X remains unchanged. The problem, then, is that if the compare&swap() operation succeeds for the process it is not sufficient to conclude that the state of X hasn't changed because X could have been updated first to some other value B and then back to A before the process applies the compare&swap() operation. To solve the ABA problem a general solution is to append a sequence number to the variable X that is incremented with each update operation. X is now composed of two fields: (value, sequence-number) In the example above, the compare&swap() operation of the process would have failed because even if the value is the same the sequence-number has been changed by the sequence of update operations performed in between.

Performance implications of Specialized Hardware Instructions As far as performance is concerned, typically concurrent objects with smaller consensus number are faster in the hardware. That is, atomic variables are the fastest form of synchronization in hardware, and the test&set, swap and fetch&add hardware primitives are likely to be more efficient than the compare&swap primitive.

2.4 Concurrent Programming in the C Programming Language

The C programming language as of the C11 standard adds support for multi-threaded programming. It does so by providing a threading library with a well-defined memory model.

An execution of a multi-threaded program in C consists of multiple threads that perform *memory actions* on *memory locations*. A memory location is defined either as an object of scalar type or as a sequence of adjacent bit fields. The types of memory actions that a thread may perform are:

Data Operations Ordinary load and store operations on non-atomic variables

Synchronization Operations These include atomic stores, atomic loads and atomic read-modify-write operations that are performed on atomic variables. Other synchronization operations include lock and unlock operations as well as standalone memory fence operations.

A well-defined program must avoid *data races*. A data race occurs whenever two or more threads access the same memory location, at least one of those accesses is a store operation and that memory location is a non-atomic variable. To avoid data races, an ordering must be enforced between these accesses, which can be accomplished with synchronization operations.

The memory model specifies constraints on the relationships between memory actions in a execution. Intuitively, the programmer must manipulate appropriately these relationships between the memory actions so that a load operation reads the desired value. Consequently, the most important relation is the *happens-before* relation that constraints loads. To be more precise, a load operation has to read from a write to the same location that immediately precedes it in happens-before order. The *reads-from* relation maps a store operation to the load operations that read the value written by that store operation. Another important relation is the *modification order* that totally order all store operations to some memory location. The modification order must be consistent among threads. On the condition that the memory location is a atomic variable this is the default behaviour. On the other hand, for non-atomic variables the programmer is responsible for ensuring that all threads agree on the same modification order using synchronization operations.

In a single thread, the happens-before order is equivalent with the *sequenced-before* order that is a partial order on the memory operations performed by that thread in program order. This order is partial due to statements in C that have an undefined argument evaluation order. The rest of the memory model is concerned with how a pair of threads can relate to each other with happens-before relation so that one thread can obtain memory visibility of the other thread, meaning that it can load the values written by the other thread. To that end, the memory model offers various synchronization operations that are annotated with a *memory order* parameter that controls the amount of memory visibility obtained by some thread. The memory visibility is determined by how much synchronization and ordering is specified by a given memory order.

In brief, a suitable memory order annotation between two synchronization operations from different threads results in a *synchronizes-with* relation that extends to happens-before and, hence, memory visibility from the thread performing the first operation to the thread performing the second operation. The first operation must be a atomic store operation and the second operation a atomic load operation. The synchronizes-with relation is a transitive relation which means that it can be used to pass memory visibility between a chain of threads using pairwise synchronization operations.

The memory model places the following constraints to satisfy some *coherence* properties.

Coherent Load-Load Two reads ordered by happens-before may not read two writes that are modification ordered in the other direction

Coherent Store-Load It is forbidden to read from a write that is not the immediately preceding write in happens-before relation

Coherent Store-Store Happens-before and modification-order may not disagree

Coherent Load-Store The union of the reads-from map, happens-before and modification-order must be acyclic.

2.4.1 Synchronization Operations and Memory Orders

The synchronization operations are provided by the *stdatomic* and *threads* header files. These operations are performed on atomic variables and mutex (lock) objects. For the rest of this discussion, we are concerned only with atomic variables.

For a standard type T the stdatomic header file provides a corresponding atomic type that has the same name with a atomic_ prefix. As an example, atomic types include atomic_bool, atomic_int, atomic_uintptr_t and atomic_uintmax_t.

There are three types of operations can be be performed on atomic types: store, load and read-modify-write operations. Each operation receives an optional memory order argument from three memory models: *sequentially consistent*, *acquire-release* and *relaxed* memory model.

The memory orders for the available memory models are:

Sequential Consistency memory_order_seq_cst

Acquire-Release memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel

Relaxed memory_order_relaxed

The memory orders some operation can receive depending on its type are:

Store Operations memory_order_relaxed, memory_order_release, memory_order_seq_cst

Load Operations memory_order_relaxed, memory_order_consume, memory_order_acquire, memory_order_seq_cst

Read-Modify-Write memory_order_relaxed, memory_order_consume, memory_order_acquire, memory_order_release, memory_order_acq_rel, memory_order_seq_cst

Each operation *op* supports an implicit form named *op* and an explicit form with name *op_explicit*. The difference between these two forms is that the first form implicitly assumes a sequentially consistent memory order whereas the second form requires the memory order as a parameter.

Load Operation A atomic load operation on some atomic variable is performed with the *atomic_load* and *atomic_load_explicit* functions. Let *v* be an atomic variable of type *atomic_int*. Then, *atomic_load_explicit(&v, mo)* returns the value of *v* as a plain integer type with the specified memory order parameter.

Store Operation A atomic store operation on some atomic variable is performed with the *atomic_store* and *atomic_store_explicit* functions. Let *v* be an atomic variable of type *atomic_int*. Then, *atomic_store_explicit(&v, 1, mo)* writes 1 to *v* with the specified memory order parameter.

Read-Modify-Write Operations These operations correspond to the specialized hardware primitives offered by the architecture. The read-modify-write operations that will be used in the implementation of this work are: *atomic_fetch_add_explicit* and *atomic_compare_exchange_explicit*.

Let *v* be an atomic variable of type *atomic_int*. Then, *atomic_fetch_add_explicit(&v, 1, mo)* atomically adds the value 1 to v and returns the previous value of v with the specified memory order parameter. There is a corresponding *atomic_fetch_sub_explicit* function that instead subtracts the parameter from the atomic variable. Let *u* be a plain integer variable. Then, the *atomic_compare_exchange_explicit(&v, &u, 1, mo_success, mo_failure)* atomically compares the value contained in *v* with the value contained in *u* and if they are equal it updates *v* to 1 using *mo_success* as a memory order and returns true. Otherwise, it loads the value of *v* into *u* using *mo_failure* as a memory order and returns false. To be precise, the *atomic_compare_exchange_explicit* is not directly provided. Rather a *weak* and a *strong* version is offered with names *atomic_compare_exchange_weak_explicit* and *atomic_compare_exchange_strong_explicit*. Both support the functionality described earlier with the difference that the weak version is allowed to fail spuriously, that is to return false even if the two values are equal.

Sequential Consistent Ordering A sequentially consistent store synchronizes-with a sequentially consistent load of the same atomic variable that reads the value stored. In addition, any sequentially consistent atomic operations that are performed after that load must also appear after the store to other threads in the system using sequentially consistent atomic operations. Intuitively, this mean that the execution is a sequential consistent execution. Figure 2.6 shows an example in C using sequential consistent ordering. The assertion in thread B can never fire because the store by thread A to flag synchronizes-with the load from flag by thread B that reads the value written by that store, and, thus, that store happens-before that load. Then, since the store to variable x by thread A happens-before the store to variable flag, it follows that the store to variable x happens-before the load from flag by thread B that reads true. That load operation, in turn, happens-before the load of variable x inside the assertion, and, hence, the store to variable x from thread A happens-before the load from x by thread B and that last load is required to read the value from 1 because that store is the immediately preceding store in the happens-before relation.

```
int x;
atomic_bool flag;

// x is initialized to 0
// flag is intialized to false

Thread A:                    Thread B:
x = 1;                           while (!atomic_load(&flag)){}
atomic_store(&flag,              assert(x == 1);
          true);
```

Figure 2.6: Sequential Consistent Memory Ordering Example in C

Relaxed Ordering Operations on atomic types performed with relaxed ordering don't participate in synchronizes-with relationships. This means that relaxed operations cannot be used to ensure memory visibility from one thread to another. In the previous example if the two atomic operations were annotated with a memory_order_relaxed memory order, then the assertion could fire because the happens-before relationship between the store to flag by thread A and the load from flag from thread A that reads the value true does no longer exist.

Acquire-Release The acquire-release memory model allows to selectively insert necessary synchronization , that is synchronizes-with edges, between pairs of operations to ensure memory visibility. In this model, atomic load operations are *acquire* operations, atomic store operations are *release* operations, and atomic read-modify-write operations can be either *acquire*, *release* or both. In the simple form, a release operation tagged with *memory_order_release* synchronizes-

with a acquire operation tagged with *memory_order_acquire*. Hence, in the example of figure 2.6 if the store to variable flag by thread A is tagged as a release operation and the load from variable flag by thread B is tagged as an acquire operation then the assertion is guaranteed not to fire since the synchronizes-with edge exists. A operation tagged with *memory_order_seq_cst* participates in acquire-release synchronization as well.
The acquire-release model permits two more kinds of synchronization. The first is the so called *release sequence* synchronization and the second is the *consume ordering*.

Consume Ordering Consider the example of figure 2.7. Assume that the producer() function is executed by thread A and the consumer() function is executed by a different thread B. Assume, that acquire-release synchronization is applied. That is, mo_p2 is *memory_order_release* and mo_c1 is *memory_order_acquire*. Then, since all store operation by the producer thread happens-before the store to the variable p, and that store happens-before the load of p by the consumer thread that reads the value stored by the producer thread because the acquire-release synchronization introduces a synchronizes-with edge, then the stores by the producer thread happens-before the loads of the consumer threads and, hence, the assertions are guaranteed not to fire. This is even if the operations on the variable *a* are tagged with *memory_order_relaxed*. The consume model allows to selectively pass visibility of store operations before a release store depending on the data dependencies of that store operation. If, for example, mo_c1 is tagged as *memory_order_consume* then the store release by thread A is said to be *dependency-ordered-before* the load consume by thread B. In this way, thread B gains memory visibility of prior stored from thread A that have a dependency on the value loaded, which, in this case, is the variable data from thread A. Thus, the consumer thread is guaranteed to see the stores to member variables *x*, *y* and *z* at lines 13-15 and the assertions 27-29 will not fire. On the other hand, since the variable *a* is not dependent on the variable *data* the consumer thread doesn't see the store to variable *a* at line 18 and, for this reason, the assertion at line 30 may fire.

Release Sequence A synchronizes-with edge may be added between a release operation and a acquire operation even if more operation occur between them, if those operations either either store operations by the thread that performed the initial release operation or read-modify-write operations by any thread. The operations that are performed in between can have any memory ordering even *memory_order_relaxed*. The rationale behind the release sequence is to support the single-producer/multiple-consumer idiom without having to synchronize the consumers, but only the producer with each consumer individually. This is done as shown in figure 2.8. The producer() function is executed by a single producer thread A. The consumer() function is executed by multiple consumer threads. The purpose is for the producer thread first to populate a queue with integers and then publish the data inside the queue with a release operation on the count variable that holds the number of integers inserted. Now consider the first consumer thread that performs the fetch_sub operation on variable count. Since that operation is a acquire operation (it is tagged with *memory_order_acquire* then that consumer gains memory visibility of the

queue items. Now consider a second consumer. The second consumer performs a fetch_sub operation on count that *loads the value stored by the first consumer*. With pairwise synchronization in mind, one could change the *memory_order_acquire* of the fetch_sub operation to *memory_order_acq_rel*. In this way, the second consumer would gain memory visibility of the first consumer and since that first consumer has gained memory visibility of the producer, it follows that the second consumer also gains memory visibility of the producer. However, the second consumer doesn't need to gain memory visibility from the first consumer but only from the producer. The release sequence guarantees that this happens. The store performed by the producer is a store release and, hence, the head of the release sequence. The second consumer now that performs the acquire operation synchronizes-with the store release because the operation performed in between is a read-modify-write operation and, thus, part of the release sequence.

```
1  struct Data
2  {
3        int x, y, z;
4  };
5
6  atomic_uintptr_t p; // A pointer to a Data object initially NULL
7  atomic_int a;
8
9  void producer()
10 {
11       struct Data *data = (struct Data*)malloc(sizeof(struct Data));
12
13       data->x = 1;
14       data->y = 2;
15       data->z = 3;
16
17       atomic_store_explicit(&a, 99, memory_order_relaxed);
18       atomic_store_explicit(&p, (uintptr_t)data, mo_p2);
19 }
20
21 void consumer()
22 {
23       struct Data *data;
24
25       while (!(data = (struct Data*)atomic_load_explicit(&p, mo_c1))){}
26
27       assert(data->x == 1);
28       assert(data->y == 1);
29       assert(data->z == 3);
30       assert(atomic_load_explicit(&a, memory_order_relaxed) == 99);
31 }
32
```

Figure 2.7: A producer/consumer example in C

```
1  let queue_data be a queue of integers
2  atomic_int count;
3
4  void producer()
5  {
6        ... popule the queue of integers with num integers...
7
8        atomic_store_explicit(&count, num, memory_order_release);
9  }
10
11 void consumer()
12 {
13       while (true)
14       {
15             int item_index = fetch_sub_explicit(&count, 1, memory_order_acquire);
16
17             if (item_index <= 0){ continue; }
18
19             .. process element at index item_index - 1...
20       }
21 }
```

Figure 2.8: A single-producer/multiple-consumer example in C showing a release sequence

Standalone Memory Fences The same synchronization functionality provided by the memory order annotation of load, store and read-modify-write operations can be performed with standalone memory fences. A memory fence is added to the program using the *atomic_thread_fence(mo)*. For example, all non-atomic and relaxed atomic stores that happen before a *memory_order_release* fence in thread A will be synchronized with non-atomic and relaxed atomic loads from the same locations made in thread B after an *memory_order_acquire* fence.

2.5 Notes

The material for shared memory correctness (cache coherence and memory consistency) is heavily based on Sorin et al. [2011]. The section for the specialized hardware primitives is based on Raynal [2015]. The discussion of concurrent programming in the C programming language is based primarily on the book Williams [2012] which targets the C++ language. A discussion of the memory model for the C and C++ languages can be found in Batty et al. [2011] and Boehm and Adve [2008].

Chapter 3

The OpenMP 4.0 Task Dataflow Model

OpenMP 4.0 Task Dataflow Programming Model In the OpenMP 4.0 task dataflow programming model, the programmer can constraint the order of execution between *sibling* tasks, using the **depend** clause on the **task** directive. The purpose of the depend clause is to specify the memory footprint of a task by listing the memory objects that the task accesses, along with a memory usage annotation per memory object, that indicates whether that task reads-only, writes and reads but always writing before reading, or both reads and writes in any order, represented respectively by an **in**, **out** or **inout** annotation, that memory object. The memory objects listed in a depend clause are hereinafter referred to as *tags*. The syntax of the depend clause is **depend(** ***dependence-type* : *list*** **)**, where *dependence-type* is one of the memory usage annotations (in,out,inout) and *list* contains the tags. List items used in depend clauses of the same task or sibling tasks must indicate identical storage or disjoint storage.

3.1 The Task Graph

The child tasks generated by a parent task together with their memory usage annotations form a directed acyclic graph (DAG) named the *task graph*, where the child tasks constitute the nodes of this task graph and the edges represent the dependencies between tasks as derived from the memory usage annotations. The dependencies are formed as follows: A child task with an *in* memory usage annotation on some tag A, depends on all previously generated sibling tasks that reference that same tag A with an *out* or *inout* memory usage annotation. In other words, a task that reads A must be executed after preceding sibling tasks that are writing A have terminated. Furthermore, a child task with an *out* or *inout* memory usage annotation on some tag A, depends on all previously generated sibling tasks that reference that same tag A with an *in*, *out* or *inout* memory usage annotation. Put another way, a task that writes tag A may begin execution after any preceding sibling tasks that are reading or writing tag A terminate.

Overall, from the aforementioned discussion evidently one concludes that the following two rules are sufficient:

- A task with an *in* memory usage annotation on A, depends on the immediately preceding sibling task with an *out* or *inout* memory usage annotation on A.
- A task with an *out* or *inout* memory usage annotation on A, depends on the immediately preceding sibling task in case that task has an *out* or *inout* memory usage annotation on A. Otherwise, it depends on the set of sibling tasks with an *in* memory usage annotation on A immediately preceding it.

In view of the fact that a child task has dependencies only on preceding sibling tasks the task graph is acyclic meaning that a child task cannot depend on a sibling task that succeeds it in program order. This property suggests that the execution of a task graph cannot *deadlock*.

The roots of the task graph consist of tasks that are either executing or are ready to execute since they have no dependencies on preceding sibling tasks. The *ready list* provides access to the latter. Once a task is placed in the ready list it may be executed by any thread participating in the execution of the enclosing parallel region.

As a final point, notice that during the execution of an OpenMP program there may be multiple task graphs active at any point in time. Each task belongs to exactly one task graph, that generated by its parent task, and may also create its own task graph by generating child tasks.

Dependencies expressed in the task dataflow model Assume that a parent task T generates a sequence of m child tasks T_1, T_2, ..., T_m, each of which accesses the same tag A. The different types of dependencies that can be expressed between two sibling tasks T_i and T_j with $1 \leq i < j \leq m$, with respect to the tag A, are:

True Dependence Task T_i writes A and task T_j reads A. T_i provides the value needed by T_j and, thus, it has to be executed before T_j.

Anti Dependence Task T_i reads A and task T_j writes A. T_i must be executed before T_j because, otherwise, T_i would read the value of A after the update of T_j violating the sequential execution of those tasks.

Output Dependence Both tasks write A. The order of execution between T_i and T_j affects the final value of A. The sequential semantics suggest that T_i must execute prior to T_j.

Table 3.1 shows the types of dependencies between tasks T_i and T_j on tag A using the available memory usage annotations. A true dependence arises whenever the first task writes A (out, inout) and the second task reads A (in, inout). In addition, an anti dependence occurs when the first task reads A (in) and the second task writes A (out, inout). Lastly, an output dependence requires the first task writing A (out, inout) and the second task also writing A (out). Observe that this last case does not result in a true dependence, even though the out annotation permits the second task to read A, because the initial value of A is irrelevant for the second task.

T_j \ T_i	**in**	**out**	**inout**
in	*none*	*true*	*true*
out	*anti*	*output*	*output*
inout	*anti*	*true*	*true*

Table 3.1: Dependencies between two sibling tasks.

Figures 3.1, 3.2 and 3.3 show examples of true, anti and output dependencies, respectively, expressed in OpenMP.

```
1  #include <stdio.h>
2
3  int main()
4  {
5        int x = 1;
6
7        #pragma omp parallel
8        #pragma omp single
9        {
10             #pragma omp task shared(x) depend(out:x)
11             x = 2;
12
13             #pragma omp task shared(x) depend(in:x)
14             printf("x = %d“n". x);
15       }
16
17       return 0;
18 }
```

Figure 3.1: In this example, there exists a true dependence between the two generated tasks. Hence, the first task is executed before the second and the printf() statement is guaranteed to output the value 2 for x.

```
1  #include <stdio.h>
2
3  int main()
4  {
5        int x = 1;
6
7        #pragma omp parallel
8        #pragma omp single
9        {
10             #pragma omp task shared(x) depend(in:x)
11             printf("x = %d“n", x);
12
13             #pragma omp task shared(x) depend(out:x)
14             x = 2;
15       }
16
17       return 0;
18 }
```

Figure 3.2: In this example, there exists a anti dependence between the two generated tasks. Hence, the first task is executed before the second and the printf() statement is guaranteed to output the value 1 for x.

```
1  #include <stdio.h>
2
3  int main()
4  {
5        int x;
6
7        #pragma omp parallel
8        #pragma omp single
9        {
10             #pragma omp task shared(x) depend(out:x)
11             x = 1;
12
13             #pragma omp task shared(x) depend(out:x)
14             x = 2;
15
16             #pragma omp taskwait
17             printf("x = %d“n", x);
18       }
19
20       return 0;
21 }
```

Figure 3.3: In this example, there exists an output dependence between the two generated tasks. Hence, the first task executes before the second task. For this reason, the second task is the last task that writes to x and the printf statement is guaranteed to output the value 2 for x.

Renaming Anti and output dependencies can be trivially resolved, thus increasing parallelism, using a technique named *renaming*. Renaming works by introducing a new *version*, that is a copy, of the object whenever it is written to by a task and there exists some preceding task still referencing the object. If that preceding task reads the object an anti dependence is resolved, whereas if that preceding task writes the object an output dependence is resolved. Each new version is recorded as the current version of the tag for succeeding references. Table 3.2 repeats table 3.1 with renaming applied to the out memory usage annotation. Notice how the dependencies in the second row have all been resolved. Table 3.3 repeats table 3.1 when renaming is applied to both out and inout memory usage annotations. This last table contains only true dependencies.

Despite its potential as a technique especially for increasing parallelism, renaming cannot be generally implemented for OpenMP. More precisely, the specification explicitly states that a task with an *out* memory usage annotation on some tag A has to wait for all preceding sibling tasks with a reference to that same tag A.

Renaming is not implemented in this work. As a consequence, any *out* memory usage annotation can be replaced with an *inout* memory usage annotation.

T_j \ T_i	**in**	**out**	**inout**
in	*none*	*true*	*true*
out	*none*	*none*	*none*
inout	*anti*	*true*	*true*

Table 3.2: Dependencies between two sibling tasks when renaming is applied to *out* annotations.

T_j \ T_i	**in**	**out**	**inout**
in	*none*	*true*	*true*
out	*none*	*none*	*none*
inout	*none*	*true*	*true*

Table 3.3: Dependencies between two sibling tasks when renaming is applied to both *out* and *inout* annotations.

Operations on the Task Graph The task graph is managed using the following two operations:

Issue Operation The *issue* operation is called whenever a task is generated and has the effect of inserting the task in the task graph of its parent task linking it with all tasks it directly depends. If the task becomes a root of the corresponding task graph, it is placed directly in the ready list.

Release Operation The *release* operation is called by a task when it finishes execution. At first, the task removes itself from the task graph it belongs to. Then, it traverses the tasks that directly depended on it, and inserts each task that has now become a root of the task graph in the ready list of that task graph.

On the Parallel Nature of the Task Graph To express parallelism between the tasks in a task graph, we use the notion of a *generation*. A *generation* is a set of tasks that belong to the same task graph and may execute in parallel.The execution of a task graph is confined by the properties of generations:

Serialization between generations Generations must execute in program order.

Parallelism between generations The tasks inside a generation may execute in parallel.

Figure 3.4 illustrates the task graph for an example task sequence with the generations explicitly shown. The task sequence utilizes a single tag A for the memory usage annotations. Notice that the generations are numbered sequentially, starting with the *oldest generation* with number 0 and ending with the *youngest generation* that contains the task(s) most recently inserted in the task graph. To be precise, the oldest generation is not the generation with number 0 but the generation that is currently ready, meaning that it consists of tasks that are either in the ready list or are ready to execute with respect to that tag. For the example shown in the figure the generation with number 0 could be the oldest generation if task T_1 hasn't terminated yet.

How are generations formed? The conditions under which a new generation is created when a task is added to the task graph stem from table 3.1: the only case where there is no dependency between tasks is when those tasks have an *in* memory usage annotation on the tag. As a result, if the youngest generation consists of tasks with an *in* memory usage annotation on the tag and the new task also has a *in* memory usage annotation, then the new task can be executed in parallel with the tasks currently in the youngest generation and, that being the case, the new task is inserted in the currently youngest generation. Otherwise, the new task has a dependency on the tasks inside the youngest generation and, as a consequence, a new generation is created for the new task.

The above condition indicates that a generation consists either of a single task that may write to the tag (with a *out* or *inout* memory usage annotation), or of one or more tasks that may only read the tag (with a *in* memory usage annotation). In the example task sequence of figure 3.4, the generations are formed

with the following reasoning: The first task T_1 is a writer task and thus is alone in its generation which is also the first generation with number 0. Then:

- When task T_2 is issued to the task graph, the current youngest generation is generation 0. Because the current generation doesn't consist of tasks with *in* memory usage annotation, task T_2 is added to a new generation with number 1.
- When task T_3 is issued to the task graph, the current youngest generation, which is generation 1, consists of a task with a *in* memory usage annotation. In addition, task T_3 also has a *in* memory usage annotation and, as a result, it is added to the current youngest generation.
- When task T_4 is issued to the task graph, the current youngest generation is generation 1. Even though this generation consists of tasks with *in* memory usage annotation on the tag, task T_4 has a *inout* memory usage annotation and, hence, cannot be executed in parallel with tasks T_2 and T_3. Consequently, task T_4 is inserted into a new generation with number 2.

The same reasoning applies to the rest of the tasks.

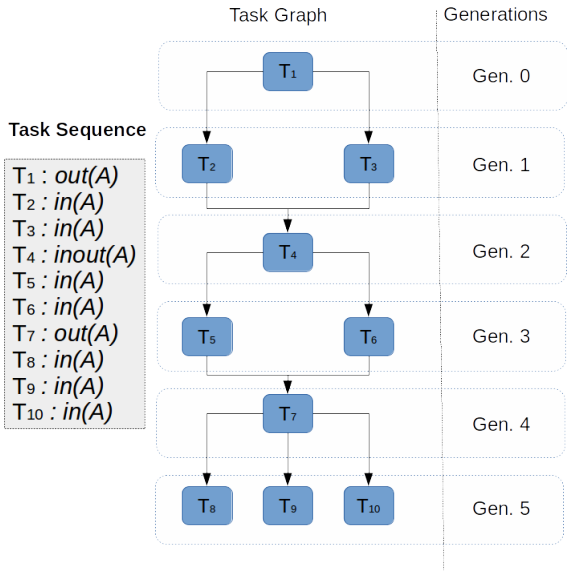

Figure 3.4: On the left, an example task sequence with memory usage annotations on tag A. On the middle the task graph that results from that task sequence illustrating the dependencies. An arrow from a source task to a destination task means that the source task must execute before the destination task. On the right, the generations for that task graph.

On the Concurrent Nature of the Issue and Release Operations Because a task graph exists per task, or, put another way, a parent task generates the task graph, it means that the *issue* operation on a task graph is executed sequentially by the parent task and, therefore, there are no concurrent *issue* operations on the same task graph. In contrast, a *release* operation may be concurrent with another *release* operation on the same task graph. As an illustration, consider a generation consisting of multiple tasks with *in* memory usage annotation on the same tag. Moreover, assume that these tasks access only this tag. The *parallelism between generations* property indicates that these tasks may execute in parallel, in which case they could terminate their execution simultaneously and, consequently, execute the *release* operation concurrently. Last but not least, an *issue* operation may be concurrent with multiple *release* operations for the simple reason that, when the *issue* operation is called for a task, preceding sibling tasks could finish execution.

3.2 Algorithms for Maintaining Dependencies at Runtime

In this section, i present two major classes of algorithms used for managing dependencies at runtime for cache-coherent shared-memory parallel systems. The two classes are:

The List Scheme The List Scheme is an *edge-centric* scheme. The characteristic of this edge-centric scheme is that it builds some form of the task graph at runtime, hence explicitly representing some of the edges resulting from the dependencies.

The Tickets Scheme The Tickets Scheme is an *edgeless* scheme. Contrary to the edge-centric scheme, the Tickets Scheme maintains conditions under which a task is ready to execute instead of building the task graph at runtime.

The algorithms require that the parent task maintains a dictionary from tags to *metadata* structures that are used for the purpose of dependence tracking. Henceforth, it is assumed that one tag can be used to express dependencies. Then i discuss how they can be extended to the general case where multiple tags can be listed in the depend clauses of the tasks.

3.2.1 The Tickets Scheme

In this work the Tickets Scheme has not been implemented and, for this reason, it is briefly described here.

Under the Tickets Scheme, all tasks belonging to a task graph are placed in the same *list*. The threads scan this list and perform a *ready-check* operation on each task. This operation consults the task's internal state as well as the metadata of the tag and indicates whether the task is ready for execution or not.

Internal representation of the metadata object To support this reasoning, the metadata for the tag has to include information on the number of reader and writer tasks that have been generated and how many of them have terminated. The solution counts tasks with an *in* memory usage annotation as *reader-only* tasks, and tasks with an *out* or *inout* memory usage annotation as *writer* tasks. The variables readers_generated and writers_generated count, respectively, the number of reader and writer tasks that have been issued. The variables readers_terminated and writers_terminated count, respectively, the number of reader and writer tasks that have finished.

Description of the issue operation. The first action to be taken is to determine the conditions under which the task can be scheduled for execution. To begin with, observe that since the issue operation is executed sequentially by the parent task, the metadata member variables readers_generated and writers_generated store the number of reader and writer tasks that have been generated thus far, respectively. For a reader-only task, the issue operation copies in a variable local to the task named w the value of the writers_generated member variable. The value of the variable w now denotes the number of writer tasks that the reader-only task has to wait for. For a writer task, both the writers_generated and readers_generated member variables are copied into local to the task variables named w and r respectively, because the task has to wait for all reader tasks, whose number is stored in r, and writer tasks, whose number is stored in w, to finish. Then the issue operation accounts for the generated task by incrementing the readers_generated counter in case of a reader-only task, and the writers_generated counter for a writer task. It is important that these two actions are performed in this exact order; otherwise, a writer task, for example, would wait for itself to finish, thus leading to deadlock.

Description of the release operation. The release operation is rather trivial. Each task increments the counter for terminated tasks that corresponds to its class. In more detail, a reader-only task increments the readers_terminated counter, whereas a writer task increments the writers_terminated counter.

Description of the ready-check operation. The property of serialization between generations indicates that the local variable w of a reader-only task, and the local variable r of a writer task, counts the writer tasks and reader-only tasks, respectively, that have to be executed first. Therefore, the ready-check operation for a reader-only task is $writers\_terminated == w$, and for a writer task is $(writers\_terminated == w) \wedge (readers\_terminated == r)$

3.2.2 The List Scheme

To put it briefly, the issue operation for the List Scheme inserts the task in the task graph and the release operation removes the task from the task graph. The List Scheme requires the task descriptor to contain a *dependency counter* named *depend_count*, which stores the number of dependencies that must be resolved before the task becomes ready to execute.

Internal representation of the task graph. The List Scheme represents the task graph as a list of tasks. Figure 3.5 shows the list representation of the task graph for the example task sequence of figure 3.4. The generations of interest to the List Scheme are (1) the *oldest generation* which consists of the tasks that are either executing or are ready to execute, and (2) the *youngest generation* with the tasks most recently added to the task graph. In detail, the List Scheme manages the following information:

Information needed for the oldest generation The number of tasks that are still executing or ready to execute. The last task that terminates from the oldest generation must visit the tasks of the next generation and update their dependency counters.

Information needed for the youngest generation The memory usage annotation used by tasks belonging to the youngest generation. That annotation is used as a condition to test if the insertion of the next task to the task graph entails the creation of a new generation.

In order to delimit generations inside the list, the List Scheme inserts an *end-of-generation marker* to each node of the list named *last_in_generation*. This flag is set for each node that stores that stores a pointer to the last task, according to their insertion order in the task graph, of some generation. In addition, observe in the example of figure 3.5 that the oldest generation is not explicitly represented in the list.

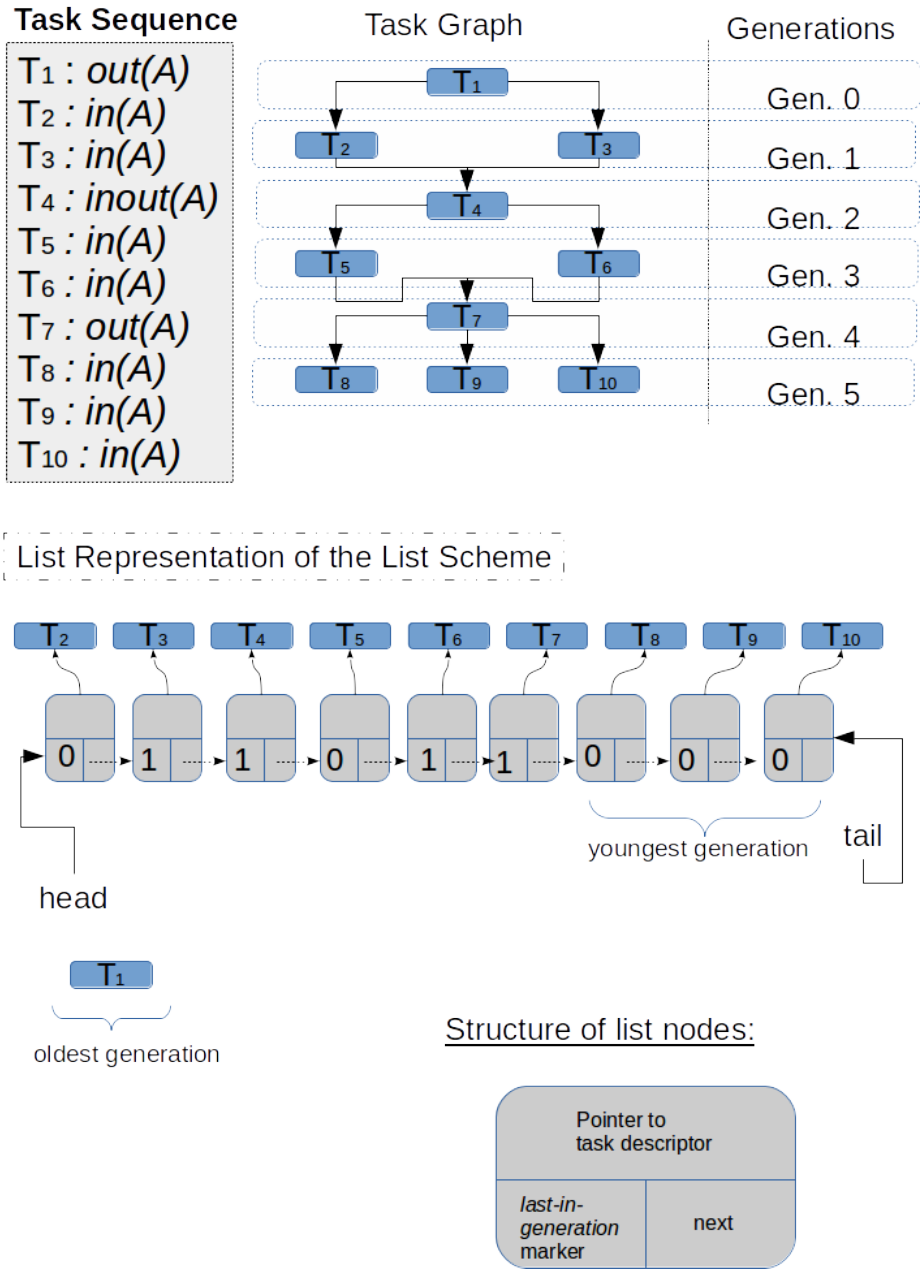

Figure 3.5: The list representation of the task graph for the List Scheme.

Internal representation of the metadata structure. The data structures managed by the List Scheme are shown in figure 3.6. The metadata associated with the tag is of type struct metadata. The head and tail pointers delimit the list of tasks representing the task graph. Their initial value is NULL. In case both head and tail are non-NULL pointers, then the head pointer points to the first task of the generation following the oldest generation and the tail pointer points to the last task of the youngest generation. On the other hand, given that both head and tail pointers are NULL pointers, either there are no generations or there is a single generation which is not represented explicitly in the list. These two cases can be distinguished by the value of the num_gens member variable, that counts the number of generations that have been issued and not yet terminated. The num_gens member variable is initialized with zero. Moreover, the member variable oldest_num_tasks stores the number of tasks belonging to the oldest generation which are either executing or ready to execute and is initialized with zero. In addition, the variable youngest_annot of type enum Annot_Type indicates the the memory usage annotation of the tasks that constitute the youngest annotation. Its initial value is Annot_Invalid. For the simple reason that out memory usage annotations are replaced with inout memory usage annotations, no corresponding enumeration value is provided for out memory usage annotations. The purpose of the youngest_annot variable is to determine whether to create a new generation upon task addition or not. Finally, lock is a mutex object that is used to protect the shared metadata object from being simultaneously accessed by concurrent issue and release operations.

```
1  struct task_descriptor
2  {
3        // ... other task-related info ...
4        mutex lock;
5        int depend_count;
6  };
7
8  struct list_node
9  {
10       struct task_descriptor *task;
11       int last_in_generation;
12       struct list_node *next;
13 };
14
15 enum Annot_Type
16 {
17       Annot_Invalid, Annot_In, Annot_Inout
18 };
19
20 struct metadata
21 {
22       struct list_node *head;
23       struct list_node *tail;
24       int num_gens;
25       enum Annot_Type youngest_annot;
26       int oldest_num_tasks;
27       mutex lock;
28 };
```

Figure 3.6: The data structures for the List Scheme.

The algorithm implementing the issue operation. The algorithm implementing the issue operation is listed in figure 3.7. The issue operation receives the following arguments:

struct task_descriptor *task A pointer to the task that is inserted into the task graph.

struct metadata *md a pointer to the metadata structure associated with the tag listed in the task's depend clause.

enum Annot_Type annot The memory usage annotation used in the depend clause of the task for the tag.

struct list_node *node A pointer to a struct list_node node to use in order to insert the task into the list inside the tag's metadata structure.

The issue operation returns the value 1 if it determines that the task has a dependency on some previous generation of tasks. Otherwise, it returns zero. As fas as the *md* parameter is concerned, the parent task upon creation of the child task that is inserted into the task graph, searches the dictionary it maintains that maps tags to metadata structures, in order to locate the metadata associated with the tag listed in the task's depend clause.

To begin with, the issue operation initializes a local variable has_deps, which indicates whether the task has a dependency on some previous generation or not, to zero (line 4), and, then, initializes the node to insert into the list (lines 6-8). Next, in line 10 the issue operation acquires the mutex variable lock to ensure mutual exclusion for the rest of its operation. Afterwards, there are three cases to consider:

Case 1 (lines 12-17) In the event that there are no generations, or, equivalently, when the num_gens variable of the metadata structure equals zero, then the task forms the only active generation which is both the oldest and the youngest generation. That being the case, the node is not inserted into the list, since the oldest generation is never explicitly represented in the list, and, due to the fact that there doesn't exist a previous generation, the task has no dependency on previous generations and, consequently, the depend_count of the task is not incremented. The variable num_gens is set to 1 in line 14 to account for the new generation. As this generation is the oldest generation, the oldest_num_tasks variable is set to 1 in line 15 to account for the new task. Lastly, this generation is also the youngest generation and, hence, the annot argument is saved in the youngest_annot variable in line 16.

Case 2 (lines 18-38) This case tests whether the task creates a new generation or not. Note that there exists at least one generation in the list, since the test at line 12 of the algorithm failed. The condition $(annot == Annot\_Inout \vee md \rightarrow youngest\_annot! = annot)$ indicates that the task must be inserted to a new generation, due to the fact that this condition is equal to the complement of the condition $(annot == Annot\_In \wedge md \rightarrow youngest\_annot == annot)$ which represents the conditions under which the task is inserted in the current youngest generation

(see paragraph On the Parallel Nature of the Task Graph for more details). In consequence, this case handles the addition of the task to a new generation, whereas the next case (Case 3) handles the addition of the task to the current youngest generation. Line 20 remembers the annotation for the new generation and line 21 increments the num_gens variable to account for the new generation. Also, the task has a dependency on the previous generation and, as a result, its dependency counter is incremented by one in line 24 and the has_deps local variable is set to 1 in line 37. Lines 27-36 insert the task at the end of the list. If the previous generation is not the oldest generation then it is explicitly stored in the list and, hence, the last_in_generation flag of the last task inside the previous generation which is pointed by definition by the tail pointer must be set to 1 in line 29 of the algorithm.

Case 3 (lines 39-55) When the task is inserted immediately into the youngest generation, the variables num_gens and youngest_annot remain unchanged because no new generation is created. The subtle point here is to test if the youngest generation is the only generation, that is if the youngest generation is also the oldest generation. In that case (lines 41-44), the task is added into the oldest generation and, as a result, the node is not inserted into the list. Moreover, the depend_count variable of the task is not incremented because the task has no dependency on previous generations. However, the oldest_num_tasks is incremented by one in line 43 to account for the addition of the task into the oldest generation. On the other hand (lines 45-54), the task is not added to the oldest generation. The node is inserted at the end of the list in lines 51-52. Moreover, the dependency counter of the task is incremented by one in line 48 and the has_deps local variable is set to 1 in line 53 to account for the dependency of the task on the previous generation. Note that the last_in_generation flag is not updated because the task is inserted into the current youngest generation and not in a new one.

At last, the mutex variable lock is released in line 57 and the has_deps local variable is returned in line 59 as the return status of the issue operation.

```
1  int issue(struct task_descriptor *task, struct metadata *md,
2            enum Annot_Type annot, struct list_node *node)
3  {
4        int has_deps = 0;
5
6        node->task = task;
7        node-> last_in_generation = 0;
8        node->next = NULL;
9
10       acquire_mutex(&md->lock);
11
12       if (md->num_gens == 0)
13       {
14             md->num_gens = 1;
15             md->oldest_num_tasks = 1;
16             md->youngest_annot = annot;
17       }
18       else if (annot == Annot_Inout || md->youngest_annot != annot)
19       {
20             md->youngest_annot = annot;
21             ++md->num_gens;
22
23             acquire_mutex(&task->lock);
24             ++tak->depend_count;
25             release_mutex(&task->lock);
26
27             if (md->tail != NULL)
28             {
29                   md->tail->last_in_generation = 1;
30                   md->tail->next = node;
31             }
32             else
33             {
34                   md->head = node;
35             }
36             md->tail = node;
37             has_deps = 1;
38       }
39       else
40       {
41             if (md->num_gens == 1)
42             {
43                   ++md->oldest_num_tasks;
44             }
45             else
46             {
47                   acquire_mutex(&task->lock);
48                   ++tak->depend_count;
49                   release_mutex(&task->lock);
50
51                   md->tail->next = node;
52                   md->tail = node;
53                   has_deps = 1;
54             }
55       }
56
57       release_mutex(&md->lock);
58
59       return has_deps;
60 }
```

Figure 3.7: The issue operation for the List Scheme

The algorithm implementing the release operation. The algorithm appears in figure 3.8. To begin with, the mutex variable lock is acquired in line 3. Secondly, as the task finished its execution and it belonged to the oldest generation, the oldest_num_tasks variable is decremented by one in line 5. If the task was the last task still executing among the tasks in the oldest generation the test in line 5 of the algorithm succeeds. In that case (lines 7-34), the release operation decrements the num_gens variable in line 29 to account for the termination of the oldest generation and, also, updates the dependencies for the next generation in lines 7-27. Since the oldest generation is not explicitly represented in the list, the next generation, if it exists, is pointed to by the head pointer. Hence, the release operation traverses the next generation starting with the head pointer in the first iteration of the loop in lines 10-26. For each task encountered, its dependency counter is decremented by one in line 15 and it is marked for insertion into the ready list of the task graph in line 17 if the task is now ready to execute. These tasks are part of the oldest generation now and, thus, the oldest_num_tasks variable is incremented by one in line 21 for each such task. Note the list node will not be referenced again and can be reclaimed (line 24). The end of the next generation is determined by the last_in_generation flag inside the nodes of the list in line 26, or when there is no next node. If the next generation traversed in lines 7-27 was the youngest generation, the list becomes empty (line 31), and, thus, the tail pointer is reset to NULL (line 33). In the end, the mutex variable lock is released in line 37.

```
1  void release(struct task_descriptor *task, struct metadata *md)
2  {
3       acquire_mutex(&md->lock);
4
5       if (--md->oldest_num_tasks == 0)
6       {
7            if (md->head != NULL)
8            {
9                 struct list_node *t, *r;
10                do
11                {
12                     t = md->head; r = t;
13
14                     acquire_mutex(&t->task->lock);
15                     if (--t->task->depend_count == 0)
16                     {
17                          // flag t->task for insertion into ready-list
18                     }
19                     release_mutex(&t->task->lock);
20
21                     ++md->oldest_num_tasks;
22                     md->head = t->next;
23
24                     // reclaim r
25
26                } while (t->last_in_generation == 0 && md->head != NULL);
27           }
28
29           --md->num_gens;
30
31           if (md->head == NULL)
32           {
33                md->tail = NULL;
34           }
35      }
36
37      release_mutex(&md->lock);
38 }
```

Figure 3.8: The release operation for the List Scheme

Time Complexity. A simplistic analysis follows. Assume that a generation of M tasks is followed by a generation of N tasks. Note that either M or N must be 1 because there cannot exist two successive generations consisting of tasks with *in* memory usage annotation on the tag.

Time Complexity of the issue operation The cost of the issue operation is O(1) per task, because the issue operation performs a constant number of updates on the metadata structure. Thus, the cost of the issue operation for the two generations is O(M+N)= O(max(M,N)) considering that either M or N is 1.

Time Complexity of the release operation The first M-1 tasks of the first generation have an O(1) cost since they only decrement the oldest_num_tasks variable. The last task of the first generation, though, has a O(N) cost since it must traverse the tasks of the next generation which are N in number. The cost of the release operation for the first generation is, thus, O(M+N). As far as the second generation is concerned, there is an O(1) cost per task and, hence, O(N) in total. As a result, the cost of the release operation for the two generations is O(M+N) = O(max(M,N)) considering that either M or N is 1.

Space Complexity. For the same scenario used in the analysis of the time complexity, the space complexity for the List Scheme is O(M+N), that is linear in the number of tasks, since one metadata structure and M+N nodes of constant size are used. If the first generation with M tasks is the oldest generation, the List Scheme uses N nodes in the list instead and, consequently, the space complexity is O(N).

A starvation-free implementation. Assuming that the parent task creates a finite number of child tasks, the implementation of the *issue* and *release* operations is starvation-free even if the mutex variable lock of the metadata structure is deadlock-free and not starvation-free. Consider the issue operation for some child task. Since there is a bounded number of preceding sibling tasks, the issue operation can fail to acquire the mutex lock only due to a bounded number of concurrent release operations. Similarly, a release operation at the worst case will fail to acquire the mutex lock against all the issue operations and the rest of the release operations for the same generation, which are bounded in number.

The case of multiple tags. The parent task maintains a dictionary from tags to metadata structures. In case a child task lists multiple tags in its depend clauses, the parent task calls the issue operation for each tag. The task is inserted to the ready list if all issue operations return a status value of zero. Moreover, the task descriptor is augmented with a list of pointers to the metadata structures associated with the tags. When the task terminates its execution it calls the release operation for each such metadata structure. Furthermore, as it is done in the algorithms in figures 3.7 and 3.8, accesses to the dependency counter depend_count of a task are pretected by a mutex variable lock in the task descriptor. This mutex is needed because, for example, when the issue operation is called for a task T_1 with some tag A and that issue operation increments

the dependency counter of T_1, even if no release operation can decrement the dependency counter of T_1 with respect to the tag A due to the mutual exclusion between the issue and the release operations on that tag, a release operation for another tag B that was listed in the depend clauses of task T_1 may execute concurrently with the issue operation since that release operation operates on another metadata structure. To make the above reasoning correct there is a subtle point to consider as illustrated in figure 3.9. There are two solutions to this problem. Either the parent task must initialize the depend_count variable of the task to 1 and decrement it by one after all issue operations are performed, thus ensuring that in between no release operation can decrement that dependency counter to zero, or the parent task must acquire the mutex variable lock of the task before performing the issue operations and release it afterwards, thus ensuring that no concurrent release operations updates the dependency counter of the task in between.

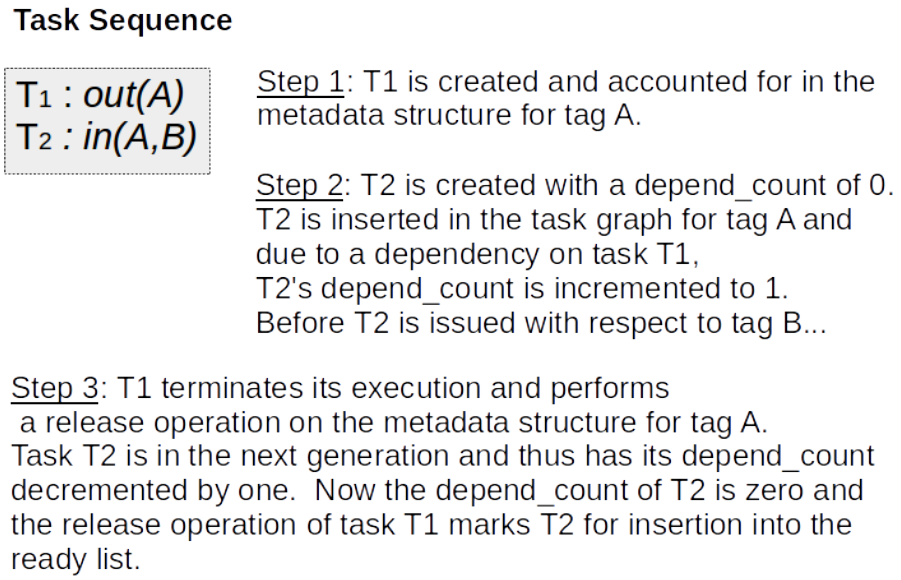

Figure 3.9: The problem with the dependency counter of a task in the general case of multiple tags.

3.3 Notes

The theory behind the task graph on this chapter and the description of the algorithms for maintaining dependencies at runtime are based on [Vandierendonck et al., 2013].

Chapter 4

A Lock-Free List Scheme

In this section i describe a lock-free version of the list scheme presented in the previous chapter that i have developed for my thesis. The algorithms presented assume a sequential consistent memory model. Relaxing the memory orderings for the algorithms is a future work.

4.1 Internal Representation of the task graph

Figure 4.1 shows the list representation of the task graph for the example task sequence of figure 3.4. The task graph is represented as a list of nodes. The list starts with a *sentinel* node since it makes concurrent manipulation more convenient. Moreover, notice that all generations are explicitly represented in the list. Each node of the list contains (1) a pointer to the task, (2) the number of the generation the task belongs to (notice that since the sentinel node has a generation number of 0 the tasks start with a generation number of 1 contrary to the lock-based list scheme of the previous chapter), (3) a boolean flag indicating if the task associated with the node is a reader-only task, (4) a pointer to the metadata structure, (5) a pointer to the previous node in the list, and, finally, (6) a pointer to the next node in the list.

The metadata object contains the following information: (1) head and tail pointers that delimit, respectively, the beginning and end of the list (note that the head pointer always points to a sentinel node), (2) the number of generations that have been added to the task graph by the issue operation (variable num_gens_issued), and, finally, (3) a flag that indicates whether the youngest generation consists of task with an *in* memory usage annotation or not. Moreover, the task descriptor is augmented with a dependency counter, similarly to the lock-based list scheme of the previous chapter, and an array of pointers to the nodes used for the task in the metadata structures associated with the tags in the depend clauses of that task. These additions are necessary to deal with the general case of multiple tags.

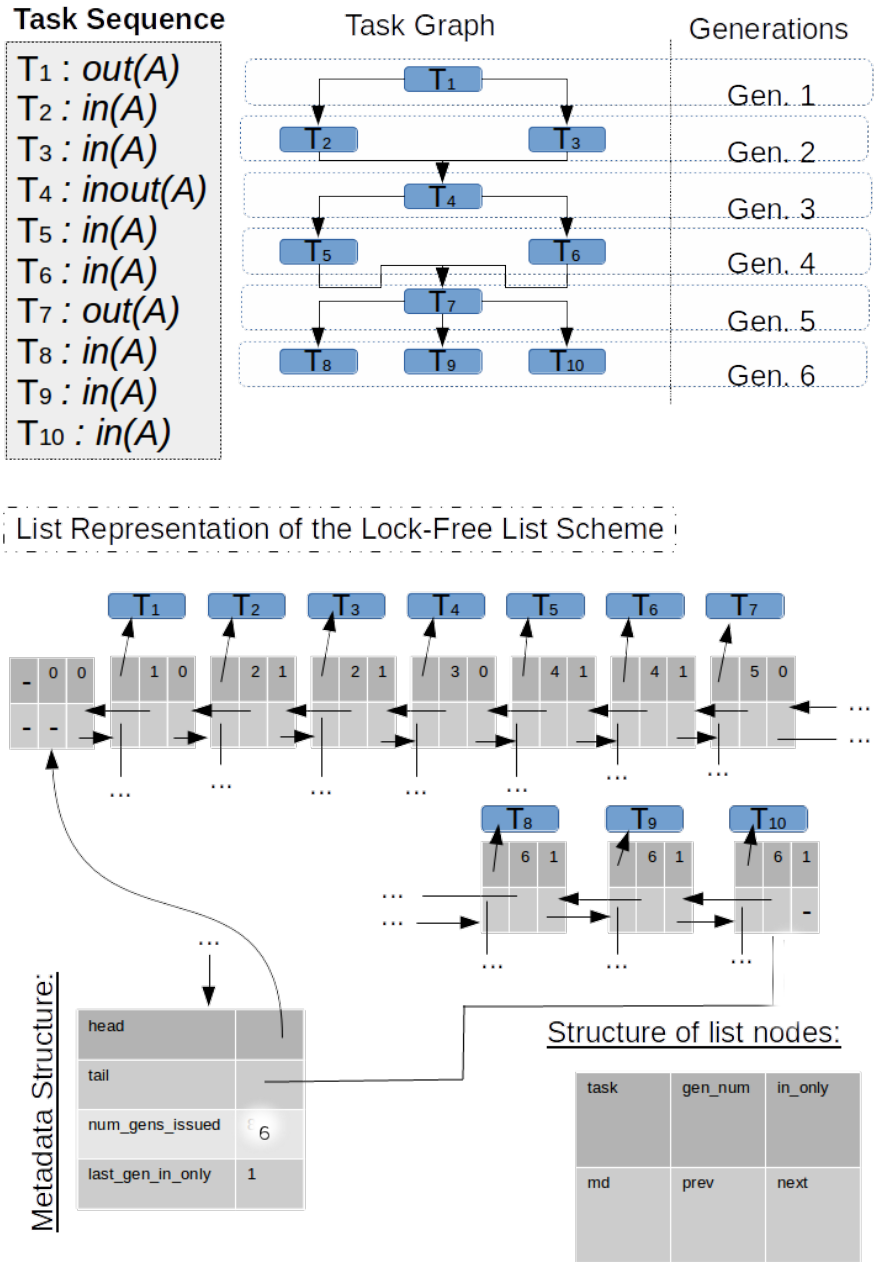

Figure 4.1: The list representation of the task graph for the Lock-Free List Scheme.

```
1  typedef enum task_depend_annotation
2  {
3        task_depend_annotation_output,
4        task_depend_annotation_input,
5        task_depend_annotation_inout
6  } task_depend_annotation_t;
7
8  typedef struct task_depend_metadata
9  {
10       _Alignas(CACHE_LINE) atomic_uintptr_t tail;
11       uint_fast64_t num_gens_issued;
12       bool last_gen_in_only;
13       _Alignas(CACHE_LINE) atomic_uintptr_t head;
14 } task_depend_metadata_t;
15
16 typedef struct task_depend_node
17 {
18       _Alignas(CACHE_LINE) ort_task_node_t *task;
19       task_depend_metadata_t *md;
20       struct task_depend_node *prev;
21       uint_fast64_t gen_num;
22       bool in_only;
23       _Alignas(CACHE_LINE) atomic_uintptr_t next;
24 } task_depend_node_t;
25
26 typedef struct ort_task_node
27 {
28       _Alignas(CACHE_LINE) atomic_uint_fast64_t depend_count;
29       _Alignas(CACHE_LINE) task_depend_node_t **which_dependencies;
30       uint_fast64_t which_dependencies_count;
31 } ort_task_node_t;
```

Figure 4.2: Data structures used for the lock-free list scheme

.

Figure 4.2 details the data structures employed by the lock-free list scheme. An explanation follows:

task_depend_annotation_t A enumeration type for the memory usage annotation.

task_depend_metadata_t The metadata structure. The *tail* and *head* pointers may be concurrently accessed by both the issue and a release operation and, hence, are declared as atomic variables (the type atomic_uintptr_t is capable of holding a pointer value). Moreover, to avoid false-sharing they are cache-aligned. The num_gens_issued and last_gen_in_only variables are only updated by the issue operations and, thus, are declared as normal variables.

task_depend_node_t Except from the *next* pointer all the other fields are updated by the issue operation and later read by release operations. As a result, they are declared as normal variables. The *next* pointer is declared as a atomic variable because it may be concurrently accessed by the issue and a release operation. To avoid false-sharing the *next* pointer is cache-aligned.

ort_task_node_t The task descriptor is augmented with the dependency counter which is a atomic variable since in the general case of multiple tags multiple release operations may concurrently update the dependency counter of some task. The which_dependencies and which_dependencies_count variables are updated by the issue operations and later read by the task itself when it performs release operation and, hence, they are declared as normal variables.

The special format of the tail pointer The tail pointer of the metadata structure is special to this algorithm in the sense that it consists of two fields. The first field is the pointer value and the second field is a status indication. The possible status values are the following:

none The tail pointer points to the sentinel node. In other words, the task graph is empty and contains no generations.

oldest The tail pointer points to a node associated with a task that belongs to the oldest generation.

youngest The tail pointer points to a node associated with a task that doesn't belong to the oldest generation (the current youngest generation is not the oldest).

These two fields are represented in the tail pointer by stealing two bits from the least-significant-bits of the pointer value which are enough to represent three states. This is possible because memory allocators can be requested to return pointer values that comply with certain alignment restrictions. Thus, these two fields are packed into a single atomic variable and the indent is to emulate a *double compare-and-swap* operation using a single compare-and-swap operation. Alternatives would be either to require the presence of a *double compare-and-swap* primitive which is not portable among architectures, or to implement a software version of the *double compare-and-swap* primitive from multiple single *compare-and-swap* operations which would result in a less efficient solution. In the code presented afterwards, the status values are denoted as none_tail_status, oldest_tail_status and youngest_tail_status. Moreover, a macro EXTRACT_PTR_FROM_TAIL(tail) returns the pointer field of the parameter and a macro EXTRACT_STATUS_FROM_TAIL(tail) returns the status field of the parameter. Last but not least, a macro MAKE_TAIL(node, status) packs the pointer value node and the status value into a single value to be stored to the tail pointer.

4.2 The algorithm implementing the top-level issue operation

Figure 4.3 shows the top level issue operation that is used in the general case of multiple tags. The function issue_task takes as arguments the task descriptor, an array of tags (each tag is a void * pointer), and the number of dependencies for each memory usage annotation. First, the operation updates the dependency counter of the task to the number of tags in line 6. This ensures that while the operation is in the process of issuing the task to each tag separately, concurrent release operations (from the tags it has already been issued to) cannot decrease the dependency counter to zero, unless of course all the issue operations have completed. The issue operations for each tag are performed in the while loop at lines 8-12. In the variable num_true_dependencies the operation keeps track of the number of times the issue operation at line 9 returns 1, meaning that for that tag the task has a dependency. After all the issue operation have been completed, the variable surplus in line 14 indicates the number of tags for which the task didn't had a dependency. Since the dependency counter is initialized with the total number of tags, the surplus in case it is non-zero must be subtracted from the dependency counter as it is done in line 16. If that atomic_fetch_sub operation decrements the dependency counter to zero the task has no dependencies and must be inserted in to the ready list.

```
1  void issue_task(ort_task_node_t *tnode, void **dep_array, int num_output, int num_input, int num_inout)
2  {
3        int num_dependencies = num_output + num_input + num_inout;
4        int num_true_dependencies = 0;
5
6        atomic_store(&tnode->depend_count, num_dependencies);
7
8        for (int i = 0; i < num_dependencies; ++i){
9              int dep = issue(get_metadata(dep_array[i]), GET_ANNOTATION_TYPE(i, num_output, num_input, num_inout), tnode);
10
11             num_true_dependencies += dep;
12       }
13
14       int surplus = num_dependencies - num_true_dependencies;
15
16       if (surplus > 0 && atomic_fetch_sub(&tnode->depend_count, surplus) == surplus){
17             ... insert task into ready list ...
18       }
19 }
```

Figure 4.3: The algorithm implementing the top-level issue operation with a sequential consistent memory model

4.3 The algorithm implementing the issue operation

Figure 4.4 shows the issue operation for a single tag. The operation receives as arguments (1) the metadata structure associated with the tag, (2) the memory usage annotation and (3) the task. To begin with, the issue operation allocates a node to use for insertion into the list in line 3. Line 7 tests whether the task is inserted in the current youngest generation or in a new one. In the latter case, the num_gens_issued variable of the metadata structure is incremented by one to account for the new generation, the last_gen_in_only variable is updated to indicate if the new generation consists of task with a *in* memory usage annotation, and since the task is being added to a new generation it is the first in this generation and, hence, the first_in_gen variable is set to true (lines 8-10). Next, the issue operation initializes the node with the task and metadata (lines 13-14), the generation number of the task which simply equals the number of generations issued (line 15), whether the task is a reader-only task (line 16) and records the node in the which_dependencies table of the task (line 17). After this initialization is performed, the node is inserted at the end of the list. For this to happen, the next pointer of the node is set to NULL (line 18), the next pointer of the node currently pointed to by the tail pointer is set to the new node and the prev pointer of the new node is set to that node (lines 24-25). The last step is to update the tail pointer to point to the new node. In this last step, the issue operation takes into account the current status of the tail pointer to determine the new status of the tail pointer and whether the task has a dependency or not. There are three cases to consider depending on the current status of the tail pointer which is retrieved at line 22.

Case 1 (lines 29-39) In this case the tail status is none. This means that there are no generations in this list and, hence, the task has no dependency. For this reason, in this case, the issue operation returns 0 in line 39. Normally, the issue operation would have to update the tail pointer to point to the newly inserted node with a simple store operation as it is done in line 37. Notice that a compare-and-swap operation is not necessary because the task graph is empty and no release operation can occur. Also, the new status of the tail pointer is set to oldest (line 37) since the node is part of the oldest generation. However, due to a special case in the release operation whenever the tail status is none the state of the list may not be correct, meaning that normally the head would have to point to the same node as the tail pointer and that node being the sentinel node, but in that special case this is not true. For this reason, the issue operation reads the head pointer at line 31 and if that happens, reclaims the nodes prior to the sentinel node (line 32), marks the prev pointer of the sentinel node to NULL since it is the first node of the list now (line 33) and sets the head pointer to the sentinel node (line 34). After these actions the state of the list has been corrected.

Case 2 (lines 41-48) In this case the tail status is oldest. Hence, if the task is inserted in the current youngest list (first_in_gen is false) the tail status remains oldest and if the task is inserted in a new generation (first_in_gen is true) the tail status must become youngest (line 41). Since there exists a generation in the list, a release operation may arrive and hence the issue operation must update the tail pointer with a compare-and-swap operation. This is done in line 43. The tail pointer is updated to point to the new node with a new status as computed in line 41. If that operation succeeds (line 45) then the issue operation returns the variable first_in_gen which indicates if the task had a dependency or not. That is, because if that value was false then the task has been inserted into the oldest generation and, hence, has no dependency on prior generations. On the other hand, if that value was true then the task has been inserted into a new generation and, hence, it has a dependency on the current oldest generation which we know hasn't terminated yet due to the compare-and-swap operation. If the compare-and-swap operation fails this can only happen due to a release operation. Moreover, since the status of the tail was oldest that release operation can only have changed it to none. The issue operation, then, retries in the while loop. Since the local_tail_status variable is updated at line 48 at the next iteration of the while loop the issue operation will enter case 1.

Case 3 (lines 50-55) In this case the tail status is youngest. Regardless of the value of the first_in_gen variable the status of the tail variable will remain youngest. Also, a release operation may arrive and update the tail pointer concurrently with the issue operation and, for that reason, the issue operation updates the tail pointer with a compare-and-swap operation (line 50). If that operation succeeds, the issue operation returns 1 in line 52 since at the time of insertion the tail status was youngest meaning that the task was not inserted into the oldest generation and, hence, it has a dependency. If that operation fails, this can only happen due to a release operation. In that case the tail status may have been updated either to oldest or to none. The issue operation retrieves again the pointer fields and status fields in lines 54-55 (notice that line 54 is not necessary since the release operation only changes the status of the tail) and in the next iteration of the while loop will enter either case 1 or case 2.

```
1  int issue(task_depend_metadata_t *metadata, task_depend_annotation_t annot, ort_task_node_t *tnode)
2  {
3        task_depend_node_t *node = task_depend_node_alloc();
4
5        bool first_in_gen = false;
6
7        if (annot != task_depend_annotation_input || !metadata->last_gen_in_only){
8              ++metadata->num_gens_issued;
9              metadata->last_gen_in_only = (annot == task_depend_annotation_input);
10             first_in_gen = true;
11       }
12
13       node->task = tnode;
14       node->md = metadata;
15       node->gen_num = metadata->num_gens_issued;
16       node->in_only = (annot == task_depend_annotation_input);
17       tnode->which_dependencies[tnode->which_dependencies_count++] = node;
18       atomic_store(&node->next, NULL);
19
20       task_depend_node_t *local_tail = atomic_load(&metadata->tail);
21       task_depend_node_t *local_tail_ptr = EXTRACT_PTR_FROM_TAIL(local_tail);
22       uintptr_t local_tail_status = EXTRACT_STATUS_FROM_TAIL(local_tail);
23
24       node->prev = local_tail_ptr;
25       atomic_store(&local_tail_ptr->next, node);
26
27       while (true){
28             if (local_tail_status == none_tail_status){
29                   task_depend_node_t *local_head = atomic_load(&metadata->head);
30
31                   if (local_head != local_tail_ptr){
32                         reclaim_depend_nodes(local_tail_ptr->prev);
33                         local_tail_ptr->prev = NULL;
34                         atomic_store(&metadata->head, local_tail_ptr);
35                   }
36
37                   atomic_store(&metadata->tail, MAKE_TAIL(node, oldest_tail_status));
38
39                   return 0;
40             } else if (local_tail_status == oldest_tail_status){
41                   uintptr_t new_status = first_in_gen ? youngest_tail_status : oldest_tail_status;
42
43                   bool succ = atomic_compare_exchange_strong(&metadata->tail, &local_tail, MAKE_TAIL(node, new_status));
44
45                   if (succ){ return first_in_gen; }
46
47                   local_tail_ptr = EXTRACT_PTR_FROM_TAIL(local_tail);
48                   local_tail_status = EXTRACT_STATUS_FROM_TAIL(local_tail);
49             } else{
50                   bool succ = atomic_compare_exchange_strong(&metadata->tail, &local_tail, MAKE_TAIL(node, youngest_tail_status));
51
52                   if (succ){ return 1; }
53
54                   local_tail_ptr = EXTRACT_PTR_FROM_TAIL(local_tail);
55                   local_tail_status = EXTRACT_STATUS_FROM_TAIL(local_tail);
56             }
57       }
58 }
```

Figure 4.4: The algorithm implementing the issue operation with a sequential consistent memory model

4.4 The algorithm implementing the release operation for a writer task

Figure 4.5 shows the release operation for a writer task. The function is called with the node corresponding to the task and a boolean variable i_am_writer which in this case is true. To begin with, the operation retrieves the metadata structure in line 3. Notice that at this time the head pointer points to the sentinel node and the sentinel node points to the node of the writer task since the writer task is the only task in the oldest generation. Thus, the operation must make the head pointer point to the latter node which now serves as the sentinel node (line 8) and reclaim the previous sentinel node. This is done by remembering the node at line 5 and at the end reclaiming it at line 71. The prev pointer of the new sentinel node is made NULL at line 6 since it is now the start of the list. The operation is responsible for updating the dependency counters of the tasks belonging to the next generation. Since the issue operation has recorded the generation number of the writer task in the gen_num variable of the node, the operation knows the number of the next generation which it computes in line 11. The strategy employed by the release operation is to read the tail pointer in line 10 and act accordingly depending on the generation number of the node pointed to by the tail (lines 14-15). There are three cases to consider.

Case 1 (lines 18-33) In this case the next generation is not the youngest one. Thus, the operation does not have to update the tail pointer. The aim is to traverse the nodes following the next pointers updating the dependency counters for each node whose generation number is equal to the next generation number (variable next_gen_num). This is accomplished in lines 22-31. There is a subtle point to consider here. Notice that the algorithm starts with the node following the task's node (line 18). The traversal starts with that node (local_next) and continues in lines 22-31. This is the current node. Observe that first the generation number of the next node is obtained (lines 23-24), then the dependency counter is updated for the task of the current node (lines 26-28) and, lastly, the current node is set to the next node (line 30). This is done to avoid the following problem. Assume that the operation doesn't perform lines 23 and 24. Rather, it immediately updates the dependency counter of the task corresponding to the current node (local_next). Then the operation would update the local_next variable with a load operation on the next pointer of the local_next node. Before it does so, however, assume that that task whose dependency counter was just updated had its dependency counter updated to zero (from other release operations), that this task was a writer task, it terminates and it performs a release operation on the same list as we are now. This would make the local_next node be the sentinel node. This task will update the dependencies of the next generation and it doesn't take long to realize that the local_next node may be reclaimed and thus we cannot read its next pointer (also we cannot read the generation number of the next node since it too may have been reclaimed). To avoid this problem, we first read the generation of the next node and then update the dependency counter for the task associated with the current node. Clearly this strategy solves the problem in case the next generation

we are updating here consists of a single writer task. In case the next generation consists of multiple reader tasks, this strategy also avoids any problems because, as we will see in the release operation for reader tasks, only the last reader (the notion of a last reader will be made clear in the description of the release operation for reader tasks) will reclaim the nodes and since we are not updating the dependency counter of the reader associated with the current node that reader cannot make a release operation on the same metadata structure and hence no node will be reclaimed. Lastly, the operation returns in line 33 with a break statement.

Case 2 (lines 35-58) In this case the next generation is the youngest generation. The release operation in this case considers the memory usage annotation of the youngest generation. If the youngest generation consists of a writer task, in which case the test at line 35 is true, the tail status need not be updated to oldest since the issue operation will insert a task in a new generation (a generation can consist of only one writer task). Hence, in this case the release operation updates the dependency counter of the task in lines 36-38 and then returns in line 40. Otherwise, the youngest generation consists of reader tasks and the issue operation may insert more tasks in the same generation if they are reader tasks. For that reason the tail status is updated to oldest in line 43. This is done with a compare-and-swap operation since a issue operation may concurrently try to update the tail pointer. If the operation fails it means that a issue operation has just added a node at the end of the list and hence the release operation retries in the while loop and will enter one of the cases again. On the other hand, if the operation succeeds, it traverses the nodes up to the node that the tail pointer pointed to when it succeeded the operation in line 43. Notice that any node inserted thereafter, will be handled by the issue operation since the tail has been updated to oldest. Also, as explained in the previous case, since the traversal at lines 49-55 involves reader tasks there is no problem with reclamation of the nodes.

Case 3 (lines 60-64) In this case the tail pointer points to the node of the writer task. In this case the tail status must be updated to none since there exists no next generation to wake up. This is accomplished in line 60 with a compare-and-swap operation since a issue operation may concurrently arrive and update the tail pointer. If that operation fails the release operation retries in the while loop. Otherwise, it returns in line 64.

```
1  void release_out(task_depend_node_t *node, bool i_am_writer)
2  {
3       task_depend_metadata_t *md = node->md;
4
5       task_depend_node_t *reclaim = node->prev;
6       node->prev = NULL;
7
8       atomic_store(&md->head, node);
9
10      task_depend_node_t *local_tail = atomic_load(&md->tail);
11      uint_fast64_t next_gen_num = node->gen_num + 1;
12
13      while (true){
14           task_depend_node_t *local_tail_ptr = EXTRACT_PTR_FROM_TAIL(local_tail);
15           uint_fast64_t local_tail_gen_num = local_tail_ptr->gen_num;
16
17           if (local_tail_gen_num >= next_gen_num + 1){
18                task_depend_node_t *local_next = atomic_load(&node->next);
19                task_depend_node_t *local_next_next;
20                uint_fast64_t local_next_next_gen_num;
21
22                do{
23                     local_next_next = atomic_load(&local_next->next);
24                     local_next_next_gen_num = local_next_next->gen_num;
25
26                     if (atomic_fetch_sub(&local_next->task->depend_count, 1) == 1){
27                          ... mark task for insertion into ready list ...
28                     }
29
30                     local_next = local_next_next;
31                } while (local_next_next_gen_num == next_gen_num);
32
33                break;
34           } else if (local_tail_gen_num == next_gen_num){
35                if (!local_tail_ptr->in_only){
36                     if (atomic_fetch_sub(&local_tail_ptr->task->depend_count, 1) == 1){
37                          .. mark task for insertion into ready list ...
38                     }
39
40                     break;
41                }
42                else{
43                     bool succ = atomic_compare_exchange_strong(&md->tail, &local_tail, MAKE_TAIL(local_tail_ptr, oldest_tail_status));
44
45                     if (!succ){ continue; }
46
47                     task_depend_node_t *local_next = node;
48
49                     do{
50                          local_next = atomic_load(&local_next->next);
51
52                          if (atomic_fetch_sub(&local_next->task->depend_count, 1) == 1){
53                               ... mark task for insertion into ready list ...
54                          }
55                     } while (local_next != local_tail_ptr);
56
57                     break;
58                }
59           } else{
60                bool succ = atomic_compare_exchange_strong(&md->tail, &local_tail, MAKE_TAIL(local_tail_ptr, none_tail_status));
61
62                if (!succ){ continue; }
63
64                break;
65           }
66      }
67
68      if (i_am_writer){
69           reclaim->prev = NULL;
70      }
71      reclaim_depend_nodes(reclaim);
72 }
```

Figure 4.5: The algorithm implementing the release operation for a writer task with a sequential consistent memory model

4.5 The algorithm implementing the release operation for a reader-only task

Figure 4.6 shows the release operation for a reader-only task. This operation is performed by every reader task belonging to the oldest generation. The strategy is that every reader task performing the release operation moves the head pointer to the right (using the next pointers). The last executing reader task from the oldest generation is the last reader task that will try to update the head pointer, and is responsible for reclaiming the previous nodes. In more detail, the release operation starts by loading the head pointer in line 4. The head pointer points to the sentinel node. Then it reads the next node in line 11 into local_next and the node after the next node in line 12 into local_next_next. There are two cases to consider depending on the value of local_next_next.

Case 1 (lines 15-21) In this case the head points to the sentinel node, the sentinel node points to local_next and the local_next node points to local_next_next. If the generation number of the local_next_next node is the next generation number then the operation proceeds to line 34 and, in essence, the local_next node plays the role of the writer task. Otherwise, the head is updated to the local_next node using a compare-and-swap primitive since multiple reader tasks may concurrently try to update the head pointer. If that operation succeeds, the release operation returns (the value of last is false cause otherwise it wouldn't make the compare-and-swap operation). Otherwise, it retries in the while loop. Notice that the release_out operation is called with false as a second parameter. This is done to indicate that the release_out operation is called from a reader task and hence full reclamation of nodes must be done. In that case, the release_out operation doesn't set the prev pointer of the sentinel node to NULL (lines 68-70).

Case 2 (lines 23-29) In this case the value of the local_next_next variable is NULL. This can happen if the head points to the sentinel node and the sentinel node points to the last node of the list. Thus, the tail pointer points to that last node (which is node local_next). Because there may be conflicts with release operations of reader tasks that may be issued, the easiest thing to do is to update the tail pointer to none. After all, all the tasks currently belonging to the oldest generation have finished execution. But since the nodes haven't been reclaimed, the issue operation then after it reads a none status for tail will correct things. To that end, the release operation reads the value of the tail variable in line 23. But, even though at the time that local_next_next was read with a value of NULL the tail pointer points to local_next, at the time the tail is read at line 23 multiple issue operations may have arrived and changed the value of the tail pointer. We want to change the status of the tail pointer to none only if it points to node local_next. The test at line 25 makes that happen. If the tail points to another node the operation retries the while loop in line 25. Note that since no nodes are reclaimed we can continue with the same value of the local_head pointer. Otherwise, the tail pointer is updated to status none in line 27. If that succeeds the operation returns in line 29. Otherwise, the release operation retries in the while loop.

```
1  void release_in(task_depend_node_t *node)
2  {
3       task_depend_metadata_t *md = node->md;
4       task_depend_node_t *local_head = atomic_load(&md->head);
5       task_depend_node_t *local_next;
6       task_depend_node_t *local_next_next;
7       bool last;
8       uint_fast64_t next_gen_num = node->gen_num + 1;
9
10      while (true){
11           local_next = atomic_load(&local_head->next);
12           local_next_next = atomic_load(&local_next->next);
13
14           if (local_next_next){
15                last = (local_next_next->gen_num == next_gen_num);
16
17                if (last){ break; }
18
19                bool succ = atomic_compare_exchange_strong(&md->head, &local_head, local_next);
20
21                if (succ){ break; }
22           } else{
23                uintptr_t local_tail = atomic_load(&md->tail);
24
25                if (EXTRACT_PTR_FROM_TAIL(local_tail) != local_next){ continue; }
26
27                bool succ = atomic_compare_exchange_strong(&md->tail, &local_tail, MAKE_TAIL(local_next, none_tail_status));
28
29                if (succ){ return ; }
30           }
31      }
32
33      if (last){
34           release_out(local_next, false);
35      }
36 }
```

Figure 4.6: The algorithm implementing the release operation for a reader-only task with a sequential consistent memory model

4.6 Notes

The function reclaim_depend_nodes is a simple function that traverses the list starting from the node passed as parameter to the left following the prev pointers and reclaiming each node until the end is found (prev == NULL). When the dependency counter of a task is made zero in the release_out operation, that task is added in a local list and after the release_out operation returns, all the tasks in that local list are added to the ready-list of the task performing the release_out operation.

The time and space complexity analysis remains mostly the same for the lock-free variant presented in this chapter. Also, the algorithms presented here are non-blocking assuming that a task generates a bounded number of children tasks using the same reasoning as in the lock-based list scheme.

I have based the construction of the lock-free list scheme on the lock-free management of queue and list concurrent data structures. These constructions can be found, for example, in Raynal [2015] and Herlihy and Shavit [2012].

Chapter 5

Performance Evaluation

Background on OMPi task scheduling OMPi uses a work-stealing scheduler. Each worker thread has its own ready-list which is handled as a dequeue. A thread adds tasks at the bottom of its ready-list and steals tasks from the top of either its own ready-list or the ready-lists of other threads. To take advantage of the NUMA characteristics of modern architectures, a thread tries to steal a task from the ready-list of a thread that is close in terms of the memory hierarchy. More information about the OMPi compiler can be obtained from http://paragroup.cs.uoi.gr/wpsite/. A relevant paper for the tasking runtime is Agathos et al. [2011].

The experiments have been executed in a machine with two AMD Opteron 6166 processors (Magny-Cours architecture) with 12 cores each for a total of 24 cores. Each processor contains two dies with 6 cores each. Each die has two dedicated memory channels and thus is a NUMA node. The dies are connected with a point-to-point interconnect. Each core has private L1 and L2 caches (64KB and 512KB respectively) and each die has a 6 MB L3 cache. Magny-Cours uses the directory-based cache-invalidation protocol explained in the second chapter.

The code that is generated by OMPi is compiled using gcc version 5.1.0 with -O3 optimization flag.

5.1 Micro Benchmarks

This section presents experiments on microbenchmarks that compare the lock-based and the lock-free list schemes. The lock-based list scheme is denoted as LS-LB and the lock-free list scheme as LS-LF. The microbenchmarks are performed on a single-tag since the issue and release algorithms operate on a single-tag. The benchmarks in the Application Studies section involve multiple-tags.

5.1.1 Single-Tag Case

The setup for the experiments in the single-tag case is as follows. The master thread of the parallel team generates (iteratively 100 times) first W tasks that write a single tag X and then R tasks that read that same tag. An additional parameter Workload specifies the amount of work performed by each task (a loop incrementing a volatile counter). Each experiment is executed 10 times and the average execution time is reported.

Figures 5.1, 5.2, 5.3 and 5.4 show the results. The lock-free method is not better only in the corner case with reader tasks only (W = 0 and R=1 – in that case 100 reader tasks are generated). This is justified since in the lock-based version the reader tasks are serialized by the lock and then they perform a normal decrement operation on the oldest_num_tasks variable. In the lock-free version, on the other hand, the reader tasks may need to contend multiple times in the head pointer resulting longer latency and more cache-misses for the other threads that succeed in the update of the head pointer. This fact, as it seems, hurts performance in this case.

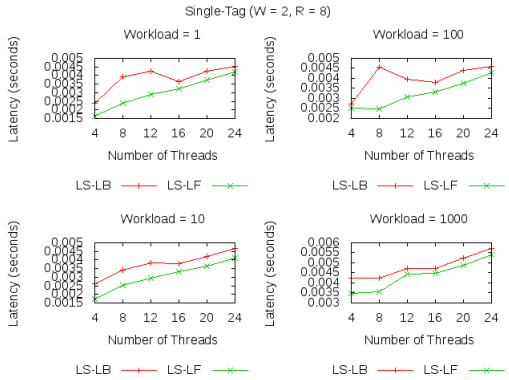

Figure 5.1: Latency results for the Single-Tag Microbenchmark with (W,R) = (2,8)

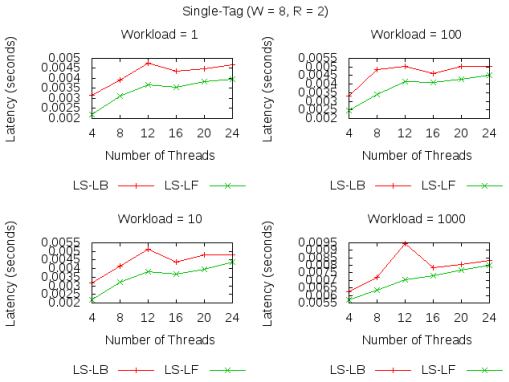

Figure 5.2: Latency results for the Single-Tag Microbenchmark with (W,R) = (8,2)

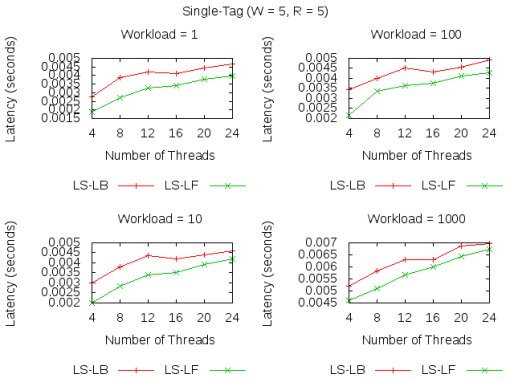

Figure 5.3: Latency results for the Single-Tag Microbenchmark with (W,R) = (5,5)

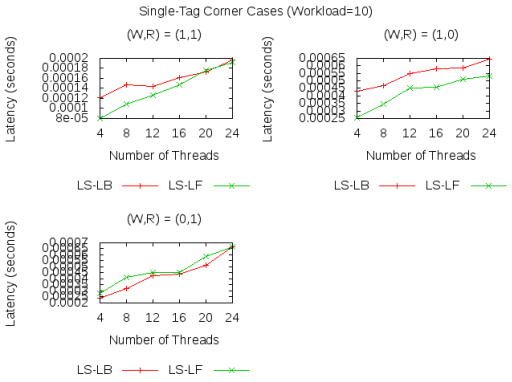

Figure 5.4: Latency results for the Single-Tag Microbenchmark with three corner cases

Figures 5.5, 5.6 and 5.7 show a microbenchmark in which 100000 tasks are issued to the same tag. Each task has a *WriterProb* of being a writer task.

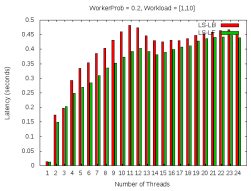

Figure 5.5: Latency results for the Single-Tag Microbenchmark with WriterProb = 0.2

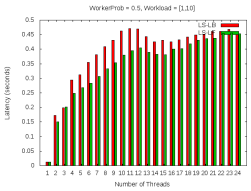

Figure 5.6: Latency results for the Single-Tag Microbenchmark with WriterProb = 0.5

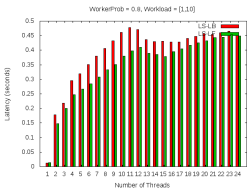

Figure 5.7: Latency results for the Single-Tag Microbenchmark with WriterProb = 0.8

5.1.2 Oldest-Only

The purpose of this micro benchmark is to show the disadvantage of the lock-free scheme compared to the lock-based scheme when there is only one generation active at any time in the task graph. Notice that the lock-based scheme in that case doesn't allocate a node to insert into the list, whereas the lock-free list scheme does so and, thus, needs extra steps to first allocate the node, fill in the details and then insert the node by manipulating the tail pointer. To illustrate this point, i first issue a task that writes some tag and after a while (using the sleep function) i insert another task that writes the same tag. Due to the sleep function i am almost certain that the second writer task is inserted into an empty list. This micro-benchmark is executed only with 2 threads (there is no need for more because at any time only 1 task is active) and the results is 0.00019 for for the lock-based version and 0.00021 for the lock-free version. These numbers show the benefit of the lock-based version in this case.

5.1.3 Top Level Issue

The purpose of this micro benchmark is to evaluate the effectiveness of the top-level issue operation with the following aspect in mind: in the lock-based version while the issue operation is being called to each of the tags the lock for the depend_count variable of the task is being hold. Thus, release operations to tags that the task has already been issued to cannot complete before the top-level issue operation unlocks the lock for the depend_count variable. On the other hand, the lock-free version uses a atomic depend_count variable and, hence, these release operation can complete before the top-level issue operation is finished. This aspect has the potential of increasing parallelism and, consequently, performance. To evaluate this, i have implemented a micro benchmark that uses 100 tags. First, to each tag, a writer task is issued. After a writer task that has a dependence on all tags is being issued. Hence, the first set of tasks will access the depend_count of the latter task in their release operations.

Figure 5.8 shows the results. Clearly, the lock-free version shows some advantage over the lock-based version.

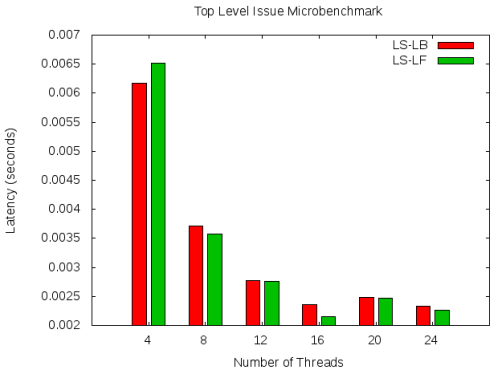

Figure 5.8: Latency results for the Top Level Issue Microbenchmark

5.2 Application Studies

This section presents experiments on three set of realistic applications: LU factorization, Strassen Multiplication and a 2D Recurrence (Stencil). The experiments evaluate the effectiveness of the task dataflow model compared to the task model without dependencies in the OMPi compiler. The lock-based version is denoted with the postfix LB and the lock-free version with the postfix LF.

The setup for each experiment is as follows. First the parameters for the application are chosen. For example, Strassen Multiplication needs the dimensions of the matrix. Next, the number of threads are specified. The experiment is, then, executed for 10 times and a mean value is computed which is reported.

Each benchmark operates on a matrix. To increase the memory bandwidth the array is distributed to the available memory modules. This is accomplished using the *first touch policy*. The recurrence benchmark doesn't initialize the matrix and, hence, each block that is assigned to a thread is allocated when the thread first touches that block to a near memory module. The other benchmarks initialize the array, and by using parallel initialization the array is distributed to the available memory modules.

As far as the lock-free version is concerned, these applications fit in the categories (1) corner cases and (2) oldest-only. For example, in the Strassen application, each tag receives on average 2 nodes. What is more, it is very likely that these nodes will be issued to an empty list, and, hence, fit in the oldest-only category. There are also 8 tags with only reader tasks which as we saw in the corner cases category has the potential of hurting performance. For the Recurrence application, strangely enough, the lock-free version performs better even though it also fits in the cases above. This also happens for the LU application, except that for the largest matrix size.

5.2.1 Recurrence

The first application is a 2D recurrence. In this application a function f is applied to all elements of a 2-dimensional matrix with a row size and column size equal to N. At the beginning, the matrix is uninitialized. The function f assigns the value 1 to all elements in the first row and column, and for each other element the function is $f(i,j) = f(i,j-1) + f(i-1,j)$, where i is the row number and j is the column number with $1 \leq i < N$ and $1 \leq j < N$. In other words, the function f assigns to each element the sum of the elements on the top and on the left. Figure 5.9 shows the data dependencies for the function f on a matrix with N = 8.

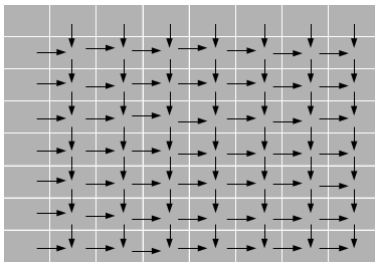

Figure 5.9: Data dependencies for the Recurrence application for a matrix with N = 8

Sequential Implementation A straightforward sequential implementation of the recurrence application involves a traversal of the array row-by-row as shown in figure 5.10.

Parallelization Strategy To parallelize the recurrence application we traverse the array in *diagonal order* as shown in figure 5.11. For example, as soon as the diagonal D(6) has been computed, D(7) can be computed in parallel since the elements in D(7) depend on elements on the previous diagonal D(6).

```
1  void recurrence_seq(int size, int *array)
2  {
3        for (int i = 0; i < size; ++i)
4        {
5              for (int j = 0; j < size; ++j)
6              {
7                    if (i == 0 || j == 0)
8                    {
9                          array[i*size + j] = 1;
10                   }
11                   else
12                   {
13                         array[i*size + j] = f(array[i*size + j-1],array[(i-1)*size + j]);
14                   }
15             }
16       }
17 }
18
```

Figure 5.10: Straighforward sequential implementation for the recurrence application. The parameter **array** is a pointer to a 2-dimensional matrix with row size and column size equal to **size**. The matrix is stored in row-major order and, thus, element (i,j) is accessed as array[i*size + j].

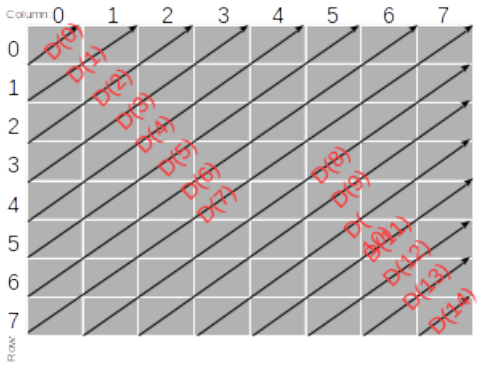

Figure 5.11: Parallelization strategy for the Recurrence application for a matrix with N = 8

Task Implementation The implementation using tasks is based on the parallelization strategy shown above. To begin with, a task operates on a block of the original matrix with dimensions *block_sizexblock_size*. The choice of the block size depends on the overhead of maintaining the tasks generated. Inside a block the task computes the recurrence function f using the straightforward sequential implementation. For that reason, to take advantage of cache-locality it suffices that one row of the block fits in the L1 cache. This, however, is almost always the case for any choice of N and block_size with a large L1 cache. Thus, the block_size parameter is not affected by cache considerations. However, a very small block_size parameter results in a larger amount of tasks being generated and the overhead of maintaining those extra tasks could overcome the benefit of parallelization. Thus, the block_size must be chosen so that the sequential implementation on that block is faster than splitting the block into sub-blocks and generating for tasks for the sub-blocks. A last consideration is that the N and block_size parameters in the experiments shown later are chosen to be multiplies of the cache-line size in order to avoid false-sharing between writes to different blocks.

The implementation using tasks uses the *taskwait* construct for synchronization. In order to compute a diagonal D(i) the previous diagonal D(i-1) must have been computed, and for that reason the task implementation traverses the matrix in diagonal order starting with D(0), and for each diagonal it first generates the tasks for each block on that diagonal and then performs a *taskwait* to wait for the termination of those tasks. In that way, we know that the current diagonal has been computed and we can proceed on to the next diagonal. Figure 5.12 shows the implementation of the recurrence application using tasks and the taskwait construct for synchronization.

In more detail, the recurrence_task() function first creates a parallel team comprised of num_threads threads to collectively evaluate the recurrence. Then, one thread is assigned the job of traversing the array in diagonal order using the master construct in line 5. The number of diagonals is computed in line 9 and the for loop in line 11 traverses each diagonal in order. The for loop at lines 21-44 generates the tasks for the current diagonal. At the end, at line 46 the master thread waits on the completion of the tasks for the current diagonal using the taskwait construct. The rest of the code consists of details on how to handle block boundaries for each task generated.

```
1  void recurrence_task(int size, int *array, int num_threads, int block_size)
2  {
3      #pragma omp parallel num_threads(num_threads) firstprivate(size,array,block_size)
4      {
5          #pragma omp master
6          {
7              int i, j;
8              int sub_size = size/block_size;
9              int num_sweeps = 2*sub_size-1;
10
11             for (i = 0; i < num_sweeps; ++i)
12             {
13                 int sweep_size = (i < sub_size) ? (i + 1) : (2*sub_size - i-1);
14                 int x = (i < sub_size) ? (i) : (sub_size - 1);
15                 int y = (i < sub_size) ? (0) : (i - sub_size + 1);
16
17                 x *= block_size;
18                 y *= block_size;
19
20                 // spawn tasks for this sweep
21                 for (j = 0; j < sweep_size; ++j)
22                 {
23                     #pragma omp task firstprivate(x,y,array,block_size,size)
24                     {
25                         for (int ii = x; ii < x + block_size; ++ii)
26                         {
27                             for (int jj = y; jj < y + block_size; ++jj)
28                             {
29                                 if (ii == 0 || jj == 0)
30                                 {
31                                     array[ii*size + jj] = 1;
32                                 }
33                                 else
34                                 {
35                                     array[ii*size + jj] = f(array[ii*size+jj-1]
36                                                             ,array[(ii-1)*size + jj]);
37                                 }
38                             }
39                         }
40                     }
41
42                     x -= block_size;
43                     y += block_size;
44                 }
45
46                 #pragma omp taskwait
47             }
48         }
49     }
50 }
51
```

Figure 5.12: OpenMP task implementation for the Recurrence application using the taskwait construct for synchronization.

Task Dataflow Implementation The task dataflow adaptation for the recurrence application is straightforward. First, the taskwait statements are removed. Then, for each task generated we add, using the depend clause, the block that it reads and writes. In this way, the runtime is responsible for executing the tasks in the correct order. The opportunities for increased parallelism is that the task graph generated at runtime is the one shown in figure 5.9. In this way, as soon as the two blocks needed for some block have been computed, that block will be executed immediately by the runtime system. On the contrary, in the task implementation discussed previously, first all tasks in one diagonal must terminate and then the tasks of the next diagonal can be generated. Actually, for the task dataflow implementation a traversal in row order is also correct because the runtime system will honor the dependencies. In addition, as the experimental results, a traversal in row order results in better results which i attribute to better cache locality for the metadata structures. The dataflow annotations with a traversal in row order are shown in figure 5.13.

The for loop (lines 9 and 11) traverse the matrix in row order. Inside the loop body (lines 13-53) the (x,y) pair indicates the start of the current block. There are four cases to consider: If the block is the first one (x == 0 and y == 0), then the task doesn't read anything but writes itself. Thus a depend clause with an *inout* annotation is used on the block itself (line 16). If the block is in the first row, but not the first, then it reads the block on its left and also writes itself. For the block on its left a *in* annotation is used in line 25 and a *inout* annotation is used for the block in line 26. If the block is in the first column, but not the first, then it reads the block above it and also writes itself. For the block above it a *in* annotation is used in line 35 and a *inout* annotation is used for the block in line 36. Otherwise, the block has to read both the block on its left and the block above it, so the last case uses two *in* annotations in line 45 and, also, the *inout* annotation for the write on the block in line 47. Note that the memory locations that are used in the depend clauses are the (x,y) entries, that is the first element of each block, since that is sufficient.

```
1  void recurrence_taskdep_rowwise(int size, int *array, int num_threads, int block_size)
2  {
3      #pragma omp parallel num_threads(num_threads) firstprivate(size,array, block_size)
4      {
5          #pragma omp master
6          {
7              int x, y;
8
9              for (x = 0; x < size; x += block_size)
10             {
11                 for (y = 0; y < size; y += block_size)
12                 {
13                     if (x == 0 && y == 0)
14                     {
15                         #pragma omp task firstprivate(x,y,array,block_size,size)
16                                 depend(inout: array[x*size+y])
17                         {
18                             // ...
19
20                         }
21                     }
22                     else if (x == 0)
23                     {
24                         #pragma omp task firstprivate(x,y,array,block_size,size)
25                                 depend(in: array[x*size + y - block_size])
26                                 depend(inout: array[x*size+y])
27                         {
28                             // ...
29
30                         }
31                     }
32                     else if (y ==0)
33                     {
34                         #pragma omp task firstprivate(x,y,array,block_size,size)
35                                 depend(in: array[(x-block_size)*size + y])
36                                 depend(inout: array[x*size+y])
37                         {
38                             // ....
39
40                         }
41                     }
42                     else
43                     {
44                         #pragma omp task firstprivate(x,y,array,block_size,size)
45                             depend(in: array[x*size + y - block_size],
46                                 array[(x-block_size)*size + y])
47                             depend(inout: array[x*size+y])
48                         {
49                             // ...
50
51                         }
52
53                     }
54                 }
55             }
56         }
57     }
58 }
59
```

Figure 5.13: OpenMP task dataflow annotations for the Recurrence application with a traversal in row order.

Comparative Study between Task and Task-Dataflow Implementation The recurrence application has been executed with $(N, block\_size) = (32768, 512)$ and $(N, block\_size) = (16384, 256)$, and for 1 up to 24 number of threads. Figures 5.14 and 5.15 show the results. From the figures it can be seen that the task dataflow versions perform better because they exploit more parallelism than the task version.

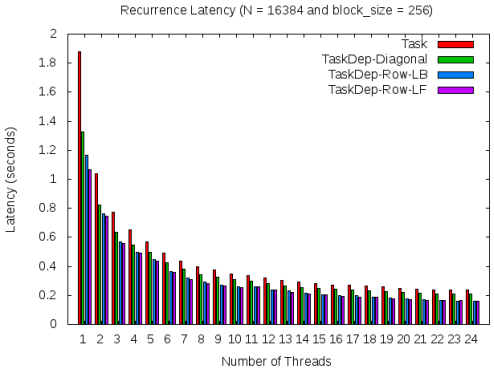

Figure 5.14: Latency results for OMPi (task versus task-dep implementation) for the recurrence application

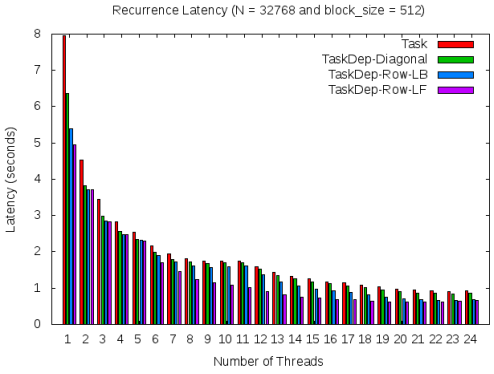

Figure 5.15: Latency results for OMPi (task versus task-dep implementation) for the recurrence application

5.2.2 Strassen Multiplication

The implementation for the Strassen Multiplication benchmark is from the KASTORS benchmark suite (http://kastors.gforge.inria.fr). Implementation details are omitted here. Figure 5.19 shows the parallelization patterns for the task and task-dataflow implementations.

Comparative Study between Task and Task-Dataflow Implementation The strassen multiplication has been executed with block_size equal to 64 and matrix sizes 1024, 2048, 4096. The block size is 64 because this benchmark multiplies submatrices together and with a 64 by 64 matrix this computation fits in the L1 cache, thus, producing better results. Figures 5.16, 5.17 and 5.18 show the results. From the figures it can be seen that the task dataflow version performs better because it exploits more parallelism than the task version.

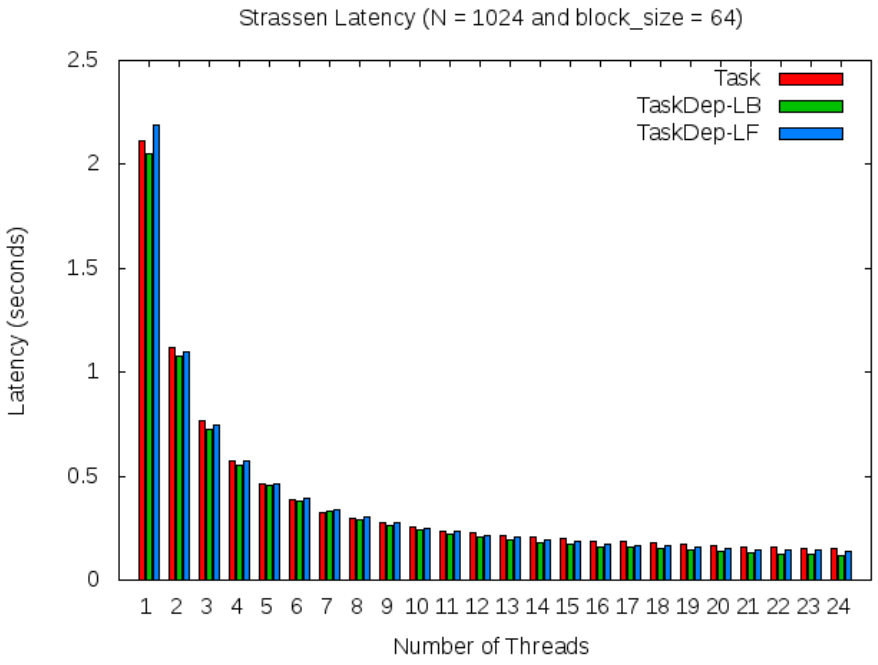

Figure 5.16: Latency results for OMPi (task versus task-dep implementation) for the strassen application

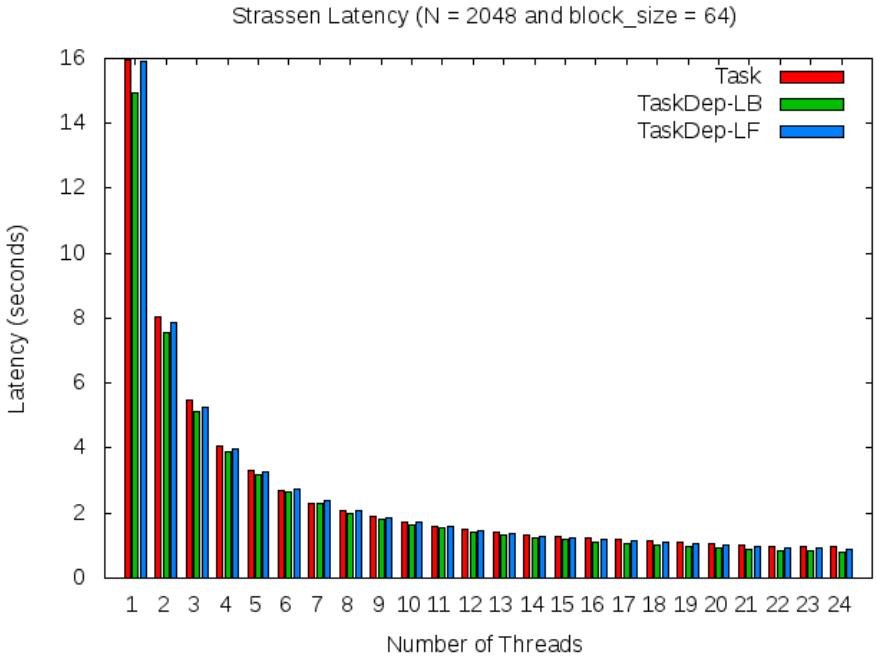

Figure 5.17: Latency results for OMPi (task versus task-dep implementation) for the strassen application

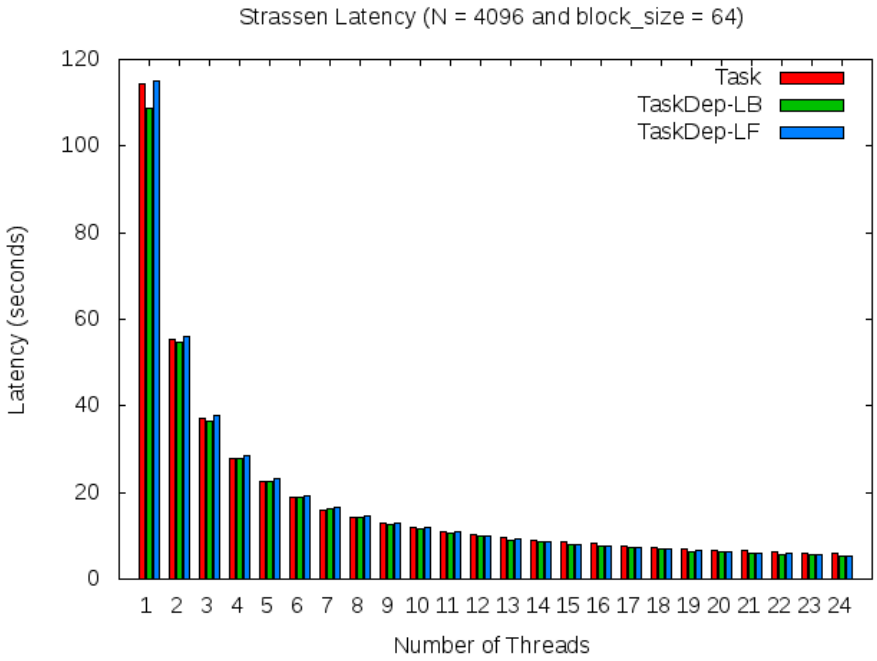

Figure 5.18: Latency results for OMPi (task versus task-dep implementation) for the strassen application

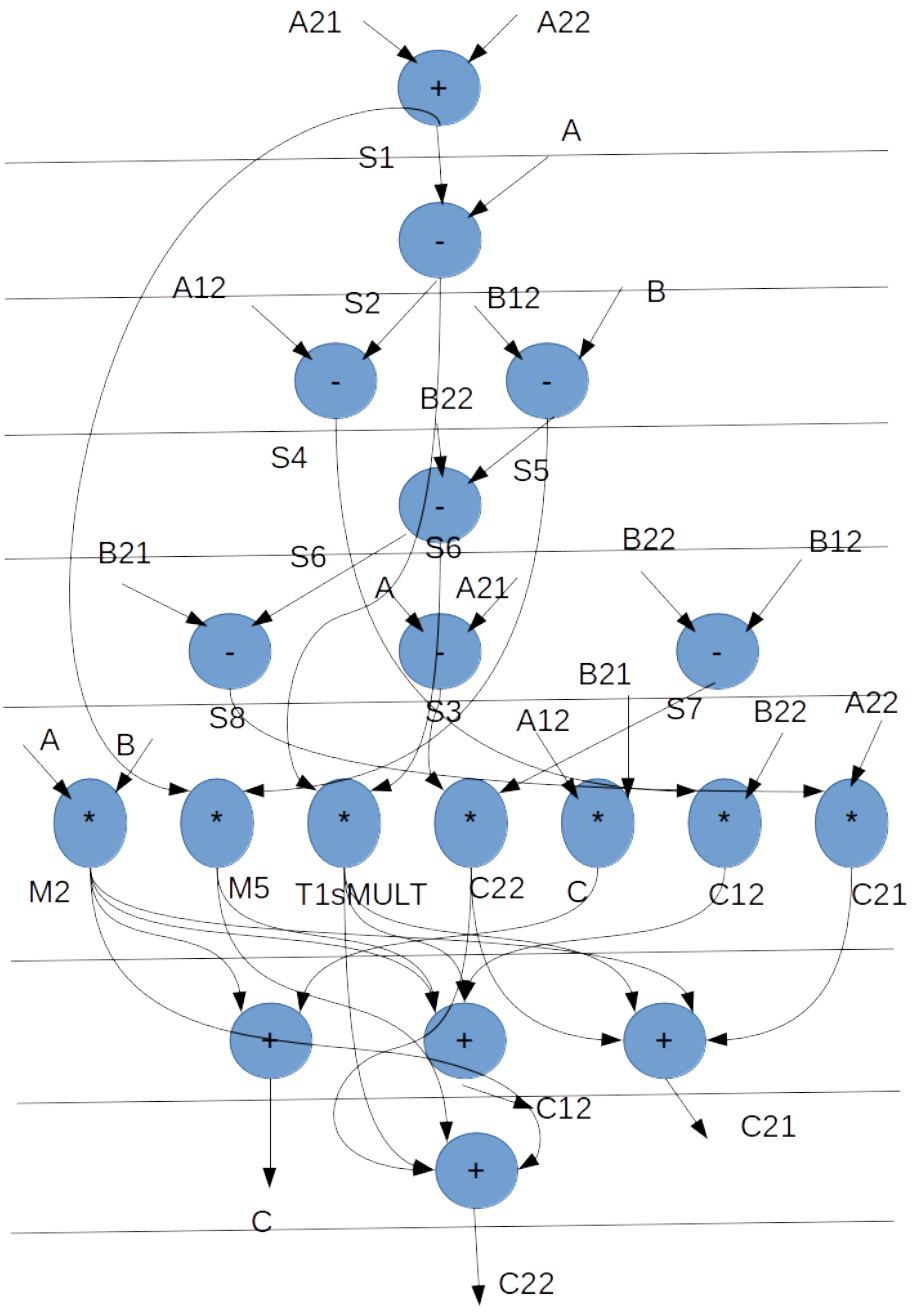

Figure 5.19: Parallelization patterns for the task and task-dataflow implementation of the Strassen Multiplication benchmark. Each node denotes a computation on the input matrices. The task-dataflow implementation generates the graph and performs one taskwait statement at the end. The task implementation generates the graph level by level as shown by the horizontal lines (that is, each horizontal line is a taskwait statement).

5.2.3 LU Factorization

The LU decomposition of a N by N matrix essentially performs Gaussian elimination on that matrix and stores in-place the multipliers used for each step of the Gaussian elimination. For this presentation the exact computation performed is irrelevant. Instead, the parallelization strategy is presented in figure 5.20.

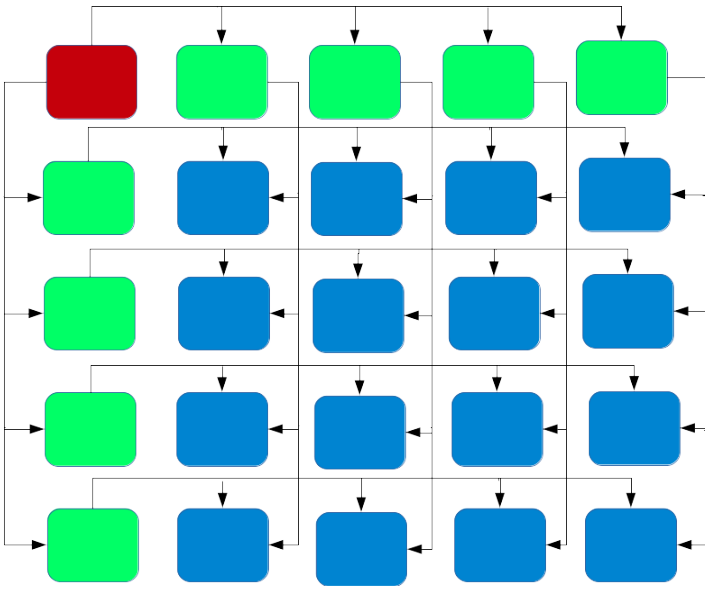

Figure 5.20: Parallelization strategy for the LU application

The matrix is partitioned into sub-blocks. The LU kernel applies the pattern shown in figure 5.20 to each block on the diagonal from top to bottom. The exact example shown in the figure is the first iteration. On the second iteration, the block on the second row and column would be red, the blocks on the right and below it green and the rest of the blocks blue. First, Gaussian elimination is performed in the red block. After this computation is finished, computation starts in the green blocks. Each step of the Gaussian elimination performed in the red block, normally needs to be performed in the entire row of the array, that is also for the rest of the blocks on the same row which are the green blocks (on right of the red block). As a result, the green blocks on the right of the red block have an input dependency on the red block. Moreover, the purpose of the first iteration is to eliminate the entries below the diagonal of the red block. Consequently, the process must continue on the green blocks below the red block. This must be done after Guassian elimination is performed in the red block because elimination proceeds from top to bottom. Hence, the green blocks below the red block have a dependency on the red block. Now consider some green block below the red block. Each step performed there must be propagated to the entire row of blocks on the left. This must be done for each such green block and, thus, the blue blocks are computed after these green blocks. Each blue block has a dependency on the greek block on its left and above it.

Task Implementation Figure 5.21 shows the task implementation for the LU kernel. The master thread at lines 9-38 makes a diagonal sweep over the array. For each iteration the lines 11-37 implement the aforementioned pattern. Function lu0 performs the computation on the red block. This function is called sequentially. After the function returns, the green blocks are generated at lines 13-23. At line 25, with a taskwait statement, the implementation ensures that all green blocks have terminated. Thus, it proceeds on generation the blue tasks at lines 27-35. The taskwait statement at line 37 waits for the blue tasks to terminate before the master thread continues on the next block on the diagonal.

```
1  void lu_decomposition_task(double *array, size_t size, size_t block_size, unsigned int nthreads)
2  {
3   size_t diagonal_index, right_index, i_index, j_index,down_index;
4
5   #pragma omp parallel firstprivate(size,block_size) shared(array) \
6       private(diagonal_index,right_index,i_index,j_index,down_index) num_threads(nthreads)
7
8   #pragma omp master
9   for (diagonal_index = 0; diagonal_index < size; diagonal_index += block_size)
10  {
11    lu0(array, diagonal_index, diagonal_index, block_size, size);
12
13    for (right_index = diagonal_index + block_size; right_index < size; right_index += block_size)
14    {
15       #pragma omp task untied firstprivate(diagonal_index, right_index, block_size, size) shared(array)
16       fwd(array, diagonal_index, diagonal_index, diagonal_index, right_index, block_size, size);
17    }
18
19    for (down_index = diagonal_index + block_size; down_index < size; down_index += block_size)
20    {
21       #pragma omp task untied firstprivate(diagonal_index, down_index, block_size, size) shared(array)
22       bdiv(array, diagonal_index, diagonal_index, down_index, diagonal_index, block_size, size);
23    }
24
25    #pragma omp taskwait
26
27    for (i_index = diagonal_index + block_size; i_index < size;i_index += block_size)
28    {
29       for (j_index = diagonal_index + block_size; j_index < size; j_index += block_size)
30       {
31         #pragma omp task untied firstprivate(i_index, diagonal_index, j_index, block_size, size) \
32                          shared(array)
33         bmod(array, i_index, diagonal_index, diagonal_index, j_index, i_index, j_index, block_size, size);
34       }
35    }
36
37    #pragma omp taskwait
38  }
39 }
40
```

Figure 5.21: Task Implementation for the LU kernel

Task Dataflow Implementation Figure 5.22 shows the task dataflow implementation for the LU kernel. The differences are: first, the lu0 task on the red block is generated with an inout dependency on that block. The green blocks are then generated with a input dependency on the red block and an inout dependency on themselves. Then, without a taskwait statement, the blue tasks are generated with an input dependency on the two green blocks on the left and above and an inout dependency on themselves. The parallelization benefit of this dataflow implementation is that each blue block doesn't need to wait for all green blocks to finish before it can start execution (as it is done in the task implementation), but only on the two green blocks it directly depends on.

```
1  void lu_decomposition_task_dep(double *array, size_t size, size_t block_size, unsigned int nthreads)
2  {
3    size_t diagonal_index, right_index, i_index, j_index,down_index;
4
5    #pragma omp parallel firstprivate(size,block_size) shared(array) \
6              private(diagonal_index,right_index,i_index,j_index,down_index) num_threads(nthreads)
7    #pragma omp master
8    for (diagonal_index = 0; diagonal_index < size; diagonal_index += block_size)
9    {
10       #pragma omp task shared(array) firstprivate(diagonal_index, block_size, size) \
11                        depend(inout: array[diagonal_index*size+diagonal_index])
12       lu0(array, diagonal_index, diagonal_index, block_size, size);
13
14       for (right_index = diagonal_index + block_size; right_index < size; right_index += block_size)
15       {
16            #pragma omp task firstprivate(diagonal_index, right_index, block_size, size) shared(array) \
17                        depend(in: array[diagonal_index*size+diagonal_index]) \
18                             depend(inout: array[diagonal_index*size+right_index])
19            fwd(array, diagonal_index, diagonal_index, diagonal_index, right_index, block_size, size);
20       }
21
22       for (down_index = diagonal_index + block_size; down_index < size; down_index += block_size)
23       {
24            #pragma omp task firstprivate(diagonal_index, down_index, block_size, size) shared(array) \
25                                  depend(in: array[diagonal_index*size+diagonal_index]) \
26                                  depend(inout: array[down_index*size+diagonal_index])
27            bdiv(array, diagonal_index, diagonal_index, down_index, diagonal_index, block_size, size);
28       }
29
30       for (i_index = diagonal_index + block_size; i_index < size;i_index += block_size)
31       {
32            for (j_index = diagonal_index + block_size; j_index < size; j_index += block_size)
33            {
34                 #pragma omp task firstprivate(i_index, diagonal_index, j_index, block_size, size) shared(array) \
35                      depend(in: array[i_index*size+diagonal_index], array[diagonal_index*size+j_index]) \
36                      depend(inout: array[i_index*size+j_index])
37                 bmod(array, i_index, diagonal_index, diagonal_index, j_index, i_index, j_index, block_size, size);
38            }
39       }
40
41       #pragma omp taskwait
42   }
43 }
44
```

Figure 5.22: Task Dataflow Implementation for the LU kernel

Comparative Study between Task and Task-Dataflow Implementation
The strassen multiplication has been executed with block_size equal to 64 and matrix sizes 1024, 2048, 4096, 8192. The block size is 64 because this benchmark multiplies submatrices together and with a 64 by 64 matrix this computation fits in the L1 cache, thus, producing better results. Figures 5.23, 5.24, 5.25 and 5.26 show the results. From the figures it can be seen that the task dataflow version performs better for large matrix sizes and not so much better for small matrix sizes. In this case, the overhead of handling the task dependencies at runtime overcomes the benefits of parallelization.

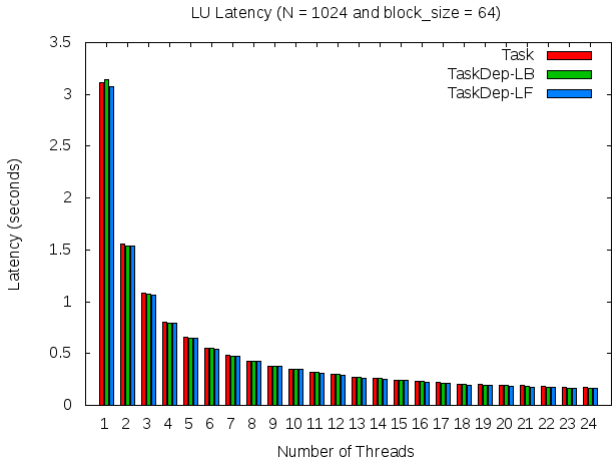

Figure 5.23: Latency results for OMPi (task versus task-dep implementation) for the LU application

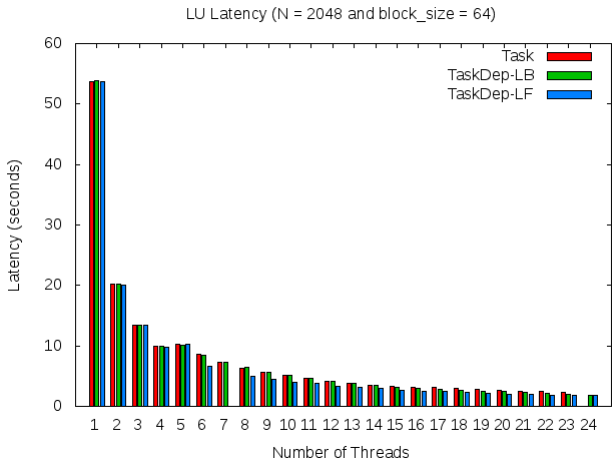

Figure 5.24: Latency results for OMPi (task versus task-dep implementation) for the LU application

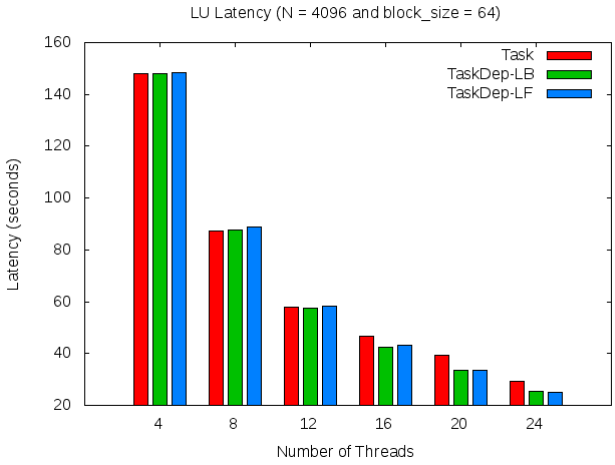

Figure 5.25: Latency results for OMPi (task versus task-dep implementation) for the LU application

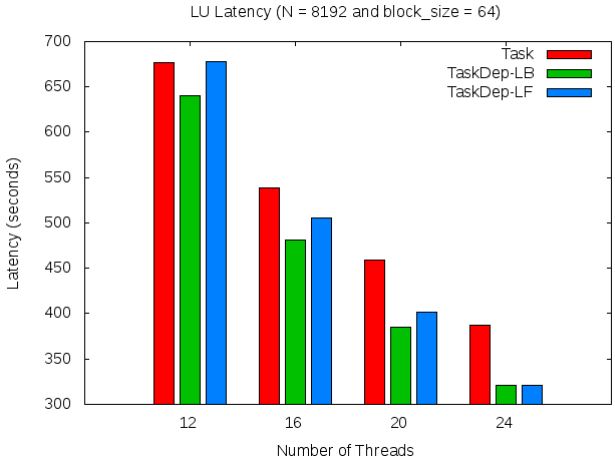

Figure 5.26: Latency results for OMPi (task versus task-dep implementation) for the LU application

5.2.4 Comparison to Other OpenMP 4.0 Task Dependencies Runtimes

In this section i present a comparative study between the OMPi runtime as implemented for the purpose of this thesis and various OpenMP runtimes. The OpenMP runtimes that i have selected are (1) the LibGOMP runtime for the GCC compiler version 5.1.0, (2) the LibIOMP runtime (libiomp5 version) from Intel that i have linked to the code generated by the CLANG compiler version 3.5.0, and the XKaapi library (version 3.0) that i link to the code generated by the GCC compiler (also version 5.1.0). For OMPi i denote as OMPi-LB the lock-based list scheme version and as OMPi-LF the lock-free list scheme version.

Recurrence Application For the Recurrence Application i have executed the various runtimes for the matrix size of 16384. The LibIOMP runtime in this case hanged and, hence, no results are reported for that runtime. Figure 5.27 shows the results. It can be seen that the OMPi versions perform favourably compared to the other runtimes.

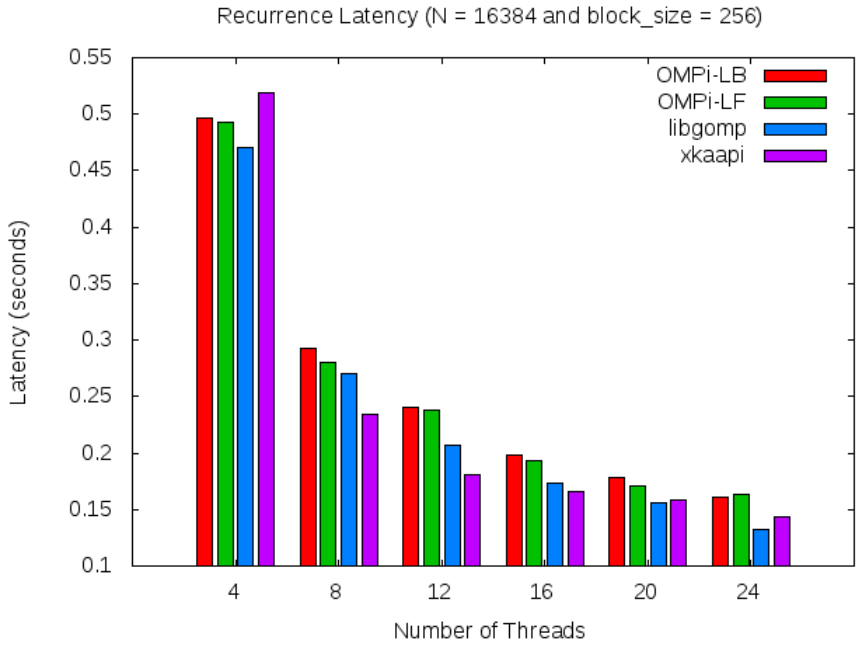

Figure 5.27: Latency results for Recurrence application for various OpenMP runtimes

Strassen Application Figures 5.28, 5.29 and 5.30 show the results. Generally, the OMPi-LB version performs better in all cases, with the LibIOMP runtime catching up for thread numbers larger than 20.

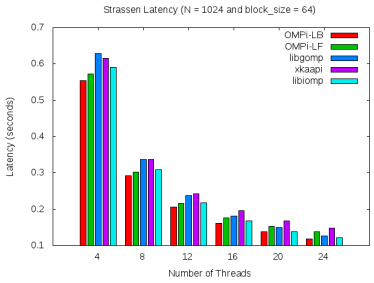

Figure 5.28: Latency results for Strassen application for various OpenMP runtimes

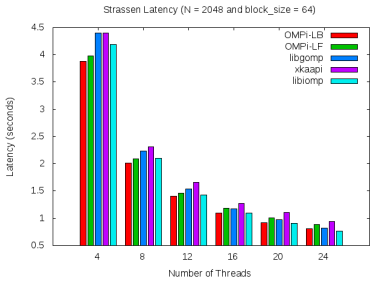

Figure 5.29: Latency results for Strassen application for various OpenMP runtimes

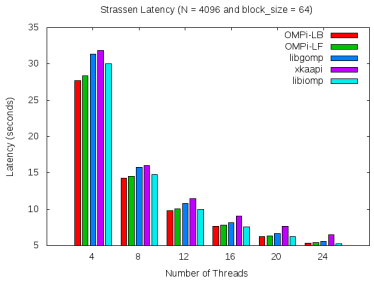

Figure 5.30: Latency results for Strassen application for various OpenMP runtimes

LU Factorization For this application i present results for matrix sizes 1024, 2048 and 4096. The LibIOMP runtime, again, hanged in my experiments and, hence, no results are presented for that runtime. Figure 5.31, 5.32 and 5.33 show the results. For matrix sizes 1024 and 2048 the OMPi-LF version outperforms the other runtimes. For matrix size 4096 the results generally favour the LibGOMP and Xkaapi runtimes.

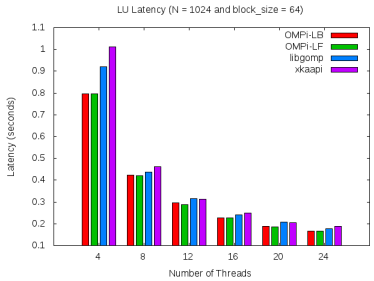

Figure 5.31: Latency results for LU application for various OpenMP runtimes

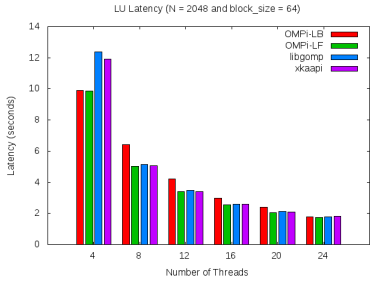

Figure 5.32: Latency results for LU application for various OpenMP runtimes

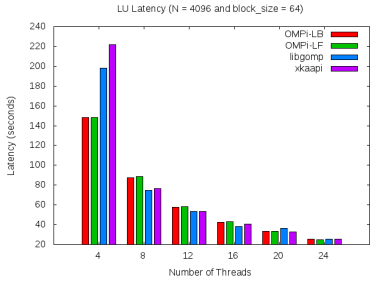

Figure 5.33: Latency results for LU application for various OpenMP runtimes

Chapter 6

Conclusion

OpenMP 4.0 boosts parallel productivity by adding support for the task dataflow model. The task dataflow model is a intuitive parallel programming model that expresses a parallel algorithm with a set of tasks together with their data dependencies. The runtime infrastructure is responsible for maintaining these data dependencies and correctly executing the resulting task graph.

In this thesis i have implemented the List Scheme as presented in Vandierendonck et al. [2013] in order to handle runtime data dependencies in the OMPi compiler. I have also developed a lock-free variant of the list scheme. The experiments presented in Chapter 4 demonstrate the effectiveness of the task dataflow model over the task model without dependencies. The experiments also show that the lock-free variant needs further improvements, since even though it outperforms the lock-based version in high concurrency scenarios as illustrated in the micro benchmarks, in the common cases utilized by applications like those studied in the Performance Evaluation chapter the lock-free scheme is less suited. Last but not least, the implementation performs well against other OpenMP runtimes, and in particular the libgomp runtime by gcc, the libiomp runtime by Intel and the Xkaapi runtime.

6.0.5 Future Work

As future work i indent to first optimize the memory orderings for the lock-free variant to reduce the cost of unnecessary memory fences. Then, my goal is to simplify and optimize algorithmically the lock-free variant even further. Another line of research involves a lock-free and highly optimized implementation of the Tickets Scheme.

Bibliography

Daniel J. Sorin, Mark D. Hill, and David A. Wood. *A Primer on Memory Consistency and Cache Coherence*. Morgan & Claypool Publishers, 1st edition, 2011. ISBN 1608455645, 9781608455645.

Michel Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer Publishing Company, Incorporated, 2015. ISBN 3642446159, 9783642446153.

Anthony Williams. *C++ Concurrency In Action: Practical Multithreading*. Manning Publications, 2012. ISBN 9781933988771.

Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing c++ concurrency. *SIGPLAN Not.*, 46(1):55–66, January 2011. ISSN 0362-1340. doi: 10.1145/1925844.1926394. URL `http://doi.acm.org/10.1145/1925844.1926394`.

Hans-J. Boehm and Sarita V. Adve. Foundations of the c++ concurrency memory model. *SIGPLAN Not.*, 43(6):68–78, June 2008. ISSN 0362-1340. doi: 10.1145/1379022.1375591. URL `http://doi.acm.org/10.1145/1379022.1375591`.

Hans Vandierendonck, George Tzenakis, and Dimitrios S. Nikolopoulos. Analysis of dependence tracking algorithms for task dataflow execution. *ACM Trans. Archit. Code Optim.*, 10(4):61:1–61:24, December 2013. ISSN 1544-3566. doi: 10.1145/2555289.2555316. URL `http://doi.acm.org/10.1145/2555289.2555316`.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming, Revised Reprint*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 9780123973375, 9780123977953.

Spiros N. Agathos, Panagiotis E. Hadjidoukas, and Vassilios V. Dimakopoulos. Design and implementation of openmp tasks in the ompi compiler. In *Proceedings of the 2011 15th Panhellenic Conference on Informatics*, PCI '11, pages 265–269, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4389-5. doi: 10.1109/PCI.2011.34. URL `http://dx.doi.org/10.1109/PCI.2011.34`.

Michael L. Scott. *Shared-Memory Synchronization*. Morgan & Claypool Publishers, 2013. ISBN 160845956X, 9781608459568.

Michel Dubois, Murali Annavaram, and Per Stenstrm. *Parallel Computer Organization and Design*. Cambridge University Press, New York, NY, USA, 2012. ISBN 0521886759, 9780521886758.

Michael McCool, James Reinders, and Arch Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2012. ISBN 9780123914439, 9780124159938.