IRWIN AND JOAN JACOBS
CENTER FOR COMMUNICATION AND INFORMATION TECHNOLOGIES

Scheduling Directives for Shared-Memory Many-Core Processor Systems

Oded Green, Yitzhak Birk

CCIT Report #803
January 2012

Electronics
Computers
Communications

DEPARTMENT OF ELECTRICAL ENGINEERING
TECHNION - ISRAEL INSTITUTE OF TECHNOLOGY, HAIFA 32000, ISRAEL

CCIT Report #803 January 2012

Scheduling Directives for Shared-Memory Many-Core Processor Systems

Oded Green, Yitzhak Birk

Electrical Engineering Department
Technion – Israel Inst. of Technology
Haifa 32000, Israel

Contents

Abstract

We consider a shared-memory (no private caches) many-core architecture. A program comprises a set of single-core tasks along with a set of precedence relations among them, which represent data dependences and ensure correct execution. For reasons such as programming convenience and reduced code foot print, multiple-instance ("duplicable") tasks are used in data-parallel situations such as summing up the rows of a matrix. Dispatching tasks to cores is done by hardware within very few clock cycles and at a very high rate. This is thus a dataflow machine at the inter-task level, with conventional control flow within each task. The Plurality Hypercore is such an architecture.

The precedence constraints guarantee correctness, and the absence of private caches obviates the need to consider which core should execute any given task. However, one must still decide the dispatching order whenever the number of runnable tasks exceeds that of available cores. This choice among correct execution orders can impact performance: 1) it can mitigate bottlenecks, namely situations wherein a task that must precede many others is scheduled later than it could have been and now causes cores to be idle awaiting its completion, and 2) it can impact the instantaneous memory footprint of the program and its data, thereby affecting the hit rate of the shared cache. For a given number of cores and a program with known task execution times, one could simply add precedence relations in order to enforce the desired scheduling order. This, however, is impossible in general, and the problem is most acute with duplicable tasks, as the precedence constraints apply jointly to all task instances.

This work focuses on scheduling constructs ("primitives") that can be used by programmers and by automatic optimization tools to further direct the runtime dispatcher, with special attention to duplicable tasks. Such constructs must express relations that occur in real programs and whose translation into scheduling directives impacts performance. Additionally, they must lend themselves to efficient implementation in hardware. We present several such primitives that increase the expressive power of the programmer and/or optimizer, along with examples in which they increase performance with different numbers of cores. We also outline their implementation in the context of a Hypercore-like system, thereby proving them to be practical.

1 Introduction

1.1 Many-core systems

A single core system is a system that has one core that can execute programs. With the advance in technology, it has become possible to place multiple cores on a single chip/die or on multiple chips/dies and interconnecting them. Furthermore, from a power performance point of view, it has become less practical to increase the clock speed of single cores system due to the increase in power requirements. The term “many-core” is used to refer to cores whose efficient use requires the effective parallelization of an application rather than merely relying on the existence of a sufficient number of concurrently running independent applications. One should thus think of at least 16 cores.

1.2 Shared-memory systems

Shared memory refers to memory that will be accessed by several cores simultaneously. The shared memory is both physically and logically shared by the different cores. Logically, the address space is the same. Physically, the same hardware-memory is used by the different cores. There are two types of shared memory systems:

- NUMA – Non Uniform Memory Access – access time of a given core to different memory addresses in the same level of the memory hierarchy varies due to the fact that different parts of the memory are at different distances from that core.
- UMA –Uniform Memory Access – access time to the entire same-level memory is equal.

Also, it is possible to have memory hierarchies such as:

- Private caches for the cores – in this hierarchy it is possible to reduce the latency of memory requests by keeping data near the core. However, this approach suffers from the need for coherency-preserving mechanisms.
- Several layers of shared memory – for example, a hierarchy with a shared cache for all the cores and an additional (larger) shared memory that the shared caches goes to for cache misses. Note that UMA/NUMA refers to same-level access times, and is thus “orthogonal” to the issue of a multi-level hierarchy.

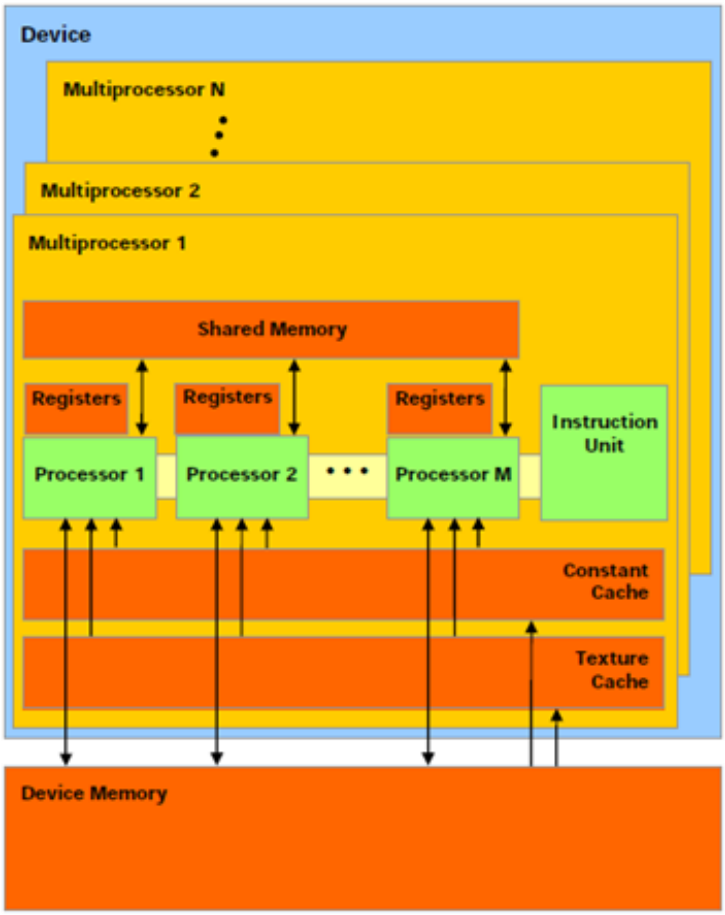


Figure 1 - A set of multi-cores with on-chip shared memory. Essentially this is the GPU .Image taken from NVidia.

1.3 Existing many-core systems

Today there are numerous many-cores systems. A few of them will now be presented.

1.3.1 Graphic Processing Units

Graphic Processing Units (GPU) are systems that originally were created for high-speed rendering of complex graphics for display on computer screens. As graphical computation is intensive and inherently parallel, these systems were widened in such a way that they would be able to deal with multiple data elements concurrently. In recent years, following Moore's law, it has become possible to create even wider machines. Furthermore, it has become clearer that these machines can be used not only to speed up graphical algorithms, but, rather to speed up algorithms that are ***data*** parallel. In fact, the so called cores, known as processing elements(PE), of the GPU are actually clustered into groups of PE that share a program counter, pipe and control unit.

The following points present some of the key architectural highlights of the platform:

- All the PEs in the same control unit execute the same instruction in each cycle(share the same program counter)
- Control units can execute several threads concurrently.
- Memory architecture is NUMA. Each of the clusters has a shared cache.
- The GPU's scheduler is fast, efficient and keeps all the cores utilized when the workload is large enough.
- High memory bandwidth:
    - On chip between PE's and GPU's main memory.
    - Between host(CPU) and GPU.
- Frequency of the systems is in the $500MHz - 700MHz$ region.
- Power consumption of the GPU is considerably high and can reach several hundred Watts.

1.3.2 Tilera

Tilera is a company that offers a wide range of many-core systems. These systems can have 36,64 or 100 cores. Each of the cores can execute OS-like threads and the executed threads may be totally unrelated. This system is meant for executing many sequential threads concurrently. With each core able to execute its own copy of the operating system, the core can be used in servers that handle numerous independent applications. Alternatively, one core runs a full OS while others run a "degenerate" version, making the core more efficient for single-application parallelization.

Some of the architectural highlights of the system are:

- Private caches for each of the cores.
- NUMA hierarchy.
- Fast and wide buses for the memory.
- Thread dispatching to the cores can take several hundred cycles.
- For TLR36480BG-7C[1]
    - 64 cores (tiles).
    - $5.6MB$ on chip cache.
    - Systems frequency is $700MHz$.
    - Power consumption is approximately $23W$

[1] Taken from Tilera website

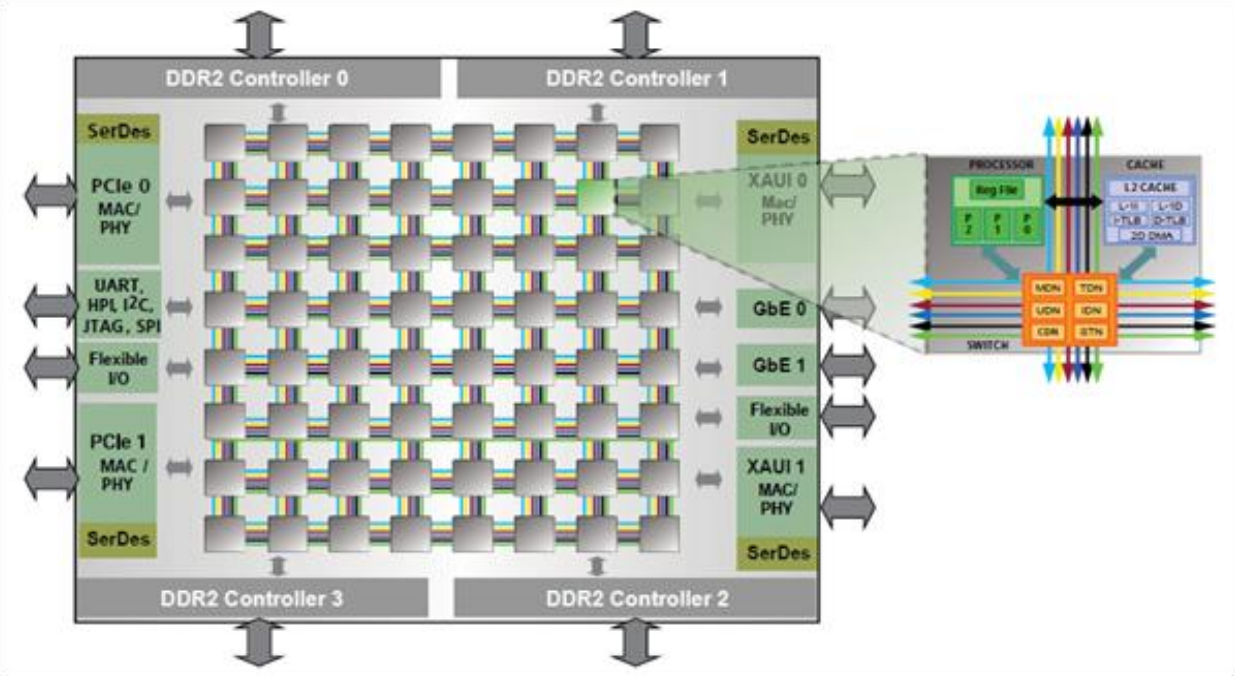


Figure 2 - Tilera'a "TILE" many core system. Image taken from Tilera.

1.3.3 XMT

XMT, Explicit Multi Threading, is many core system that was developed at the University of Maryland under Prof. Uzi Vishkin and his team [1, 2]. This system is an accelerator and is not intended for executing sequential algorithms or for running an operating system.

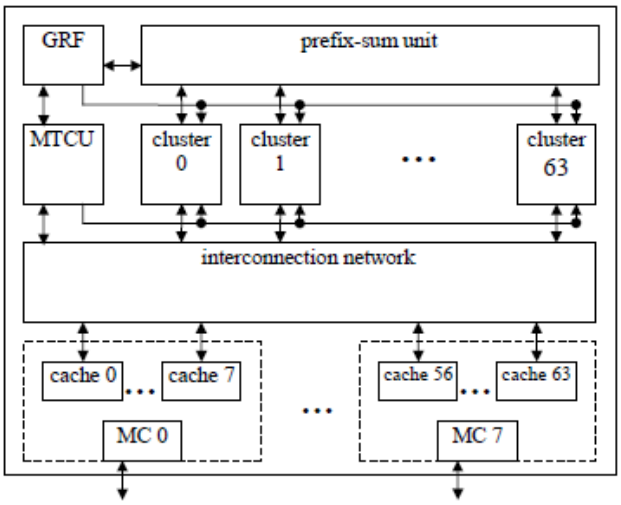


Figure 3 - XMT Architecture. Image taken from Xingzhi Wen's thesis [1]

This architecture has the following properties:

- 64 cores. Each core has a private cache.
- NUMA hierarchy.
- C-like programming language.
- Efficient online scheduler.

1.3.4 Plurality

Plurality offers a shared-memory many-core system. This system supports 32-256 identical cores with shared memory. The shared memory is both logical and physical meaning shared address space and shared physical memory, accordingly. The cores do not have private caches, thus obviating the need for cache coherency mechanisms. Same-address writes are serialized by the interconnect. The shared on-chips cache is partitioned into numerous banks (with address interleaving), and a low-latency, high-bandwidth combinational multistage interconnect carries the core-cache traffic. The memory banks are equidistant from all the cores, so this is a UMA system. The absence of private caches (and a large amount of state in them) and the UMA architecture permit any core to execute any compute task with equal efficiency. This greatly simplifies programming and runtime management.

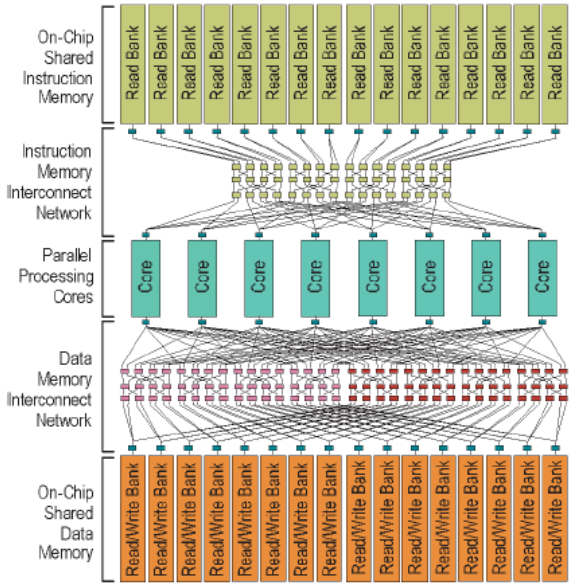


Figure 4 -Plurality's HAL architecture. Image taken from Plurality.

The shared memory offered by Plurality is $2MB - 4MB$. Several memory related issues have to be dealt with:

- Memory foot print –each core uses different memory\data, to avoid thrashing, it is critical that the total amount of memory used by all the cores be smaller than the shared memory.
- Memory access pattern – a bad access pattern can incur cache misses which reduce performance.

Plurality uses a task graph approach to dictate the flow of the application. This is essentially a dataflow graph at the inter-task level: a task may be dispatched once those upon which it depends have been completed.

Plurality has implemented an online hardware scheduler called the "Synchronizer\Scheduler" which is responsible for assigning tasks to cores. This hardware is also responsible for enforcing correct control flow (as expressed by the task graph). The fact that that the Synchronizer\Scheduler is online requires that is be fast in order to ensure that cores utilized as much as possible. To ensure fast scheduling, Plurality created a distributed dispatching network between the Synchronizer\Scheduler and the cores. From the moment the Synchronizer\Scheduler dispatches a task until the tasks reaches an idle core, it takes *O(log(cores))* cycles. The distributed network is a tree with the root at the Synchronizer\Scheduler and the cores as the leaves. The exact number of sub-trees is dependent on the implementation. Each of

the nodes in the distributed network can complete the dispatching in one cycle, meaning that the node sends the dispatch request onwards. Plurality supports two types of dispatching:

1) Dispatching a single task on each sub-tree. This limits the number of dispatched task to the number of the root's sub-trees.
2) Dispatching a single task with numerous copies on each sub-tree. This limits the number of dispatched tasks to a given sub-tree to the total number of cores in that sub-tree. This type of dispatching will become clearer in upcoming sections, but it is worth noting that it permits the dispatching of more tasks than does the first type.

It is Plurality's goal to make this system a low power system. While exact numbers cannot be given as this platform has not been fully synthesized at the date of this publication, the numbers suggest ~4 Watts for 64 cores at 500MHz with 40nm CMOS technology. This is significantly better power-performance than x86 systems.

The Plurality architecture is extremely attractive from both power-performance, applicability and ease of programming perspectives. We therefore chose it as the primary context of our work, though much of our findings and suggestions have broader applicability.

1.4 Shared-memory many-core system challenges

The progression from single core systems executing multiple threads to multi-core and many-core systems that can execute concurrent threads has raised several issues:

- Are the threads related in any way? Data? Code?
- What sort of hardware does the system have? Homogenous\heterogeneous cores? Communication methods? Memory architecture?
- How is work dispatched to the cores? What are the relations between the different tasks in the system? What programming language is used to describe the relations?
- Can the work be divided in a fashion that supplies equal work to all idle cores? How fine grain may the level of parallelism be?
- What scheduling policy should be used? What are the priorities of the tasks?

1.5 Performance evaluation for many-core systems

Following Moore's Law[3], that the number of transistors in a constant sized piece of silicon will double every 18 months, it is now possible to fabricate many core systems on a single chip. Today there are several many core systems in existence, which are different from each other in at least one aspect. These systems include Plurality's HAL[4] many core system, XMT [1, 5] which is a research product from University of Maryland, Tilera's Tile and GpGpu systems by companies such as NVidia[6-8], ATI and Intel.[2]

According to Amdahl's Law[9],the speedup of a single core system is dependent on two parameters:

- f – the fraction of the code that is speed up.
- S – The speedup factor.

Thus, the speedup is written as following:

$$Speedup(f,S)=\frac{1}{(1-f)+\frac{f}{S}}. \qquad (1.1)$$

For a many-core system such that f refers to the code that can be concurrently executed on N cores, the expression from above can be rewritten as:

[2]The purpose and target audience of each of these systems is different and thus these systems cannot be compared in all aspects.

$$Speedup_{Parallel}(f,N) = \frac{1}{(1-f)+\frac{f}{N}}. \quad (1.2)$$

With the rise in the number of cores, comes the problem of having enough jobs to dispatch to the different tasks. Even for programs that are embarrassingly parallel such that the number of different tasks available is greater than the number of cores in the systems, the problem of task dispatching is not a simple one. In a shared memory system, for example, when the memory requirement of each task is considerably large, such that all the required memory cannot fit in the shared cache. Allocating more tasks than the memory can maintain will cause memory thrashing and can cause performance loss compared to the situation wherein only some of the cores are working but there is no thrashing.

For systems in which each core has a cache of its own, cache coherency protocols are required in order to keep the data coherent. These protocols require dedicated hardware and usually reduce performance due to the time needed to execute the protocols.

The granularity of the work can also become an important factor in the execution time of a program. Some systems require thousands of clock cycles to dispatch a task for execution, rendering fine-grain parallelism ineffective and even damaging.

These scenarios cannot be seen directly in Amdahl's Law (1.2), however, in work done by [5, 10-12], in which Amdahl's Law is re-evaluated for parallel systems such that a performance model is created, it can be seen that an increase in the number of cores does not offer linear speedup as core performance is dependent on the core's size.

Furthermore, as each of these platforms is made by a different company there are no standard benchmarks. The programming conventions are different, resulting in code that is not easily ported. This makes comparing the different system difficult as each system optimizes each own benchmarks.

1.6 Scheduling

Theoretical work on scheduling appeared long before many core systems were practical. In [13], Ullman shows that even some of what seem the simplest scheduling problems are intractable NPC (Non-Polynomial Complete). In [14], Graham shows some of the scheduling anomalies that occur when different parameters of the scheduling policy/algorithm are changed. As there is usually a relationship between the tasks in a program it is necessary to have a partial order between the tasks. Partial order scheduling has been part of many research papers in the field of scheduling [13, 15-24]. The use of partial order has spawned the creation of graphs where the partial order between 2 elements can be seen as an edge in the graph. Scheduling of these graphs has been the topic of much research as well [16-21, 23-29].

In the following subsection, a brief overview is given on scheduling and scheduling parameters. While many of these parameters are significant for offline scheduling, scheduling that is done before execution of the program, these parameters are less significant, if at all, for online scheduling where the number of steps that the scheduler can take has to be minimized to increase performance.

Ullman[13] showed that finding an optimal schedule is NP-Complete even in some of the more restricted cases where :

1) All tasks in the schedule are of equal length(i.e. require one time unit)
2) All tasks require one or two time units and there are exactly 2 cores for executing the tasks.

Despite the intractable results shown in [13], it does not mean that all scheduling problems are NPC In order to achieve polynomial time, some of the restrictions must be relaxed. The following properties that will be presented are attributes that a scheduling policy can take into account. When a certain attribute is known ahead of time, it is possible to relax that attribute in the scheduling policy and simplify the process. The restrictions presented here are only are a subset of the full taxonomy as it is presented in [28]. Before the taxonomy is presented, an important assumption is presented – it is assumed that each core can process no more than one task at a given time. Here are the restrictions/relaxations:

- Task information – refers to the data that is known about the task as it will appear in the system:
    - Time that it will take to complete the task. [3]
    - A start time which refers to the time that the task becomes runnable for execution.
    - A due time which is a time that by which the task should be completed.
- Machine data – refers to the data concerning the cores and resources that are used in the system:
    - Number of cores in the system.
    - Types of cores in the system. The type of resources in the system is important as allocating the right task to the right machine becomes important when different cores work at different speeds.
    - Communication Cost – refers to the time that is needed to transfer data or messages of one task to another task. In various scheduling algorithms, it is assumed that if 2 tasks are assigned to the same machine (one after the other) the cost of communication between the two is considered to be zero. Communication cost, is dependent on the architecture of the system (i.e., cache, memory, interconnect).

[3] In the original paper, a single task can have more than one execution time, where each of the execution times refers to the execution time on a specific machine.

- Task characteristics – gives information on the way the tasks are related and on what will occur during the execution of the task.
    - Preemption – refers to if a task can be stopped in the middle of its execution and later continued.
    - Equal task length – are all the tasks of equal length or do the tasks have arbitrary lengths? If all the tasks are equal length, then it is easier to differentiate 2 tasks by its other properties.
    - Precedence relation – refers to a partial order between different tasks. It is also possible that there be no precedence relation between tasks.
- Optimality criteria – refers to the problem the scheduling algorithm is trying to optimize. There are several types of minimizations criterions that can be of interest.

As can be seen from the above there are many combinations for the types of schedules that can be created by relaxing one condition or more. By adding a restriction, for example all tasks are equal length, the problem becomes less generic and it is possible to create a polynomial schedule algorithm for a specific subset of problems. Still, for many combinations of the above restrictions, the problem of finding an optimal solution is intractable. For a comprehensive work on a large set of scheduling problems it is advisable to see Brucker [27].

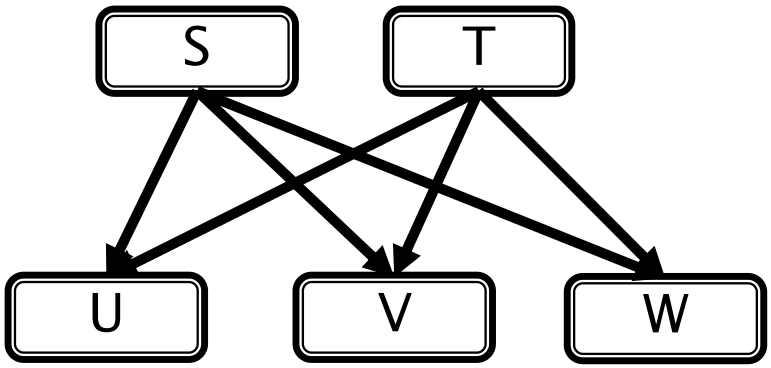


Figure 5 - An example task graph. S, T must precede U, V, W in order of dispatch and U, V, W can not be dispatched until both S, T have completed.

1.7 Online Vs. Offline scheduling

There are many differences between offline scheduling (a.k.a. static scheduling) and online scheduling. As the online scheduling schemes work at real time, they are required to make decisions at a quicker pace than offline scheduling schemes. Offline scheme can take their time and schedule a task to a certain core as it has time to scan the system and find the best match. These operations are time consuming and cannot be done in real-time, as doing them postpones the task dispatching. Online scheduling schemes usually do not have access to the same information that the offline scheduling schemes have access to. As such, online scheduling will give less efficient schedules than offline scheduling.

1.8 Overview

In this section an introduction to shared memory and many-cores was given. Also, some of the challenges of application creation for such systems were presented along with a short introduction on scheduling. This paper deals with adding new scheduling directives that allow for an application designer to better control the scheduling of the application. These directives must be simple for an online scheduler to enforce, as scheduling must be efficient. This paper does not deal with scheduling policies, but rather with the directives that should be enforced by scheduling policies or used by scheduling policies for enforcement.

This remainder of paper is organized as follows. In Section 2, we will describe how it is possible to create workloads that can utilize all the cores, these workloads are called “duplicable tasks”. Also, it will be shown that by using the current scheduling directives is not possible to express a desired ordering of tasks. In Section 3, a new scheduling directive for regular tasks will be presented. Afterwards, the different directives are compared according to their functionality and expressive power. In Section 4, scheduling directives for “duplicable tasks” will be presented. This section will start off by explaining the limitation of regular tasks and the duplicable tasks respectively. It will then be shown how it is possible to enjoy the pros of regular and duplicable tasks by creating new type of scheduling directives. The implementation of these new directives is low power and fast. We will show these directives can be implemented on Plurality like systems. Some of these directives require additional hardware. We will present this hardware and show it is low power, small in size and fast. In Section 5, an observation on inefficient use of caches for shared memory systems will be presented. This inefficiency can cause a reduction in performance of 25%. In Section 6, our conclusions will be presented as well open problems that we believe still need to be solved.

2 Fine grain parallelism for many-core systems

2.1 Introduction

With the increase in the number of cores a system has, comes the problems of creating enough work for all the cores. At first glance, creating the work does not seem like a serious problem; however, there are many issues that need to be taken into account to achieve high utilization of the system. These are a few of the issues that need to be taken into account:

- What is the level of granularity that is desired?
- Can an algorithm be designed in a fashion in that it can be executed in parallel?
- What are the practical implications of the system executing the algorithm on the algorithm itself?
    - Are there atomic instructions implemented in the system?
    - What sort of memory architecture does the system have?
    - What is the size of the memory?
    - How long does it take to dispatch work to an idle core?
    - Will it be possible to utilize all the cores of the system?
    - Does the system support preemption of tasks?

Algorithms in theory might be efficient and show a theoretical linear speedup in the number of the cores. In actuality, the speedup might be as low as 25% of the theoretical result. Explanations for this difference might be:

- Improper load balancing – division of the work is not uniform amongst the cores.
- Bad memory access patterns – can cause thrashing in different levels of the memory architecture.
- Overhead added to the algorithm in the transition from a serial execution to a parallel execution.
- Scheduling overhead and slow dispatching time for different tasks. Especially problematic when dispatching time is longer than execution time of task.
- Inability to adapt the parallel algorithm parameters to suit the hardware platform. For some of the more interesting anomalies due to change in parameters of the hardware or precedence constraints, Graham [14].

2.2 Workload breakdown

Parallelizing a serial algorithm is usually not trivial. Many considerations need to be made and sometimes algorithm simply cannot be parallelized, while others can be parallelized trivially. Algorithms

that can be parallelized trivially are known as embarrassingly parallel problems. In general, parallelization of an algorithm means that workload of an algorithm can be divided into smaller units that compute different parts of the serial algorithm where each of the parts is unrelated (works on different data) and can be executed concurrently with the other parts. The following example shows how a "Matrix Multiplication" is parallelized.

Given matrices $A_{256\times256}, B_{256\times256}, C_{256\times256}$ the algorithm computes $C = A \cdot B$:

$$A_{i,j}, B_{j,k}, C_{i,j} \quad i,j,k \in \{1..256\}$$
$$for\ i = 1\ to\ 256$$
$$for\ j = 1\ to\ 256$$
$$for\ k = 1\ to\ 256$$
$$C_{i,j} += A_{i,k} \cdot B_{k,j}$$

(a)

$$row \in \{1,2,..,256\}$$
$$i = row$$
$$for\ j = 1\ to\ 256$$
$$for\ k = 1\ to\ 256$$
$$C_{i,j} += A_{i,k} \cdot B_{k,j}$$

(b)

Algorithm 1 - Matrix Multiplication. (a) Presents the serial algorithm for matrix multiplication. (b) Presents the parallel algorithm for matrix multiplication

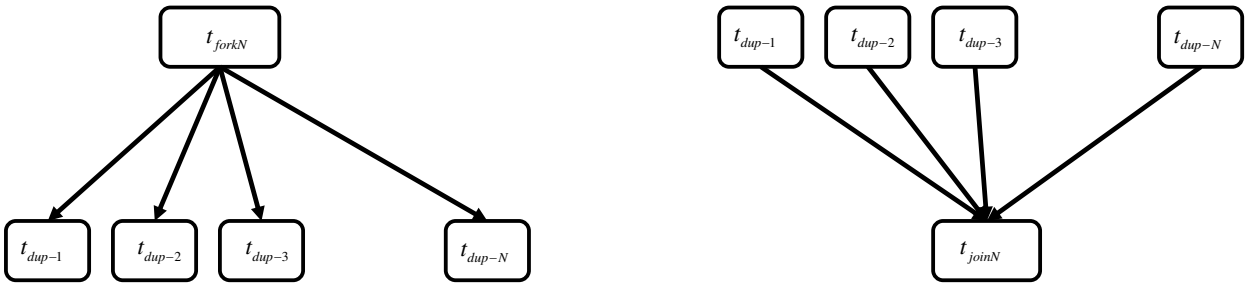
In Algorithm 1 (a), the well known algorithm for matrix multiplication is presented. This algorithm is executed sequentially on a single core. In (b), the workload of the matrix multiplication is divided into tasks, such that each task is responsible for calculating the result of a whole row in the result matrix. Since each row is computed independently of each other, it is possible to execute the different row computations concurrently.

Based on the idea that a workload is divided into smaller tasks, that each task executes the same code on different data is an efficient and simple way to implement parallel algorithms. This idea is called data parallelism and has led the development of architectures, such as NVidia's CUDA, UMD's XMT and Plurality's HAL systems. In NVidia's CUDA this is called SPMD (Single Program Multiple Data) and the tasks are called Kernels, XMT calls this idea Spawning, Plurality calls this a Duplicable task. The names are different, as are the implementations; however, the fundamental concept is the same, divide the workload of an algorithm so that the cores executes the same code on different data.

2.3 Forks and Joins

In the previous section, it was shown how a big task is broken into smaller tasks. In a serial program/algorithm, the single task that is responsible for executing an algorithm becomes runnable at a given time. From a parallel program's point of view, all the corresponding subtasks become runnable at exactly the same time. At this time, all the different sub-tasks can be executed in any order or

concurrently. For the matrix multiplication example, all the rows can begin execution at the same time and the order that the rows are computed is irrelevant given more rows than cores. This operation is depicted as a fork operation that releases all the tasks at the same time and thus it is the scheduler's responsibility to decide which task is dispatched first. In Figure 6 the fork sub-tree is depicted.



(a) Fork sub-tree (b) Join sub-tree

Figure 6 - Fork and Join sub-trees

In a serial program it is obvious when the serial part has completed as the code reaches an "end". In a parallel program the work is divided to different cores, thus, the finishing point is not as obvious. It is therefore necessary to have a synchronization point which is a blocking command that ensures that all the sub tasks have completed before moving on to the next task. This is the join sub-tree (Figure 6).By using both a fork and join on tasks that executes the same code on different data it is possible to create a "*duplicable task*".

2.4 Duplicable Tasks

In the previous sub-sections the idea of duplicable tasks was presented. Basically these are tasks that can be divided into smaller tasks that each of these smaller tasks executes the same code on different data. These smaller tasks are called replicas. In essence, a duplicable task is made up of an algorithm (code) that each of the replicas executes and a number that states how many replicas need to be executed. As we are using Plurality's programming model which defines a duplicable task we also consider their hardware and the duplicable task implementation. In hardware, there are 2 additional fields that are maintained for a duplicable task: the number of replicas that have been dispatched and the number of replicas that have completed. The difference between these two fields gives the number of active replicas. The pros and the cons of duplicable tasks relative to conventional ones are as follows.

Pros:

- Changing the number of replicas in the task graph is simple. All that needs to be done is to change the number of replicas parameter for that task.

- Task graphs that contain large number of nodes/tasks can be created by using duplicable tasks because of the ease of definition of a duplicable task.
- Task maps are more readable and easier to understand.
- It is possible to efficiently dispatch duplicable tasks to idle cores (this will be shown).
- Larger task graphs become easier to design.

Cons:

- Scheduling-related expressive power is lost because all the replicas of the task are dependent on the fork and the join is dependent on all the replicas of the task:
    - All the replicas must wait on the fork before they can begin execution.
    - The task/s following the join must wait for all the replicas to complete before they can begin execution.
- The completion of the replicas is out of order. Causes for this include replicas of different lengths (data dependent) as well as non-deterministic memory access time (cache misses).
- Architecture dependency – Optimizations done to a specific graph will most likely be system dependent, and changing systems may even render them damaging [14].

The focus of this work is to try and overcome the scheduling limitations of duplicable tasks while retaining the simplicity brought about by them. This research focuses on how to improve the expressive power that will be given to the task graph designer. This will be done by presenting new scheduling directives that we believe to be useful for designing task graphs and allow the task graph designer to insert what he/she believes to be crucial input to the runtime scheduler. Also, the practicality of implementing them will be considered.

2.5 Expressive power limitations due to precedence constraints

In this subsection, several examples will be given that show that using only precedence constraints is not enough to express the needs and desires of a task graph designer. In subsequent sections, several new scheduling directives will be presented that help overcome these problems. The first of these examples will show the limitation of precedence constraints between regular tasks. The second example will show how precedence limits the relationships between duplicable tasks.

2.5.1 Example of limited expressive power - regular tasks

The first of these examples is depicted in Figure 7 - Example that precedence is insufficient even for same length tasks. (b) & (c) both depict schedules.(a) where the schedule can initially select $S, T\ or\ A$ to dispatch to an idle core. Given a two-core system, it is possible to select two of these three tasks. In (b), $S\ and\ A$ are selected prior to T. While in (c), $S\ and\ T$ are selected prior to A. The execution time of (c) is smaller than that of (b) because $U, V\ and\ W$ can be executed concurrently to A and cannot be executed concurrently with T. So the question that is raised, can adding a prerequisite between (S, A)

and (T, A) (in Figure 8) help fix this problem? The answer to this is equivocal. On the one hand, given the two-core system, it is possible to enforce the desired schedule. On the other hand, using the new graph on a three-core system (graphs are created offline, this can be thought of a compilation operation), then a penalty in the execution is incurred compared to an optimal scheduling of the original graph on the new system.

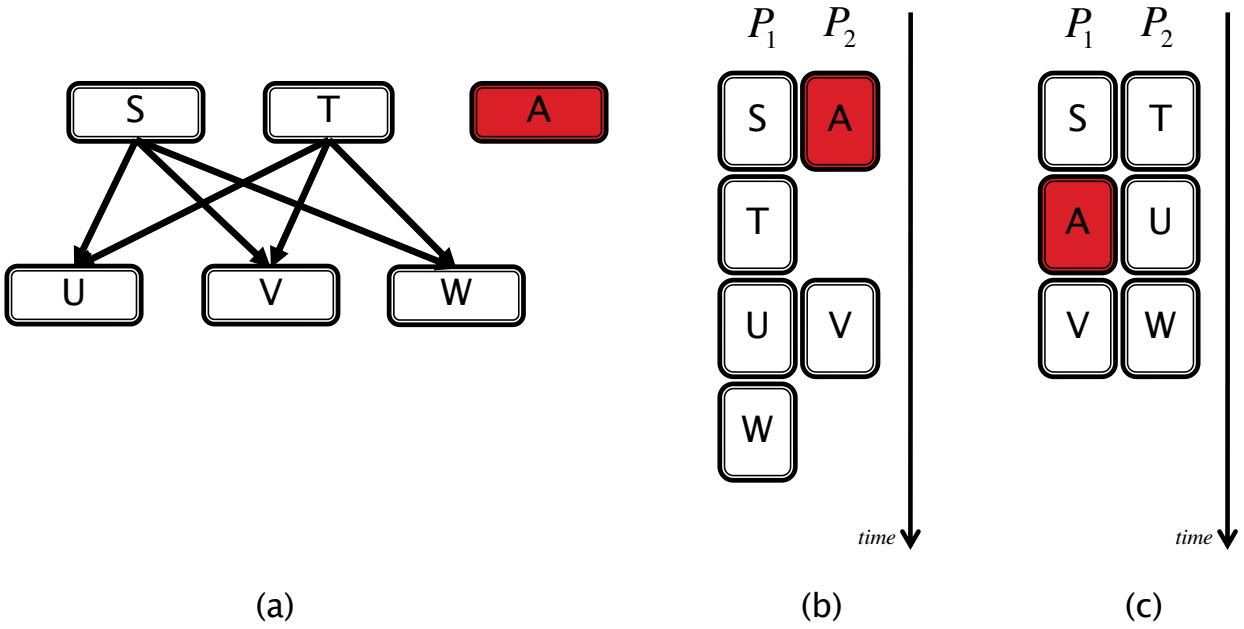


Figure 7 - Example that precedence is insufficient even for same length tasks. (b) & (c) both depict schedules.

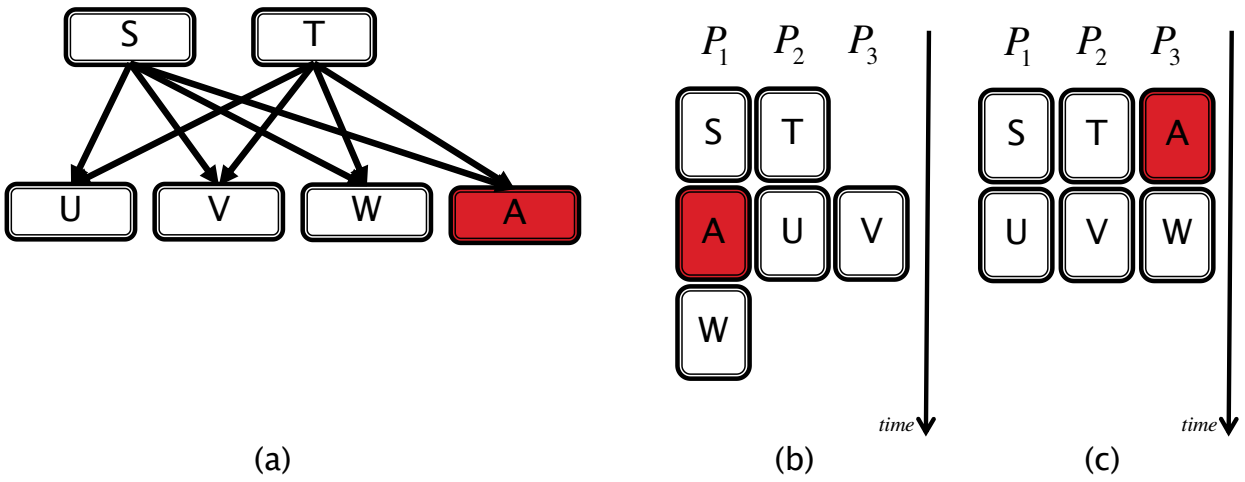


Figure 8 - Updated graph with additional precedence constraints. (a) S and T precedes A. In (b) the graph from (a) is scheduled on a new system that has an additional core. (c) Scheduling of the original graph on the new system.

This leads to the conclusion that precedence constraints can limit expressive power.

2.5.2 Example of limited expressive power - duplicable tasks

The use of precedence constraints for duplicable tasks has two major implications:

1) All the replicas become runnable at exactly the same time, when all the prerequisites of the duplicable task have completed.
2) A duplicable task is considered completed only after the last of the replicas completes.

These implications are depicted in Figure 9. Each of these implications has its pros and its cons. While these pros and cons are important[4], the following will only show the expressive limitations precedence constraints w.r.t. to duplicable tasks. Given two duplicable tasks A, B such that A precedes B. According to the second implication, each replica of B is dependent on the completion of all the replicas of A, in essence this Figure 10 (a). While in actuality the task graph designer may know of a finer grain relationship such as the one presented in Figure 10 (b). Scheduling the graph in Figure 10 (a) will require that all of $A's$ replicas complete, Figure 10 (c). Scheduling Figure 10 (b) (if it were possible to express the desired relationship), would result in a shorter execution time, Figure 10 (d). In the following section we will show that it is possible to express such a relationship.

[4] The pros and cons will be further elaborated upon in future sections.

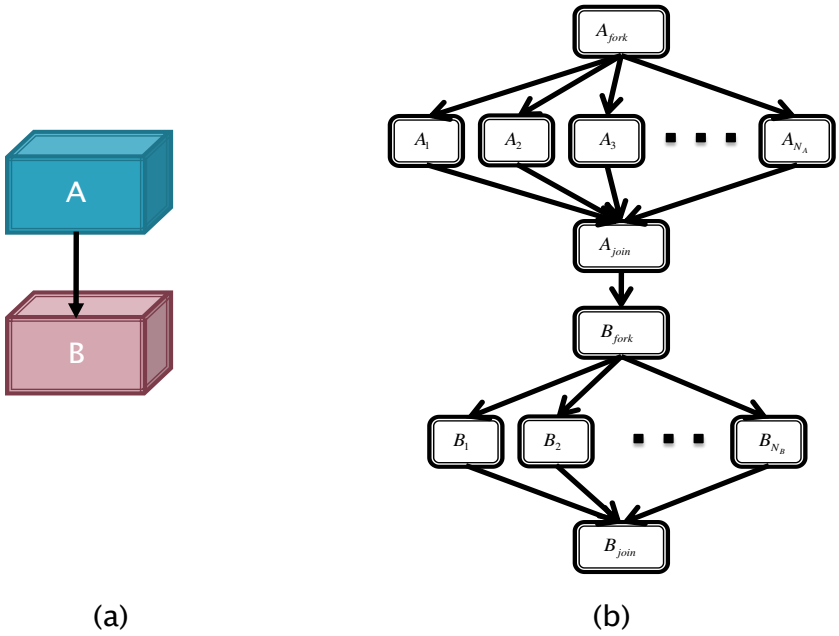


Figure 9 – (a) A and B are duplicable tasks. A precedes B. (b) Equivalent graph to the one presented in (a).

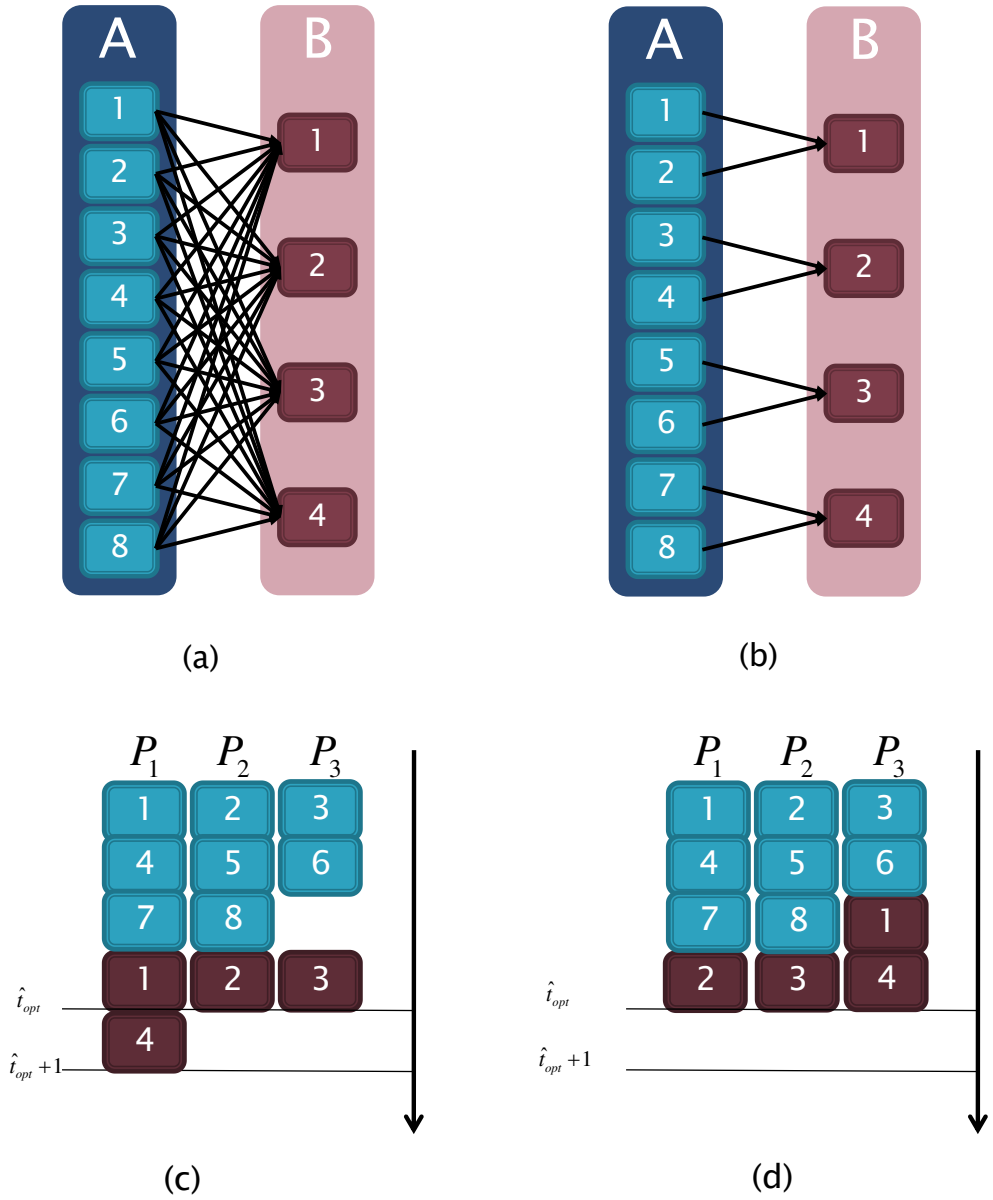


Figure 10 - Example of limited expressive power when using duplicable tasks. Using duplicable tasks adds precedence constraints that don't exist (due to duplicable task syntax). (a) A, B duplicable tasks. A precedes B. (b) A, B duplicable tasks with a new relationship (directive) between the replicas of B and A. (c) Gantt graph of (a), assuming equal length tasks. (d) Gantt graph of (b), assuming equal length tasks.

2.6 Duplicable-task scheduling

Scheduling duplicable tasks is a tougher problem than it may seem at first glance. Given a task t that has N replicas, any random permutation of the N replicas is a legitimate ordering of the replicas. Scheduling a random permutation of the replicas can have implications on the execution length. Also, randomization can slow the scheduling down as a history of which replicas have been dispatched and which haven't needs to be maintained and updated. The following presents some of the problems related to scheduling of duplicable tasks:

- Data dependant replicas – there are cases where the execution length of the replicas are data dependent. A special instance of this problem is when the longest replica is dispatched last, thus, increasing the schedule length of the duplicable task.
- Shared memory – when the memory of the system is shared by all the cores and its size is limited, an irregular ordering of the task might cause an increase in memory misses in the shared memory (cache) due to lack of data locality which hurts the performance.
- Optimality – if the scheduler used is based on a random algorithm, it is unlikely that the optimal schedule will be reached.
- Join synchronization point – the join of the replicas becomes a bottleneck on the releasing of other tasks.

A result of the aforementioned reasons makes scheduling the replicas from the lowest ID to the highest ID a good and simple approach. This approach allows for quick scheduling of the duplicable tasks as the order of dispatching "inside" the duplicable task is both deterministic, simple and "chunks" of replicas can be allocated efficiently. System's like Plurality's encourages scheduling duplicable tasks in sequential fashion as this increases the number of tasks that can be dispatched in every cycle.

Important Assumption

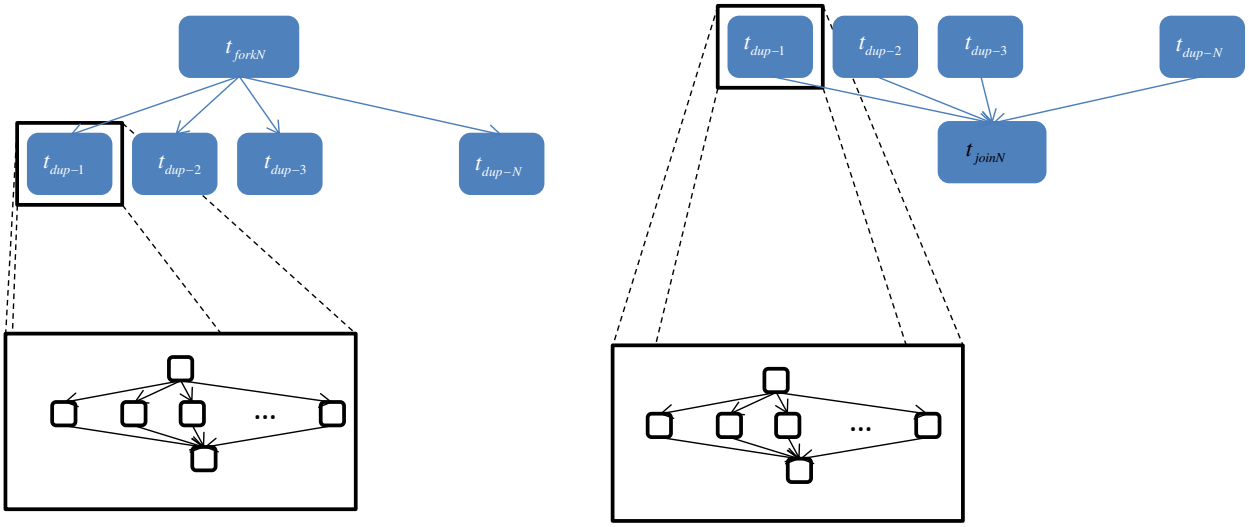
As of this section, it will be assumed that scheduling of the replicas of a single duplicable task is done in a sequential fashion, from the replica with the lowest ID to the replicas with the highest ID.

2.7 Different levels of granularity for duplicable tasks

In this section the concept of duplicable tasks was presented, as were some of the pros and the cons of duplicable tasks. Another type of task was presented as well - regular task. It might be inferred that these are the only two types of task that can be used when creating a task graph. However, it is possible to get a third level of granularity that is in between the two and that is to create several duplicable tasks that execute the same algorithm on different data by dividing the duplicable task into smaller duplicable tasks that each new duplicable tasks has fewer replicas than the original. This might be seen as

duplicable task of duplicable tasks, that is, a fork that releases a set of duplicable tasks and a join that waits on the completion of all the duplicable tasks. Although the difference between Figure 6 and Figure 11 does not seem great, the difference is that each of the tasks in Figure 11 is a duplicable task, so each of the tasks is made up of a fork and a join while in Figure 6 each task is a regular task. Also, notice that new graph does not have a join that follows all the smaller duplicable tasks. This join is optional.

By using this new type of granularity, which is between coarse grain (regular tasks) and super fine grain (duplicable tasks), it is possible to express new types of relations between duplicable tasks. Some of these relationships allow better expressing of the needs and the needs of the task graph designer. For example, each of these smaller duplicable tasks is a perquisite to another task, each of these following tasks can be dispatched upon the completion of the smaller task rather than waiting for the completion of all the replicas.



(a) Fork sub-tree (b) Join sub-tree

Figure 11 - Fork and Join sub trees for duplicable tasks.

We will show that there is a way to achieve a similar level of granularity without actually breaking the duplicable tasks into smaller duplicable tasks. In section 4 of this paper a way to achieve this finer grain of duplicable tasks will be presented, but, without paying the penalties of regular tasks and duplicable tasks. Beforehand, scheduling directives for regular tasks will be presented.

3 Scheduling directives for regular tasks

3.1 Introduction

In 1.6 the concept of scheduling based on graphs was first presented. This concept has been the topic of much research. Many of these approaches use the list scheduling approach [15, 16, 18, 24, 26]. In list scheduling each node is assigned a priority and a list is created by sorting the tasks according to their priorities. It is common practice to sort the tasks from the highest priority (head of list) to the lowest priorities (tail of list). Once the list has been sorted, the following repetitive algorithm is used to schedule the list:

1) Select (and remove) task from beginning of list. Task with highest priority is selected.

2) Select runnable resource to execute task.

These two stages are repeated until list is empty.

The criteria on which the priorities of the tasks are decided, is called the scheduling policy. The scheduling policies should try to optimize some performance measure while enforcing correctness (precedence constraints). The problems that arise from using these algorithms is that task graph designer does not have a straight-forward ability to state which of task is considered more important as priorities are not designated in the task graphs but rather by the scheduling policy.

It should be noted that priorities do not override precedence, but rather help select a task from the list of runnable tasks based on its level of importance. Priorities can be a useful tool in the hands of a task graph designer as the designer is aware of algorithm constraints and the optimization criteria(which may not be one of criterias as presented in [30]). Also,each of the tasks receives a unique priority. In this paper, it will be assumed that the priority is given by the task graph designer(i.e. offline), though this priority can also be given by a scheduling policy, and the priority will be enforced online by selecting the task with the highest priority from the list of runnable tasks.

3.2 Start After Start

In this section a new scheduling directive for regular tasks will be presented. This directive is known as Start After Start, SAS for short. Earlier in this paper, an example was shown that precedence constraints are not enough for expressing everything that is desired. Precedence states, that a task is not runnable for execution until all of its prerequisites have completed. What if all that is required is that the same task only wait until all these so called “prerequisites” have been dispatched? This cannot be expressed simply using precedence. It is the goal of SAS to achieve this. This section starts off by presenting motivation for this directive, after which an implementation of this directive is shown.

3.2.1 Motivation for Start After Start

Example 1 : Insufficiency of precedence

Consider the graph in Figure 12, where: $p_v > p_u$ and $w(v) < w(u)$. From the graph in Figure 12, it is depicted that U depends only on T, while V depends on V, S_1. When T completes, U is runnable for execution. If S_1 has yet to complete, U will be dispatched. This will result in the schedule depicted in Figure 13 (a). However, if U can be blocked until the dispatching of V, even leaving a core idle for a while, it is possible to improve the execution time as depicted in (b). Blocking U until V has been dispatched is exactly SAS.

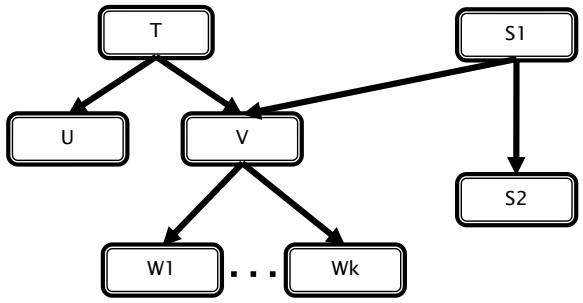


Figure 12 - Motivational example for SAS

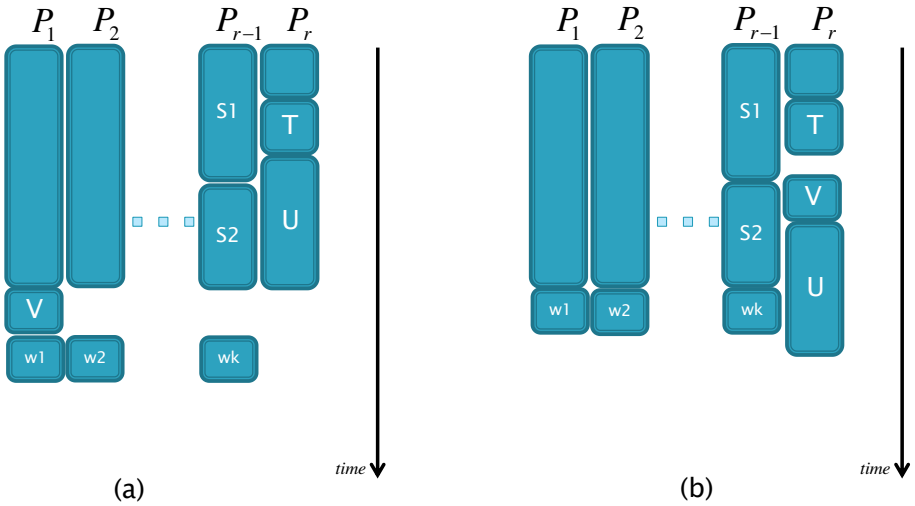


Figure 13 – Scheduling of a task graph where priority and precedence is not enough to achieve optimal execution time. (a) U is dispatched before V. (b) SAS is used. U is blocked from execution until the dispatching of V. Execution time of (b) is better than the execution time of (a).

Example 2: Reducing number of edges in graph.

Consider graph (a) in Figure 14 where S, T are prerequisites for U, V, W such that $p_u > p_v > p_w$. A corresponding graph is depicted in (b), where V can not start until U starts and W can not start until V starts. V cannot be dispatched until all of $U's$ prerequisites have completed which are also V's prerequisites. The difference between the graphs, is that the graph in (b) enforces a dispatch order.

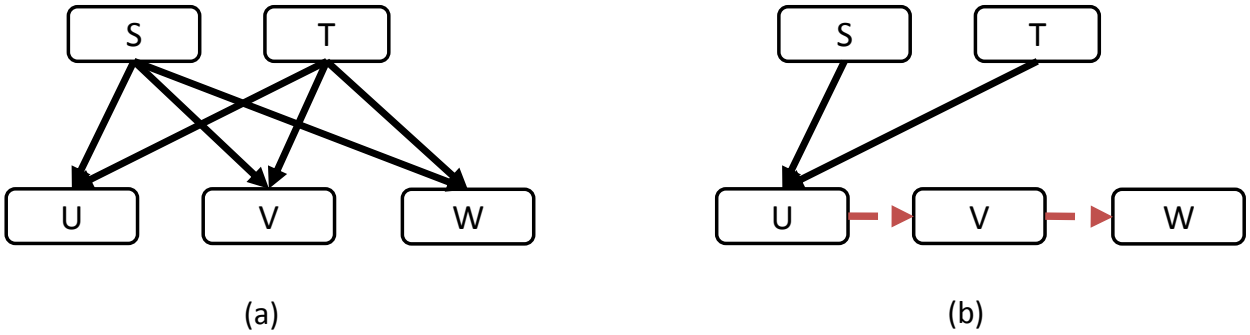


Figure 14 – Two equal task graphs (a) Graph based only precedence. (b)Graph based on precedence and SAS.

3.2.2 Implementation of SAS directive

Implementing this primitive can be done using only precedence constraints and using priority, it is not possible to implement this primitive without one of them:

- Without using priorities it is not possible to state which of the tasks is more important than the other and so the scheduler can decide which of the tasks to dispatch.
- Without using precedence constraints, as soon as one of the tasks is runnable the scheduler can schedule it and as will be shown, a common dependency is required.

By adding a unique priority level to each task it, it is possible to implement this construct. In Figure 15, two implementations show how to implement the SAS primitive:

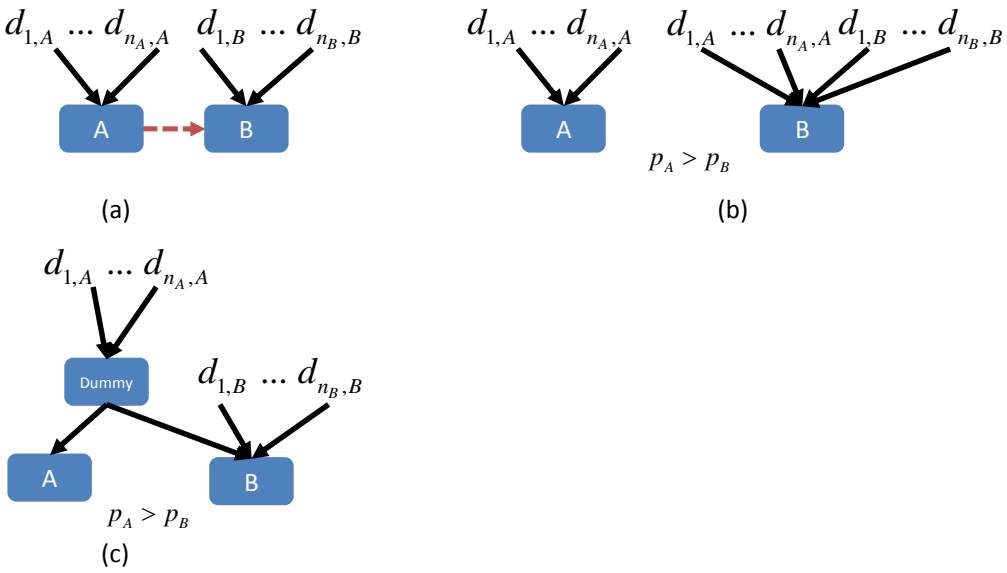


Figure 15 - SAS Implementation (a) Desired graph (b) First implementation, adding $A's$ precedences to B. (c) Adding a dummy task that receives all of A's precedence. B is dependent on $Dummy$ as well.

It is should be noted that $p_A > p_b$. Without this knowledge in both (b) and (c) of Figure 15, if A and B are both runnable, they are both viable options by the scheduler. The implementation of this primitive is based on adding A's dependencies to the already existing dependencies B has. This will ensure that B does not become runnable at least until A does . The addition of the priority, ensures that A will be dispatched before B if both are runnable.

	First implementation (b)	Second Implementation (c)
Edges Added	n_A	2
Vertices Added	0	1
During scheduling: How many additional updates are needed when completing a task A?	n_A	2

Table 1 - Comparing SAS implementations

From Table 1 it is possible to see that the second solution is cheaper in memory as it uses as fewer edges[5] and during the scheduling, only $Dummy$ needs to be informed of its preceding rather than A *and* B in the first implementation.

[5]It is necessary to compare the memory size of a vertex to that of an edge to be certain which of the approaches is cheaper in memory.

3.3 Regular tasks scheduling directives summary

In summary, three types of scheduling between regular tasks were seen. The following will present these directives according to their expressive ability, from the strongest to the weakest:

1) Start After Complete – this is another name for precedence constraints which states that certain tasks cannot be dispatched until all their prerequisites have been completed.
2) Start After Start – certain tasks cannot be dispatched until other tasks have been dispatched before them, even if a core remains idle as a result.
3) Priority – allows selecting a task from a set of tasks that can be dispatched.

4 Scheduling directives for duplicable tasks

4.1 What are scheduling directive for duplicable tasks?

A scheduling directive, for duplicable tasks can be thought of as a special relationship between replicas of two or more duplicable tasks such that replicas are no longer dependant on the completion of all the replicas of another duplicable task. For example, the i-th replica of B is dependent on the $2 \cdot i$ and $(2 \cdot i + 1)$ replicas of task A. In other words, it is possible to dispatch B_i after $A_{2 \cdot i}$ and $A_{2 \cdot i+1}$ have completed(such a relationship might be useful for Merge-Sort[31] or parallel addition of a vector). An example of this was shown earlier in section 2.5.2 (Figure 10) and another example is given here as well Figure 16.

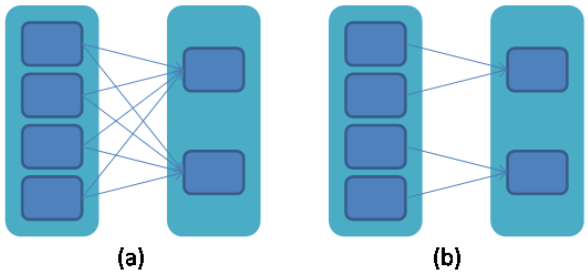


Figure 16 - Two duplicable tasks and their dependencies. (a) Merge-Sort implemented by duplicable tasks (b) A desired Merge-Sort implementation by duplicable tasks.

There are other considerations (not including, specific replicas becoming runnable earlier) that make scheduling directives desirable. These include, placing expressive power in the task graph designer hands so that he can create a more accurate task graph due to the finer-grain granularity given to him. The benefits and requirements of each directive will be discussed in the sub-section of the directive. There are certain constraints on the new directives:

- Efficiency – dispatch numerous replicas in a small number of decision steps like duplicable tasks.
- Fast – currently dispatching a replica takes $O(\log(Core))$requires a small number of cycles. Thus, it is desirable to stay with-in this time limitation and so the new directives should try and support this limitation as well.
- Low power – As the system that we are using is a Plurality-like system, there is a power constraint that needs to be meet. Any new hardware that is implemented for such a system should be low power.

Before presenting the new directives, consider the following example, given A, B duplicable tasks. Both A and B access the same data array X using the same access pattern[6]. X is exactly twice the size of the shared memory . If all of A is executed prior to any of $B's$ replicas, then the shared memory will first contain the first half of X and then this will be removed and the second half of X will be fetched. As B has the same access pattern, this will happen also for B. Both tasks will suffer from the cache misses. Rather than executing B after A, by executing $B's$ replicas concurrently(not necessarily simultaneously) to $A's$ in a way that A cannot advance enough to clear the cache, it will be possible for B to use data that has already been fetched(following A's cache misses) and that is in cache. In this example, we see a speedup that is a result of the reduction in the number of cache misses. While this idea offers a speedup due to a decrease in the number of cache misses, we will see that this idea is also parameter dependent.

In following subsections we will present directives which we have found to be interesting. While there are many interesting relationships, some are impractical as they do not allow holding up the constraints that were presented. Consider relationships that are sporadic or semi-sporadic which means that it is hard to define a regular relationship between the replicas of the duplicable tasks, such as the one depicted in Figure 17 . For such scenarios, it might not even be worthwhile to define a relationship as the overhead or implementation might be expensive. In this paper we not deal with such random relationships.

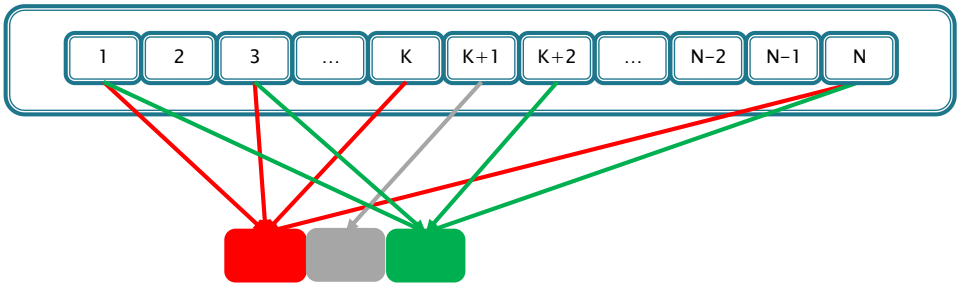


Figure 17 - Sporadic relationship

At first glance, there seem to be several obvious workarounds for creating duplicable task relations. These include using regular tasks instead of duplicable and the fusion of two duplicable tasks into one. While each of these workarounds offers a solution to the desired problem, they create new problems that need to be dealt with. In the following two sub-sections, these workarounds will be reviewed and it will be explained why they are not the desired solution.

[6] If A_i accesses an address X_j, then B_i accesses the same address X_j.

4.2 Regular tasks are simply too much

An obvious question that can be raised following the previous sub-section is: Why use duplicable tasks when regular tasks can easily do the job? In essence, replace N replicas of the duplicable task with N regular tasks. This solution is not good for several reasons:

- The task graph will bloat up in size.
- Every time that application parameter changes (for example the dimensions of a matrix in a matrix multiplication operation) the number of regular tasks needs to be updated, in comparison with simply changing the number of replicas of the duplicable task.
- In actuality, the size of the graph is limited by the hardware. Increasing the size of graph is not feasible.
- Dispatching regular tasks is less efficient than the duplicable task. This is because the scheduler can dispatch several replicas of a duplicable task in a single cycle.

Following these explanations it is possible to see that switching the duplicable tasks to regular tasks simply doesn't work.

4.3 Task fusion is not feasible

In this sub-section, the answer to the following question will be given: Given two duplicable tasks A, B, why not fuse these two tasks into one new duplicable task C? Task fusion seems almost like the intuitive approach to solving this problem. As the task graph designer knows the relationships between the tasks, the task graph designer redesign the graph accordingly. However, task fusion suffers from several deficiencies and problems:

- Concatenation of two functions into a single function is not desirable from a software point of view as this reduces code reusability.
- Duplicable tasks may not be the same length.
- Considerably simple relationships are problematic. For example, given that B_i is dependent on A_i and A_{i+2} , which replica of C, C_j should compute A_j ? Should C_j or C_{j+2} compute it?
    - Each of the C_j replicas will compute A_i and A_{i+2}. This causes redundancy in operations. While this approach is still sufferable for this scenario, for cases that B_i is dependent on more values, this approach is insufferable.
    - Let C_j, compute A_{j+2}. However, do to out of order completion, C_{j+2} which is dependent on C_j cannot be dispatched until the completion of C_j.

While task fusion might be suitable for some problems, it is not suitable for many.

4.4 What sort of directives do we want?

Earlier in this section, the need for more intricate relationships between duplicable tasks was presented. After this, two approaches to implementing the relationships were presented. While both approaches were found wanting, they approaches did not discuss what the relationships would be but rather how to implement the relationships once defined. In this sub-section, we will try and answer the question: What relationships do we want? Answering this question will help us define the new scheduling directives. We would like relationships that answer some of the following criteria:

- Correctness – Can a certain order of execution be enforced when using the directive? The order of execution must be correct for all scheduling instances and not for a single schedule. For example, if both replicas access the same memory address, can an exact order of the replicas to the data be enforced? Another example, if there is a dependency between replicas of the tasks, can this dependency be enforced?
- Fairness – Does the primitive allow sharing resources in a fair manner between the duplicable task? The following two are criterions interest us:
    - Number of cores – Do the tasks use the same number of cores? This infers to equal amount of computation time for the tasks.
    - Dispatching – Is it possible to dispatch replicas from both duplicable tasks such that neither of them is starved?
- Cache issues – By using the directive is it possible to improve performance of the memory?
    - Total memory required - By using the primitives is it possible to reduce the total amount of memory required at a given time?
    - Cache misses – is it possible to reduce the number of cache misses required by the tasks?
- Efficient scheduling –
    - Can the relationship be calculated quick enough to allow for fast and efficient scheduling?
    - Can scheduling work be done background (when there are no idle cores)?

Answering these questions is not simple for the general case as many of the answers here are dependent on the parameters of the application and the selected directive. It will be shown that some of the directives do not support all the criteria, while in actuality some of criteria conflict with each other. Motivational uses will be presented for each directive.

Before proceeding to the scheduling directives, the following observation on memory usage will be presented. While, at times the exact memory access pattern is not known, it is still possible to present an upper bound on the total amount of memory that is required.

Definition: Let $Mem(rep_{r,A})$ be the memory used by the $r-th$ replica of A.

Given a duplicable task A that has R replicas currently executed in the system, the upper memory bound (UMB) is:

$$UMB = \sum_{r \in R} Mem\left(rep_{r,A}\right).$$

For duplicable tasks that all the replicas have the same access pattern and use the same amount of memory, it is possible to get a closer approximation:

$$UMB_{similiar} = R \cdot Mem\left(rep_{r,A}\right).$$

There are also scenarios in which the replicas access some of the memory that other replicas access. For this scenario a tighter bound can be given:

$$UMB_{shared} = \bigcup_{r \in R} Mem\left(rep_{r,A}\right).$$

It should be noticed that all these expressions discuss the memory that is currently being used. However, this does not take into account the out of order completion such that different replicas in R may not be consecutive, in which case the expressions should be changed to support $\hat{R}$. $\hat{R}$ is defined as following:

$$\hat{R} = \left\{ r \middle| \quad \min\left(R\right) \le r \le \max\left(R\right) \right\}.$$

These expressions become slightly more complicated when there is more than one duplicable task involved as another memory access pattern needs to be considered. First of all there are two sets of executed replicas R_A,R_B. Also, this access patterns needs to be considered from two angles:

- Memory usage – Do the duplicable tasks use the same memory?
- Access pattern – Do the duplicable tasks have similar access patterns?

Do the duplicable tasks use the same memory?

- If they use the same memory, another question needs to be answered. Do they have a similar access pattern?
    - If they do, the size of $\hat{R}_A$ will be defined by the number of dispatched replicas that it leads B by. This is because, eventually B will need to access this memory and it is preferable that this memory not be removed from the shared cache.
    - If there is no similar access pattern, then as far as we are concerned the relationships is sporadic and this is of no interest to us.
- The tasks do not use the same memory. Do the tasks need the same cache lines at the same time? If so, this can cause memory thrashing.

4.5 Priority Primitive for Duplicable Tasks

Adding priority primitives to duplicate tasks is slightly more complicated than it is when used just on regular tasks. For a single duplicable task, setting priorities is easy due to in order allocation. The task with the smallest id receives the highest priority and the task with the largest priority receives the lowest priority.

Each replica has two priorities, external and internal. The external priority is the priority of the duplicable task. The internal priority is the priority that replica has over the other replicas in the duplicable task, which is basically its position (index) in the duplicable task. In essence the priority of a single replica depicted as a 2-tuple made up of the external priority and the internal priority (p_e, p_i).[7]

The following is based on combinations between regular and duplicable tasks:

- Given a regular task r and a duplicate task d with priorities p_r and p_d, respectively, there are several ways to schedule the tasks,
    - $p_r = p_{d,e}$. As the priorities must be unique, the internal $p_{d,i}$ priority is checked and accordingly one of the following will happen:
        - All the replicas with a $(p_{d,e}, p_{d,i})$ higher than p_r will be schedule before r. After this r will be dispatched, followed by the remaining replicas.
        - r is dispatched before all the replicas of d This situation is unlikely because in actuality $p_r > p_d$. It would be wiser to set the priorities accordingly so that efficient dispatching will be possible.
        - All the replicas of d are dispatched before r. This situation is unlikely because in actuality $p_d > p_r$. It would be wiser to set the priorities accordingly so that efficient dispatching will be possible.
    - $p_r > p_d$ – r is dispatched before a single replica of d is dispatched. This priority primitive is useful the task graph designer that knows that a certain regular task takes more time than each replica of d. This will allow, running several and different replicas of d on the same core while r is executed on a different core.
    - $p_d > p_r$ – only after all the replicas of d have been dispatched, can r's replicas be dispatched.

  It should be noted that different priorities for r and d can change the latency of the program.
- Given two duplicable task d_1 and d_2 with priorities p_{d1} and p_{d2} respectively, there are several ways to schedule the tasks:
    - Both d_1 and d_2 have the same priority – both tasks (and their replicas) can be executed concurrently and replicas can coexist side by side and so the internal independence of

[7] Out of this 2-tuple it is possible to create a singleton number which can be seen as the absolute priority of the replica. This allows giving each replica a unique priority.

the duplicable tasks decide which replica is dispatched first. This resembles Start After Start, but, for duplicable tasks.

- One of the duplicable tasks has a higher priority (without the loss generality, $p_{d1} > p_{d2}$), in which case all of its runnable replicas will be executed before the runnable replicas of the other duplicable tasks.

We did not deal with directives between regular tasks and duplicable tasks in this paper.

4.6 Applicable directives for duplicable tasks

In this sub-section new scheduling directives will be presented. The scheduling directives that will be presented in this sub-section can be implemented using existing data that is kept for each duplicable task. As a quick reminder these fields are given in the following table.

Variable	*Name*	*Type*	*Description*	*Is currently supported*
Number of replicas	n	Int	Number of replicas that need to be allocated.	Yes
Dispatched replicas counter	s	Int	Counts how many replicas have been dispatched/started.	Yes
Completed replicas counter	c	Int	Counts how many replicas have completed.	Yes

Table 2 - Mandatory information for duplicable tasks

A duplicable task is not considered complete until $c = n$. Also, it is worth noting that replica allocation is done in-order from the replica with the lowest id until the replica with the highest id.

Corollary: Given a duplicable task A, $A.s \geq A.c$ at all times. This is due to the fact that a replica can't be completed before it has been dispatched.

In the following subsection these notations will be used given a duplicable task A to state if A_i has started or completed:

- $S(A_i)$ –returns true iff A_i has started.
- $C(A_i)$ –returns true iff A_i has completed.

4.6.1 Start After Start for duplicable tasks

Start After Start, SAS for short, was introduced earlier in this paper in the context of controlling the order in which regular tasks can be dispatched. This idea comes up for duplicable tasks as well. Basically, the question that is asked, is there a way to state that certain replicas can start only after certain replicas from other task have started? The answer to this is: yes, there is a way to state that certain

replicas are dependent on the replica dispatching from a different duplicable task. Given two duplicable tasks, A and B, the replicas of B are dependent on the dispatching of A's replicas and the replicas of A are dependent on the dispatching of $B's$ replicas. If the latter had not been stated, it would be possible to dispatch all of B's replicas without dispatching a single replica of A.

In the previous sub-section, when priorities were discussed, it was stated that each replica has a priority. It is undesirable that at each dispatch the scheduler compare priorities of the task as this will reduce efficiency. It is the job of this directive to do this. The directive suggests that A will lead in the number of replicas dispatched. Initially it might seem that all that is required is a number l that states how many more replicas of A have started before replicas of B. However, the use of a single number will cause a single degenerated scheduling as will be explained, is based on the priority level alternating between the two tasks. To overcome this problem, two numbers will be used: an upper boundary and a lower boundary.

Name	Type	Descriptions
A	Task	The first task.
B	Task	The second task.
l_{upp}	Integer	Maximum number of dispatched replicas which A may be ahead of B.[8]
l_{low}	Integer	Minimal number of dispatched replicas of A may be ahead of B, except for the initial state until where fewer than l_{low} replicas of A have been dispatched.

Table 3 - SAS parameters

Remark:

Recall that A has a higher priority than B. Therefore, whenever replicas of both tasks are dispatchable from a precedence constraint perspective, A's replicas will be preferred over B's as long as A leads by at most l_{upp} . If A's replicas are dispatchable and B's are not, A can lead up to l_{upp} .Similarly, if B's replicas are dispatcheable, but A leads by fewer than l_{low} than none of B's replicas will be dispatched.

To better understand the degenerated case, consider, the case when there is one boundary, l. In this case, A will dispatch l replicas before B can dispatch a single replica. After this, B can dispatch a single task without violating the terms of the directive which states that A leads by l replicas. After this, A can dispatch a single replica. This situation of alternate dispatching will repeat itself until both tasks have completed. Several downsides of this include the fact that only a single replica can be dispatched per scheduling move/decision. Because of the desire to be able to control the size of the task dispatching another parameter is needed. This parameter will be the lower boundary.

[8] It will be assumed that $l_{upp} \geq l_{low} > 0$.

For any given duplicable task, the scheduling will be sequential (in order), which states that, for a specific replica i, i cannot be dispatched until all the $(i-1)$ replicas prior to i have been dispatched. For task A, A cannot advance more than l_{upp} replicas than task B, due to this, A_i is dependent on the starting (dispatching) of $B_{i-l_{upp}}$, this is depicted in (4.1) by placing a SAS constraint from to $B_{i-l_{upp}}$ to A_i. This ensures that the upper boundary is not violated by letting A run ahead. Task B will be behind task A by a number that is in the range (l_{low}, l_{upp}). This is done by placing a SAS constraint from $A_{i+(l_{upp}-l_{low})}$ and B_i, this ensures that the lower boundary is not violated and that B doesn't run ahead, the formal definition of this in (4.2). Now that the two boundaries for this directive were presented, it should be noted that by selecting $l_{upp} = l_{low} = 1$, it is possible to recreate the situation that was described for the single parameter.

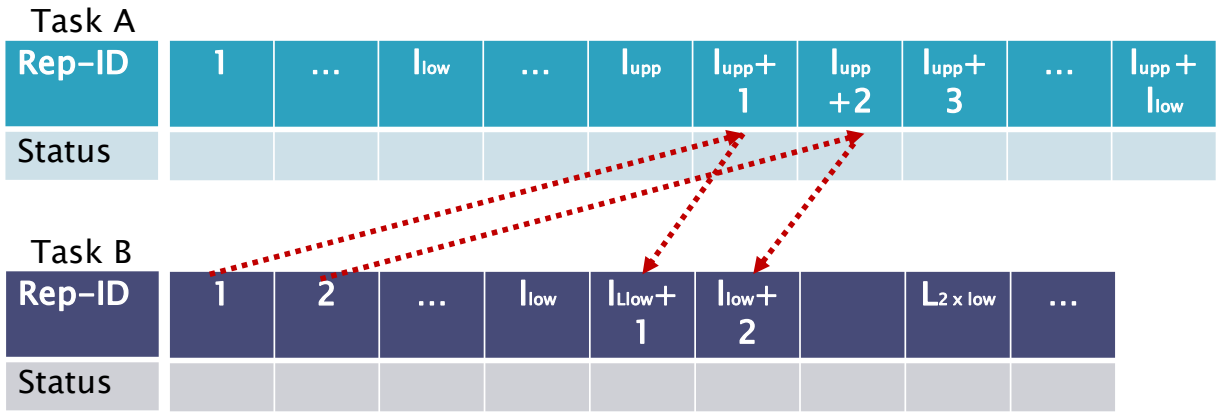


Figure 18 - SAS for duplicable tasks. The SAS edges are in both directions.

$$\begin{cases} S(A_i) \leftarrow \{S(A_{i-1})\} & i < l_{upp} \\ S(A_i) \leftarrow \{S(A_{i-1}), S(B_{i-l_{upp}})\} & i \geq l_{upp} \end{cases} \quad (4.1)$$

$$\begin{cases} S(B_j) \leftarrow \{S(B_{j-1})\} & j < l_{low} \\ S(B_j) \leftarrow \{S(B_{j-1}), S(A_{j+low})\} & j \geq l_{low} \end{cases} \quad (4.2)$$

What remains to do is to find an efficient expression that can compute the formal definition as is presented in (4.1) and (4.2) . The following expressions do exactly this. These expressions can be easily and efficiently (power-wise) implemented in hardware, which enables fast scheduling. Replica dispatching for either of the tasks will happen only when the results of the expressions are positive.

$$alloc_A = l_{upp} - (A.s - B.s) \quad (4.3)$$

$$alloc_B = (A.s - B.s) - (l_{low}) \quad (4.4)$$

A cannot advance more than l_{upp} dispatched replicas, this is seen in (4.3), where the difference in the number of dispatched, a.k.a the lead, is received from $(A.s - B.s)$. Following this the difference between the maximal lead and the actual lead is computed.

B is limited to how close it can get "near" A. The distance between them is computed. Only if the lead is greater than $(l_{upp} - l_{low})$, can B dispatch replicas some of its replicas (because B can come closer to A).

Answers to the questions raised earlier in this chapter:

- Correctness – It is not possible to enforce correctness using this directive as the replicas are scheduled only based on the dispatching of the other task and not the completion of the other replicas. Even by using substantial upper boundary correctness cannot be enforced as a certain replica might take an infinitely long time and thus the dependent task might begin dispatching before this replica has completed.
- Fairness:
    - Dispatching – This primitive ensures fair dispatching as there is an upper limit to how many replicas one of the tasks can dispatch before replicas from the other must be dispatched before it can continue.
    - Number of cores – This directive does not guarantee fairness in the number of cores. Consider the case where the replicas of B are considerably longer than those of A, for example 10^4 longer. Also, it will be assumed that there are plenty of replicas (this can be a near infinite number) for each of the tasks and the number of cores is small compared to the number of replicas. At some point in time, A will be unable to dispatch more tasks without Ballocating replicas before it. At this time, every core that becomes free will receive either a replica of Bor receive a replica of A followed by a replica of B. Due to the fact that the number of cores is small compared to the length of the replicas and the number of replicas, all the cores will be executing replicas of B. This is depicted in Figure 19.

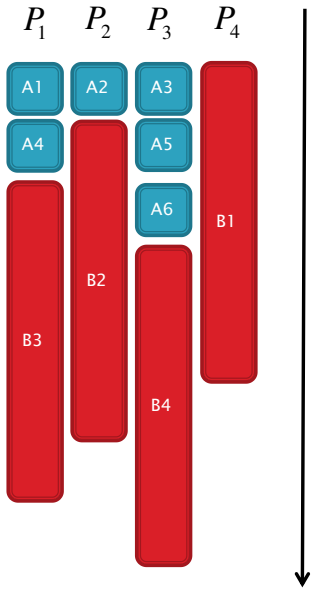


Figure 19 – Example wherein task B takes over all the cores. Upper = 3. Lower=1.

- Cache issues:
  - Total memory required – This answer is parameter dependent and can go either way, as was explained earlier in this chapter.
  - Cache misses – in general, it is possible to reduce the number of cache misses of both replicas in the case that both tasks use the same memory as was explained earlier. However, it is possible to increase the number of memory misses by causing memory thrashing. This scenario can be created for the case that the duplicable tasks access different data arrays that use the same cache lines, respectively. By using a similar access pattern to the data and dispatching replicas alternately a reduction in performance will be visible immediately.
- Scheduling efficiency
  - Fast and efficient scheduling – This directive supports efficient scheduling as it is possible to update the scheduler that numerous replicas are runnable due to the use of two boundaries .It is possible to use the fast scheduler that is currently part of the hardware.
  - Background scheduling – it is possible to do background work and prepare groups of replicas for dispatching in the background even when all the cores are busy. This is because at some point in time (when A reaches the upper boundary) the tasks will alternate in dispatching replicas. Thus, after one of the tasks dispatches a chunk of replicas (even if the chunk is the number of cores in the system), the other task can compute how many tasks it can dispatch and can feed this data to the scheduler

Motivating benefits

- Expressive power – Allows enforcing scheduling order. For example, each $A_i$must begin before each B_i.
- Can reduce cache misses.
- The directive can easily be adapted to Plurality's platform which effectively schedules multiple replicas to numerous idle cores.

This primitive does not ensure correctness, nor can it always reduce the number of cache misses. Nonetheless, in the event that both tasks are using the same data, there are ways to reduce the number of cache misses and the idle cycles when a cache miss occurs. There are two extremes to the number of cache misses and idle cycles:

1) Scheduling all of A's replicas and after that scheduling $B's$ replicas. This will result in both tasks suffering for the same cache misses.
2) Alternating between the duplicable tasks (i.e., alternately dispatching single replicas from the two tasks). In this case, the replicas of the first task, A, will cause the cache misses; however, $B's$ replicas, which arrive shortly after A's, will also wait on the same memory that is being fetched. In essence, the number of cache misses is reduced, but the number of idle cycles increases.

Remark. The above observation, whereby it is actually better to stagger the executions of two tasks that use the same memory so that only the leading one incurs miss penalties is interesting and may have broad applicability.

The main motivation for the SAS primitive is to overcome the first situation. However, by improper selection of the boundaries, the second situation can occur. The following is suggested as a rule of thumb for selecting l_{low} for the case that the memory needed by each replica is considerably small compared to the shared memory and the replicas of both duplicable tasks are an equal length. This rule of thumb solves the second problem:

$$\tilde{l}_{low} = 2 \cdot |Cores|. \tag{4.5}$$

In Figure 20 we present a chart with the execution times for the same program. The solid curve refers to an application with two duplicable tasks that are executed one after the other without the SAS directive. The dashed curve is the same application using the SAS directive. The application that was tested used $n = 4 \cdot 10^6$ replicas that each replica accessed several sequential elements in the array. The size of the array is $4MB$, so the array cannot fit into the shared memory. As this example was presented earlier, a detailed description of events will not be given here. SAS was used here where $l_{upper} = C \cdot n$ for $0 < C < 1$ and $l_{low} = 0.2 \cdot l\_upper$. The abscissa is l_{upp}. The ordinate is the number of cycles required to complete the application. In the range $0 < C \leq 0.5$ there is an obvious improvement of SAS over using duplicable tasks as the number of caches misses is greatly reduced. For $0.7 < C < 1$ there is no improvement as the cache misses are not avoided. In $0.5 < C \leq 0.7$ there is a decrease in the number

of cache misses compared to the regular situation, but there are some cache misses that can't be avoided.

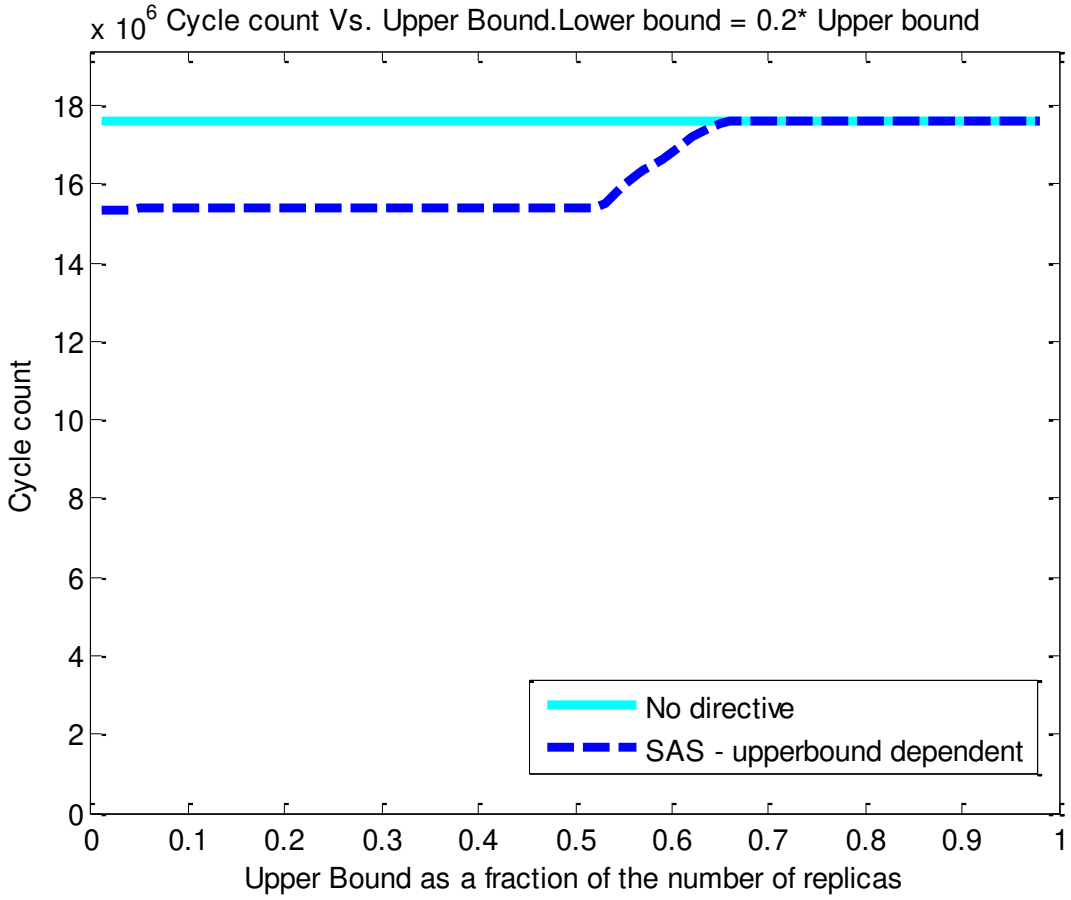


Figure 20 – Example use of SAS. The abscissa is the size of the boundary in percentage of the total number of replicas. The lower boundary is 20% of the upper boundary. The ordinate is number of cycles required to complete execution.

4.6.2 Limit Number of Active Replicas

Limit Number of Active Replicas, LNAR for short, is used to limit the number of concurrent replicas of a duplicable task. This directive is useful for the situation wherein the number of replicas exceeds the number of cores in the system and it is desirable that not all the cores execute replicas because of I/O limitations or memory footprint issues.

This directive refers only to the number of started replicas and not to the order of their completion.

Name	Type	Descriptions
A	Task	The task that is limited in the number of concurrently executed replicas.

K	Integer	Number of replicas that can be executed concurrently.

Table 4 – LNAR parameters

The formal definition of the directive is given here.

$$\begin{cases} S(A_i) \leftarrow \{S(A_{i-1})\} & i \leq K \quad (1) \\ S(A_i) \leftarrow \{S(A_{i-1}), A.s - A.c < K\} & i > K \quad (2) \end{cases} \tag{4.6}$$

In (4.6) the first expression (1) refers to the initial phase where it is possible to dispatch K replicas as long as the dispatching is in order. In (2), the dispatching of a replica is still dependent on in order dispatching, while maintaining that the of active replicas does not exceed K. The number of active replicas is the number of started replicas minus the completed replicas.

Rep-ID	1	2	3	...	K-1	K	K+1	K+2	K+3	K+4
Status	S	S	S		S	S	NS	NS	NS	NS

(a)

Rep-ID	1	2	3	...	K-1	K	K+1	K+2	K+3	K+4
Status	S	S	S		C	C	S	S	NS	NS

(b)

Figure 21 - Limit By Starts directive by K replicas – In (a) the initial phase of the dispatching can be seen. In (b) two replicas completed $k-1$ and k, therefore, two more replicas can be dispatched.

From a hardware point of view, it is simple to compute the number of replicas that can be dispatched:

$$alloc_A = K - (A.s - A.c) \tag{4.7}$$

In expression (4.7) K limits the maximal number of replicas that can be dispatched. The number of active of replicas is $A.s - A.c$, so the difference between K and the active replicas is the number of replicas that can be dispatched.

Answers to the questions raised earlier in this chapter:

- Correctness – No. Similar to the SAS directive, it is not possible to enforce correctness using only information on starts.
- Fairness – It is not possible to discuss fairness for this directive as there is a single task and its scheduling is to itself.
    - Dispatching – Task is limited by the number of dispatched replicas which means that it can't take all the resources for itself.

    - Number of cores – Task is limited in the number of cores it receives.
- Cache issues:
    - Total memory required – Will reduce the amount of memory required in comparison with the situation that the number of concurrent replicas is unlimited.
    - Cache misses – in all likelihood, the number of cache misses will not change or be reduced. This depends on the memory access pattern. For example, if a cache line is used by different replicas that require different data, performance might be reduced to cache thrashing. In all likelihood, this thrashing would also occur when a duplicable task is used and there is no limitation on the number of active replicas. Because of this, a straight answer cannot be given.
- Scheduling efficiency –
    - Fast and efficient scheduling – computing the number of replicas to dispatch in hardware is simple.
    - Background scheduling – It is not possible to prepare workloads in the background as replicas cannot be dispatched until others have completed.

Motivating benefits

- When there are a great many runnable replicas and there is an especially long regular task that needs a single core.
- When the resources (especially memory) required by a large number of replicas exceeds the shared memory supplied by the system. Rather than suffer from memory thrashing, it is preferable to dispatch fewer tasks.

4.6.3 Assign Cores Fairly

Assign Cores Fairly, ACF for short, is used to split the cores evenly between two duplicable tasks. This directive is useful for the situation wherein the number of replicas exceeds the number of cores in the system and it is desirable that not all the cores execute same-task replicas. This directive refers only to the number of started replicas and not to the order of their completion.

Name	Type	Descriptions
A	Task	The first task.
B	Task	The second task.

Table 5 – ASF parameters

The formal definition of the directive is given here.

$$S(A_i) \leftarrow \{S(A_{i-1}), (A.s - A.c) \leq (B.s - B.c)\} \quad (4.8)$$

$$S(B_i) \leftarrow \{S(B_{i-1}), (A.s - A.c) > (B.s - B.c)\} \quad (4.9)$$

In (4.8), A_i can be dispatched in order and as long as the number of active B's is greater than the number of active $A's$. In (4.9), exactly the same requirements are placed on B.

Unlike other directives that were presented in this paper, this directive requires one piece of additional information, which is the number of idle cores in the system, this number will be specified as $emptyC$.

To compute the number of replicas that can be dispatched:

$$alloc = \left(A.s - A.c\right) - \left(B.s - B.c\right). \tag{4.10}$$

In expression (4.10) $t$he numbers of active of replicas of both tasks are compared. If $alloc < 0$ then there are more active replicas of B in the system and A may dispatch accordingly the difference. If $alloc > 0$ then there are more active replicas of A in the system and B may dispatch accordingly the difference. If $alloc = 0$ then there is an equal number of active replicas and the cores should be divided equally between the duplicable tasks $emptyC/2$. Dividing by two is a simple and fast operation in hardware that can be implemented efficiently using a shift operation.

Answers to the questions raised earlier in this chapter:

- Correctness – No, the duplicable tasks are executed concurrently without any constraints between them. It is not possible to enforce correctness using only information on starts
- Fairness –
    - Dispatching – there is no fairness in the number of dispatched replicas. This directive is based on the number of active replicas. For example, consider the case where $|A_i| = 1000 \cdot |B_i|$, both tasks will receive an equal number of cores. However, B will dispatch a 1000 replicas for each of $A's$.
    - Number of cores – when A, B are the only tasks that are being executed, both tasks use receive an equal number of cores. When A,B are not the only tasks that are being executed, the cores might not be divided equally. Consider the graph in Figure 22, where A, B, C are duplicable tasks and S, T are regular task. A, B are using the ACF directive. When C has a higher priority than A, B and cores become idle, C's replicas will be dispatched and A, B will no longer have an equal number of cores.

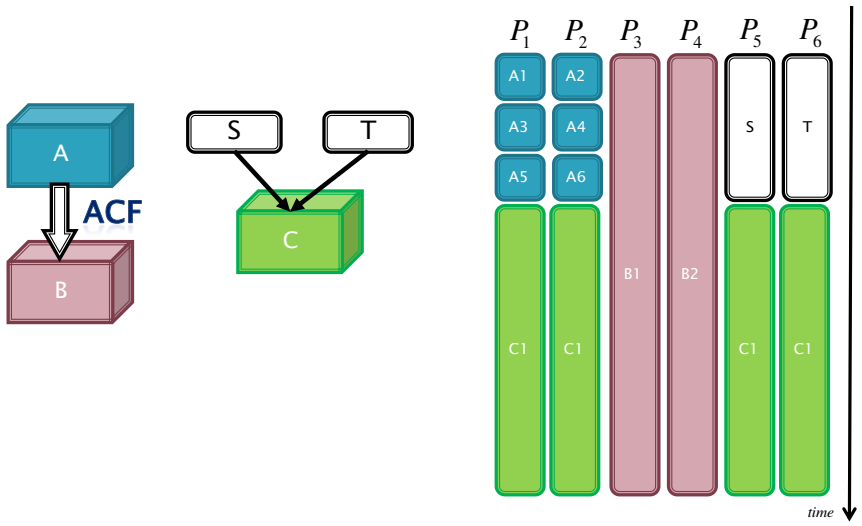


Figure 22 - asdas

- Cache issues:
  - Total memory required – Will reduce the amount of memory required in comparison with the situation that the number of concurrent replicas is unlimited.
  - Cache misses – in all likelihood, the number of cache misses will remain the same or be reduced, depending on the memory access pattern. If cache-line thrashing occurs (multiple duplicable tasks access same line) when using this directive, it is likely that this thrashing would occur when this directive is not used.[9] So there is no straight answer.
- Scheduling efficiency –
  - Fast and efficient scheduling, as computing the number of replicas to dispatch in hardware is simple.
  - Background scheduling – It is possible to prepare workloads in the background.

Motivating benefits

- When tasks are of different length and it is desired that the cores be divided equally amongst the tasks.
- When one task is memory intensive and the other is computational intensive. By executing them concurrently rather than one after the other, the memory intensive task will benefit from the reduction in the number of memory requests compared to the case that it uses the cores of the computational intensive task.

[9] When the directive is not used, it is possible to execute an equal or greater number of replicas concurrently.

- When the replicas of both tasks are equal length and they both use the same data. This is similar to SAS.

4.7 Hardware-extending directives for duplicable tasks

The reason that this sub-section is called "hardware-extending" is that the directives that will be presented here cannot be implemented using the current hardware/implementation of Plurality. We will present both the reasoning why the current hardware is not enough and the motivation for new hardware. Also, we will show that by adding simple hardware these directives can be implemented.

4.7.1 Supporting correctness for duplicable tasks

Consider the following example. A, B are duplicable tasks. B_1 is dependent on the completion of A_1, A_2, A_3. This is depicted in Figure 23. Using the fields of a duplicable task, the following questions cannot be answered:

- What is the id of most recently completed replica?
- Assuming that the id of the last replica completed is given and that id is r, how many replicas with smaller ids than r have completed?
- Assuming that the id of the last replica completed is given and that id is r, have all the replicas prior to r (in the order of the dispatch) completed?
    - How much is memory is needed to be able to answer this?
    - How fast can this be checked?

Using only data on the number of dispatched replicas and the last replica to complete is not enough, as depicted in the following Gantt charts, Figure 23 (b),(c). Replica A_3(light blue) completed at time $\hat{t}$. At time $\hat{t}$, 3 of $A's$ replicas have been dispatched. If B_1 is dispatched as it is in Figure 23 (b), then the precedence constraints are not respected. The proper schedule is seen in (b).

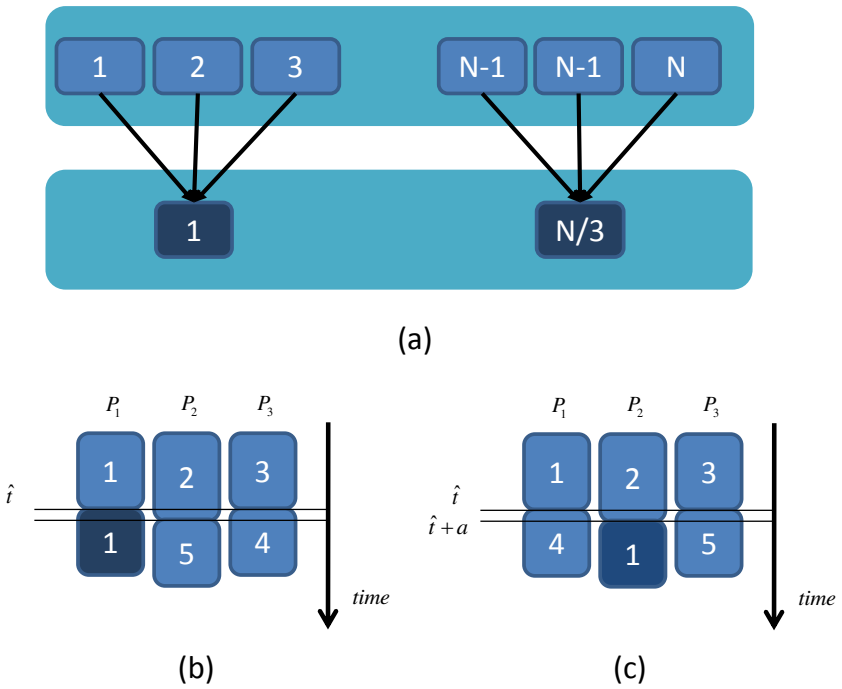


Figure 23 – (a) Given two duplicable tasks with precedence constraints as displayed. Possible schedules based on knowledge of the total number of dispatched replicas and the total number of completed replicas. (a) A schedule that does not adhere to the constraints of the tasks. (b) A schedule that does adhere to the constraints.

As can be seen from the example, because the completion signals of the replicas are not maintained, it is not possible to know when certain replicas become "really" runnable and not "assumed" runnable as was assumed in Figure 23. In order to know when a replica becomes runnable, a completion signal or something similar for each replica needs to be maintained. In the following we will suggest a new field and explain how this field supports our requirements. Furthermore, we will show that the new field can be implemented efficiently in hardware.

The earliest active, ea for short, replica will help the scheduler receive data on the overall status of a duplicable task. By knowing the id of the ea replica, it is possible to state that all replicas with smaller ids have completed, otherwise, one of these replicas would be the earliest active due to the in order replica dispatching.

We define the additional fields for duplicable tasks in the following table.

Variable	*Name*	*Type*	*Description*	*Is currently supported*
Earliest Active	ea	Int	This variable states the index of the earliest replica that has started but has yet to complete.	No

Table 6 - Additional fields for duplicable task

Computing the Earliest Active field

Efficiency is key factor for runtime scheduling of tasks. If scheduling directives are to be created that are dependent on this field, it must therefore be efficient. Furthermore, it is possible to schedule replicas of a duplicable task in a small number of cycles: $O(\log_2(|cores|))$. To consider the computation of this field efficient, it should be possible to compute it in approximately the same number of cycles.

There are several approaches to solving this problem. One is to maintain a bitmap, one bit per replica, in order to track status of the replicas. 0 and 1 represent uncompleted and completed replicas, respectively. The first zero in the array represents the first uncompleted replica, so this is the earliest active replica. Once the ea is completed, its status changes to complete. After the status change, a sequential search over the bitmap must be resumed to find the new earliest active replica (this is mandatory if completion is not in order, but latency can be hidden by keeping track of the id of the next 0).

This approach has several drawbacks: the bitmap size can be extremely large, regardless of the number of cores; the search over the bitmap is considerably slow as it is linear. This approach also requires new hardware that collects completion signals for the completed replicas and writes the completion results to the bitmap. It should be noted that the hardware needed is not simple or straightforward, as the completion signals can arrive from multiple cores in a single cycle, so the completion signals must be sorted for the sake of speed optimization.

One way to overcome the size dependency problem is to use a predefined bitmap size. However, this raises new problems and still requires similar hardware to collect the completion signals. The new bitmap would be a cyclic bitmap, with the position of the earliest active advancing (in a cyclic fashion) whenever the ea changes. As the size of the bitmap is defined ahead of time and is limited, it is now required to limit the number of dispatched replicas. This approach suffers utilization problems for data dependent programs.

We next present a logarithmic cycle implementation for the computation of this field. We note that *ea* is analogous to the earliest instruction that still cannot be committed in a re-order buffer.

4.7.2 Logarithmic Re-Order Buffer for Duplicable Tasks

In our solution, each core – has an additional output port. Using this port, the core transmits the task id of the duplicable task that it is executing and the replica id. As the ea is the replica with the smallest id, the outputs of the different cores are compared and, whenever the task ids are the same, the minimum of these is transmitted onwards. The cores constantly transmit the id of the task and replica they are executig.In Figure 24 ,the “Min” units receive 3 inputs and have one output. The inputs of the “Min” unites are the ids that are transmitted by the cores and the duplicable task id that is of interest. The task

id is compared with the task id received from the cores. If the task ids do not match (different duplicable tasks), the input is filtered and the id is not compared to the other input. The output of the "Min" unit is the replica with the smallest id. Obviously, the "Min" unit can have more than just two id inputs, but this is an implementation detail. The re-order buffer can compute the ea of only one duplicable task at a time. By pipelining the different stages of the re-order buffer, it is feasible to change the id of the duplicable task and use the same re-order to compute the ea of two duplicable tasks with twice the latency.

The output of the unit is then propagated to the following unit until an overall minimum is reached in the final stage. A total of $O(\log(cores))$ stages are needed. The original fields of the duplicable tasks are updated quickly (as they are implemented in hardware) which is important for efficient scheduling. For this field to be both usable and practical, it must be computed efficiently. Therefore, several requirements must be met:

- The ROB has to be fast
- Low power
- Small physical size

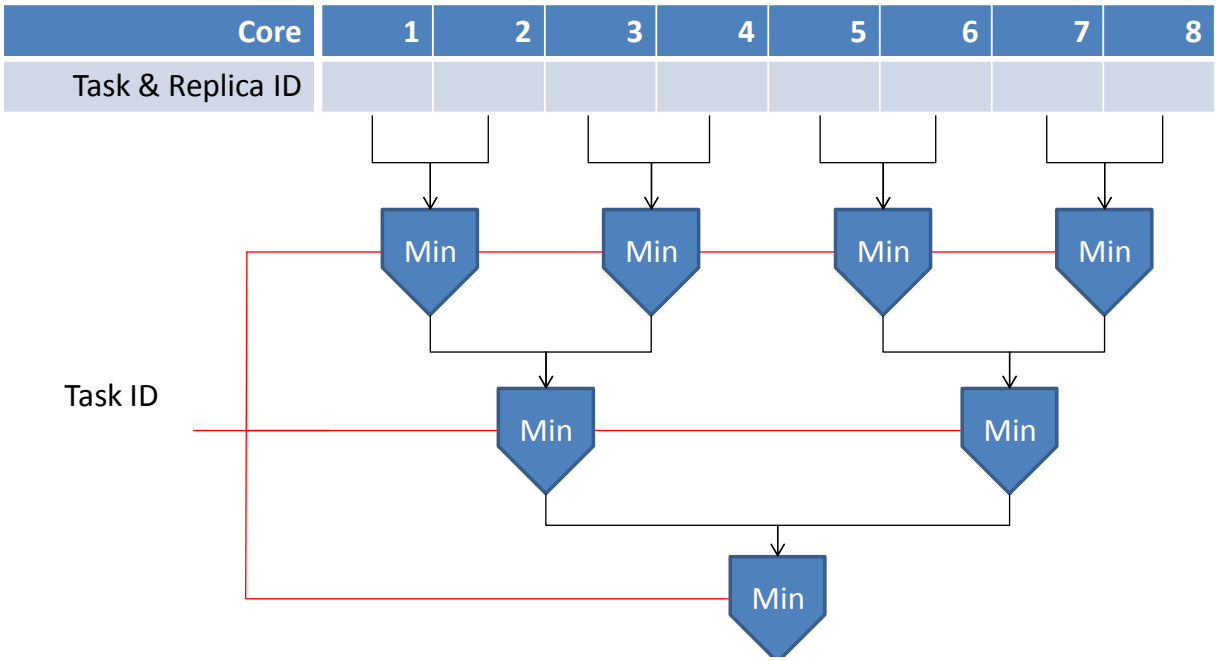


Figure 24 - Earliest Active – A logarithmic Re-Order Buffer

We were able to implement the ROB with the following specifications based on Synopsis's Design Vision:[10]

Variable	*Name*

[10] There are several ways to implement the "Min" unit in hardware.

Number of cores in the system	64
Process used	65 nm
Number of cycles(assuming $400MHz$ clock)	3
Total dynamic power	7.0065 mW
Physical size	0.024687 mm^2

Table 7 – Rob specifications

This hardware has not been optimized. The main goal of synthesizing the hardware was proof of concept. For actual systems the parameters can vary. Now that this hardware has been explained and an efficient implementation been shown, we next present several directives that utilize this field.

4.7.3 Start After Complete for duplicable tasks

Start After Complete, SAC for short, requires that certain replicas complete, i.e., precedence constraints, before specific replicas can be dispatched. It requires the ea field. We present two types of SAC:

- Type 1 – one task's replicas are dependent on the starting of replicas in the other task, while the other task's replicas are dependent on the completion of replicas from the first task.
- Type 2 – both tasks require the completion of certain replicas in the other task before dispatching other replicas.

SAC Type 1 – A Doesn't wait

The SAC type 1 directive is used for the situation that there is a precedence constraint between A and B, such that any given A replica precedes certain B replicas. The relationship is restricted to "leads by k" in order to permit efficient and speedy implementation. A is in no way constrained by B, which is the reason that only one boundary is needed.

Name	Type	Descriptions
A	Task	The first task.
B	Task	The second task.
l_{low}	Integer	States the distance in replicas of the last replica that B is dependent on in A.

Table 8 - SAC Type 1 parameters

The lower boundary refers to the last replica of A that a specific B is dependent on. B might be dependent on several replicas of A. The last one is the one that interests us. The reason that only the last replica interests us, is that we are using the ea field. By using this field, we are ensured that if the last replica that we are interested equals ea than all replicas prior to this have completed. This is suboptimal but makes implementation practical.

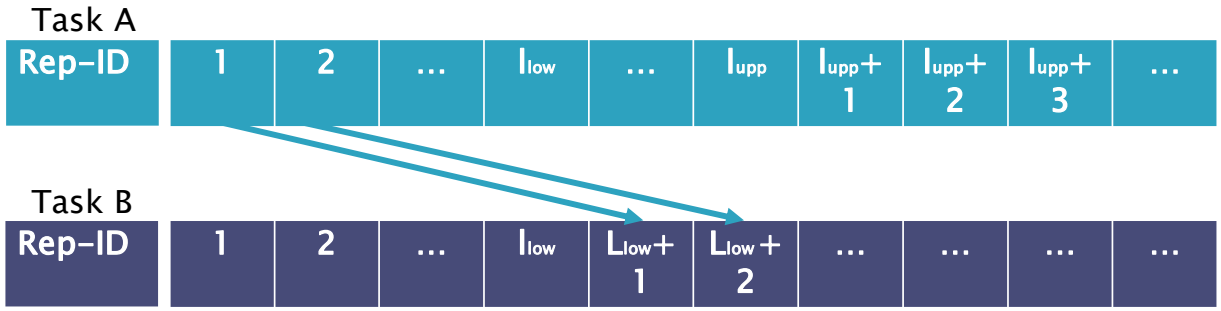


Figure 25 – SAC Type 1 for duplicable tasks. Edges from A to B are precedence.

The formal definition of the directive is given here.

$$S(A_i) \leftarrow \{S(A_{i-1})\} \tag{4.11}$$

$$\begin{cases} S(B_j) \leftarrow \{S(B_{j-1})\} & j < l_{low} \\ S(B_j) \leftarrow \{S(B_{j-1}), C(A_{j-l_{low}})\} & j \geq l_{low} \end{cases} \tag{4.12}$$

Replicas from A can be dispatched whenever there is an idle core. However, when there are replicas of B that can be dispatched, these replicas should be selected. This can be done by giving task B a higher priority than A. If B does not receive a higher priority, then A can continue dispatching replicas until it is done. $B's$ replicas are runnable only after replicas in A that are l_{low} replicas distance away have completed. Computing these rules in hardware is easy.

Hardware computation:

$$alloc_B = A.ea - B.s - l_{low}. \tag{4.13}$$

The first part of the expression $A.ea - B.s$ refers to the distance between the earliest active in A and the last replica to start in$B$. This distance has to be at the very least l_{low} for there to be replicas of B that can be dispatched.

Answers to the questions raised earlier in this chapter:

- Correctness – It is possible to enforce correctness using this directive as it supports precedence constraints.
- Fairness:
    - Dispatching – This primitive does not support fairness as A can dispatch as many replicas as it likes.

- Number of cores – similar to the example in SAS, it is possible that all of the cores be used by one duplicable task.

- Cache issues:
  - Total memory required – by selecting a proper lower bound, it is possible to reduce the amount of memory required in the system.
  - Cache misses – in general it is possible to reduce the number of cache misses of both replicas in the case that both tasks use the same memory, by properly selecting the bounds of the directive. Similarly to the total memory requirement, selection of the wrong bounds might increase the cache misses.
- Scheduling efficiency –
  - Fast and efficient scheduling – As this directive is dependent on the ea field, the answer to this is dependent on the hardware implementation of this field. Other than this requirement, computation of this field can easily be implemented in hardware.
  - Background scheduling – it is possible to do background scheduling. Whenever a core becomes idle, it is possible to dispatch a replica of A. While $A's$ replicas are being executed it is possible to compute $alloc_B$. Next time a core becomes idle and $alloc_B > 0$, dispatch replicas from B.

Motivating benefits

- In the case that the replicas of both duplicable tasks access the same memory:
  - It is possible to ensure that only one duplicable tasks accesses the memory at any given time.
  - It is possible to reduce the number of cache misses.
    - A specific use might be, given an array X, compute $f(X)$ in task A and $g(X)$ in task B. Placing a precedence constraint between the replicas can ensure that B doesn't wait on the same cache miss that A does.
- B is dependent on the computation of A.

SAC Type 2 - A Does Wait

The SAC type 2 directive is used for the situation that there is a precedence constraint between replicas of both tasks. Similar to SAS, one of the tasks leads the other task in the number of dispatched replicas. Unlike with SAS, in SAC it is not possible to dispatch more replicas of the leading task than the upper boundary as there is a dependence on the completion of replicas from the other task.

Name	Type	Descriptions
A	Task	The first task.
B	Task	The second task.
l_{upp}	Integer	Upper boundary on how many more replicas A can

		start before a specific B finishes.
l_{low}	Integer	Minimal distance between a specific completion of A and a start of B.

Table 9 - SAC type 2 parameters

The formal definition of the directive is given here.

$$\begin{cases} S(A_i) \leftarrow \{S(A_{i-1})\} & i < l_{upp} \\ S(A_i) \leftarrow \left\{S(A_{i-1}), C\left(B_{i-(l_{upp}-l_{low})}\right)\right\} & i \geq l_{upp} \end{cases} \tag{4.14}$$

$$\begin{cases} S(B_j) \leftarrow \{S(B_{j-1})\} & j < l_{low} \\ S(B_j) \leftarrow \{S(B_{j-1}), C(A_{j-l_{low}})\} & j \geq l_{low} \end{cases} \tag{4.15}$$

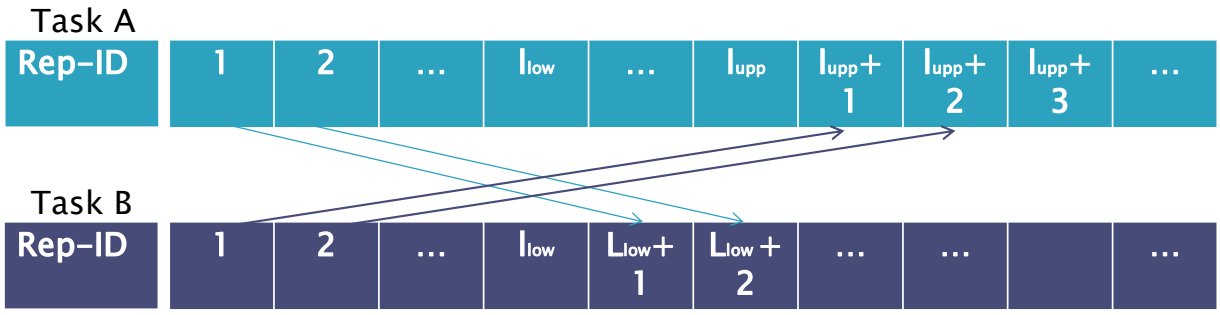


Figure 26 - SAS Type 2 - Example

For this directive, the explanation of the formal definition will begin with B. Similar to SAC Type 1, B's replicas are dependent on A's replica l_{low} positions back. There is always a minimal distance between A and B. The A's can continue dispatching up to the point that there is a difference of l_{upp}. As there is a minimal distance at all times, the maximal number of replicas that A dispatches before it must to wait on the completion of a replica of B is $(l_{upp} - l_{low})$. Computation of these limitations in hardware is givn here.

$$alloc_A = l_{upp} - (A.s - B.ea) \tag{4.16}$$

$$alloc_B = A.ea - (B.s - low) \tag{4.17}$$

In expression(4.16), $(A.s - B.ea)$, refers to the difference between the number of started A's and the longest completed prefix of replicas $B's$. Due to the minimal distance between the tasks and $A's$ ability to dispatch more tasks, this number is limited to l_{upp}.

For expression (4.17), $(A.ea - B.s)$ refers to the distance between the earliest active in A and the longest completed prefix of replicas B. This distance has to be at the very least l_{low} for there to be replicas of B that can be dispatched.

Answers to the questions raised earlier in this chapter:

- Correctness – It is possible to enforce correctness using this directive as it supports precedence constraints.
- Fairness:
    - Dispatching – This primitive supports fair dispatching as there is an upper limit to how many replicas one of the tasks can dispatch before replicas from the other must be dispatched before it can continue.
    - Number of cores – similar to the example in SAS, it is possible that one task gain control over all the cores in the system.
- Cache issues:
    - Total memory required – due to the ability to select the bounds of the directive based on memory access patterns of the tasks it will be possible to limit the total amount of memory required. Obviously, selecting the wrong bounds might inverse this situation.
    - Cache misses – in general it is possible to reduce the number of cache misses of both replicas in the case that both tasks use the same memory, by properly selecting the bounds of the directive. Similarly to the total memory requirement, selection of the wrong bounds might increase the cache misses.
- Scheduling efficiency –
    - Fast and efficient scheduling – As this directive is dependent on the ea field, the answer to this is dependent on the hardware implementation of this field. Other than this requirement, computation of this field can easily be implemented in hardware.
    - Background scheduling – the directive is dependent on the ea field, which means until this field is updated, availabilities cannot be updated. Background scheduling is not possible.

Motivating benefits

- Allows limiting the total number of concurrent replicas for each of the tasks.
- Similar to SAC Type 1 with the difference that A cannot dispatch endlessly, thus, it is possible to limit the total amount of memory required and reduce cache misses.
- In the case that the replicas of both tasks use the same memory
    - It is possible to enforce that only one of them can access the memory at a given time.
    - It is possible to reduce the number of cache misses:

- A specific use might be, given an array X, compute $f(X)$ in task A and $g(X)$ in task B. By limiting the number of replicas that A leads B by, it is possible to reduce the number of cache misses by 50%.
- Expressive power – allows enforcing scheduling order.

While this directive enforces correctness, it may cause performance reduction compared to SAC type 1 (this is parameter dependent) due to the limited number of replicas that can be executed concurrently. For example, a long replica of B might cause A to wait some time before the replicas are runnable. It is worth noting that whenever both duplicable tasks use ea, it may be preferable to have two units that compute the ea to improve performance.

4.7.4 Limit Number of Replicas after Earliest Active

Limit Number of Replicas after Earliest Active, LNR for short, is used to limit the span (range of ids) of active replicas of a given duplicable. This can be seen as a limited sized sliding window of dispatched replicas. Until the first replica in the window, ea, is not completed the window cannot be moved forward. This directive is similar to LNAR, so are most of the answers to the questions and the motivation. The difference between LNR and LNAR is that we are interested in limiting the number of dispatched replicas w.r.t. to the ea replica dispatched. Furthermore, this directive enforces correctness unlike LNAR.

Another way to treat this directive is to consider this as an internal precedence constraint within the duplicable task, as a certain replica cannot be dispatched until the replica whose id is lower by the window size has completed.

Name	Type	Descriptions
A	Task	The task that is limited in the number of concurrently executed replicas.
K	Integer	Maximal number of replicas that can be executed concurrently. The dispatching of A_{i} is dependent on the completion of A_{i-k}.

Table 10 – LNR parameters

The directive is formally defined as:

$$\begin{cases} S(A_i) \leftarrow \{S(A_{i-1})\} & i \le K \quad (1) \\ S(A_i) \leftarrow \{S(A_{i-1}), C(A_{i-K})\} & i > K \quad (2) \end{cases} \tag{4.18}$$

In (4.18) the first expression (1) refers to the initial phase where it is possible to dispatch K replicas. In (2), the replica dispatching is still dependent on linear order dispatching and completion of a ea.

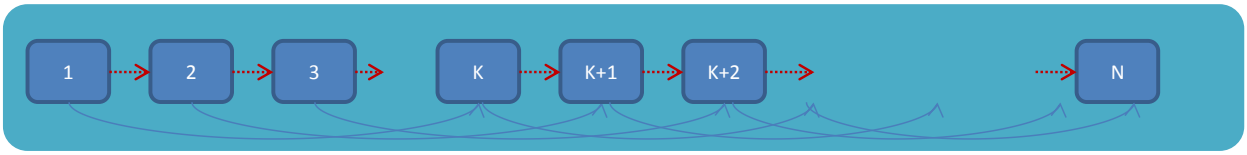


Figure 27 – Graph of LNR

From a hardware point of view, it is simple to compute the number of replicas that can be dispatched.

$$alloc_A = K - (A.s - A.ea) \quad (4.19)$$

In (4.19), K limits the number of replicas that can be dispatched. The number of replicas that can be dispatched is dependent on the number of tasks that are currently in the system which is the difference in the number of started replicas and the ea.

Rep-ID	1	2	3	...	K-1	K	K+1	K+2	K+3	K+4
Status	S	S	S		S	S	NS	NS	NS	NS

(a)

Rep-ID	1	2	3	...	K-1	K	K+1	K+2	K+3	K+4
Status	S	S	S		C	C	NS	NS	NS	NS

(b)

Rep-ID	1	2	3	...	K-1	K	K+1	K+2	K+3	K+4
Status	C	S	S		C	C	S	NS	NS	NS

(c)

Figure 28 – Limit By Complete directive by K replicas – In (a) the initial phase of the dispatching can be seen. In (b) two replicas complete $k-1$ and k, however, no replicas can be dispatched. In (c), replica 1 completes, therefore, replica $K+1$ can be dispatched.

Answers to the questions raised earlier in this chapter:

- Correctness – Correctness can be enforced as precedence constraints are used.
- Fairness – It is not possible to discuss fairness for this directive as there is a single task and its scheduling is to itself.
    - Dispatching – Similar to .
    - Number of cores – In comparison with LNAR, performance here might be hindered as replicas cannot be dispatched until the completion of a the ea. Long replicas can be extreme bottlenecks. However, when enforcing correctness, utilization comes secondary.

- Cache issues:
    - Total memory required – Will reduce the amount of memory required in comparison with the situation that the number of concurrent replicas is unlimited.
    - Cache misses – in all likelihood, the number of cache misses will not change or be reduced. This depends on the memory access pattern. For example, if a cache line is used by different replicas that require different data, performance might be reduced to cache thrashing. In all likelihood, this thrashing would also occur when a duplicable task is used and there is no limitation on the number of active replicas. Because of this, a straight answer cannot be given. By using prerequisites, the task graph designer can select K in a fashion that reduces the number of replicas that can access a specific cache line and remove the thrashing. Similarly to LNAR, there is no straight answer due to parameter dependency.
- Scheduling efficiency –
    - Fast and efficient scheduling – computing the number of allocable tasks in hardware is simple which allows for fast and efficient scheduling.
    - Background scheduling – It is not possible to prepare workloads in the background as replicas cannot be dispatched ea has been completed.

Motivating benefits

- For cases that there is an internal dependence between the tasks.
    - For example, it is possible to calculate the Fibonacci series by setting K=1. For the Fibonacci series, parallelism may not be achieved. However, expressive power is.
    - Data dependencies of previous replicas, for example a replica is dependent on the memory that is fetched by a different replica. By placing the precedence constraint, only one core waits on the memory miss rather than two (or more) cores.
- When there is a known memory access pattern that by not using the directive will suffer a large number of concurrent cache misses which will result in thrashing.

4.7.5 Start After Merged Completion

Start After Merged Completion – Merge, SAMC for short, is directive that is used to state that the prerequisites between the duplicable tasks is such that each B_i is dependent on the completion of M consecutive replicas. The $B_j's$ are dependent on disjoint replicas of A. Given two duplicable tasks, A and B, the dependency between the replicas can be defined as $B_j \leftarrow A_{M\cdot j}, A_{M\cdot j+1}, \dots, A_{M\cdot(j+1)-1}$.

Name	Type	Descriptions
A	Task	The first task.
B	Task	The second task.
M	Integer	The merging factor. The number of A replicas that $B's$ replicas are dependent on.

Table 11 – SAMC parameters

Following the same scheduling technique from the other directives, the replicas will be dispatched in linear fashion. This can be seen as the starting precedence in the formulations (4.20) and (4.21).

$$S(A_i) \leftarrow \{S(A_{i-1})\} \tag{4.20}$$

$$S(B_j) \leftarrow \{S(B_{j-1}), C(A_{M \cdot j}), C(A_{M \cdot j+1}).., C(A_{M \cdot (j+1)-1})\} \tag{4.21}$$

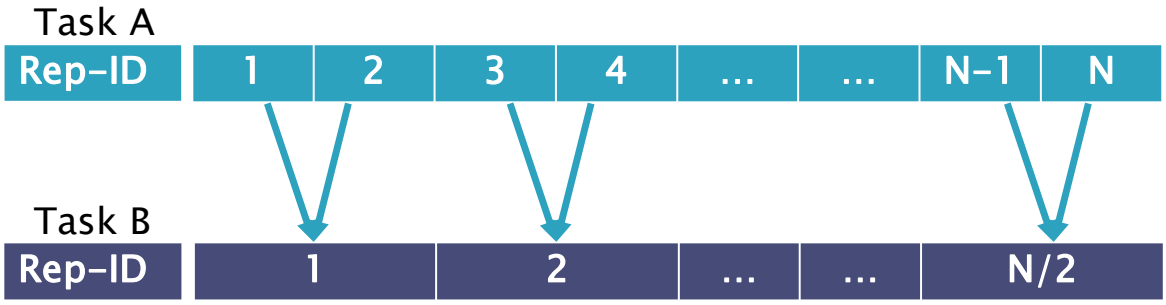


Figure 29 - SAMC example

As depicted, there are no requirements on A. $A's$ replicas be dispatched just like those of a duplicable task. However, $B's$ replicas are dependent on M independent and consecutive replicas.

The number of $B's$ that can be dispatched is the following:

$$alloc_b = \frac{A.ea}{M} - B.S \ . \tag{4.22}$$

The greatest disadvantage of this expression is that it uses a divide operation, which is considerably slower than add and multiply operations.

Before answering the regular questions, note that this directive offers an interesting perspective on the use of priority. It was shown earlier that it is possible to dispatch replicas of B concurrently to those of A based on priority, it must be decided which of the two tasks should be dispatched first in case replicas are runnable from both tasks. Given p_A and p_B, priorities of A and B respectively:

1) $p_A > p_B$: The replicas of A will be executed before any of B are disptached.
2) $p_A = p_B$: The replicas of both tasks can be executed. For the sake of simplicity and practicality, the replicas of A will be executed before the replicas of B. In effect, this scenario is similar to the previous one.
3) $p_A < p_B$: The replicas of B have higher a priority and will be dispatched before those of A as long as precedence constraints are maintained.

Scheduling for each of the priority scenarios will result in the creation of a different schedule as is depicted in Figure 30 (assuming equal length replicas). Based on the in order dispatching assumption, it is seen from Figure 30 (b) and Figure 30 (c) that the replica dispatching order of both tasks is similar. However, the execution time changes. In (b), duplicable tasks are used without directives while in (c) the SAMC directive is used based on $p_A \geq p_B$ .It should be noted that there is an improvement in the schedule length, due to the reduced constraints. In Figure 30 (d), scheduling is done assuming $p_A < p_B$. Notice that replicas from both tasks are executed concurrently, while the dispatching of same-task replicas is in order.

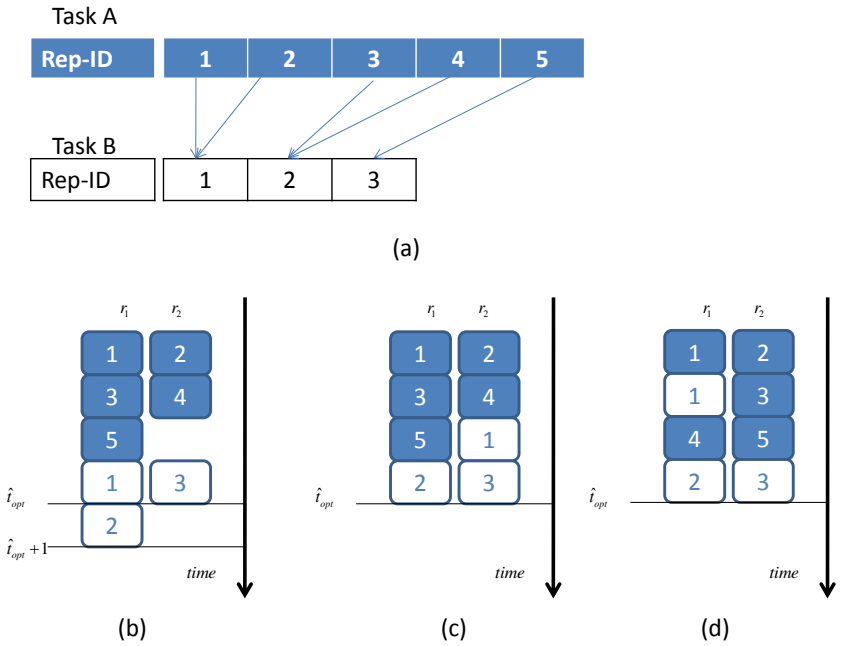


Figure 30 - SAMC example $|A| = 5, |B| = 3, M = 2$. Assuming equal length tasks. (a) $Merge - K$ directive. (b) Scheduling of graph using duplicable task without scheduling directive. (c) Schedule of directive for $p_{d_1} \geq p_{d_2}$. (d) Schedule of directive for $p_{d_1} < p_{d_2}$.

In the following part, it will proved that using SAMC over the situation where given two duplicable tasks A and B such that $B \leftarrow A$, will give better results assuming that the scheduling for both these scenarios is done in-order for each of the tasks.

Lemma 1:

The order of dispatched replicas is the same when using only duplicable tasks and when using the SAMC directive for the case that $p_A \geq p_B$(cases 1 and 2).

Proof:

When using only duplicable tasks, because of the dependency of $B \leftarrow A$, only after all the replicas of A are completed can replicas of B be dispatched. This in turn, gives us the order of dispatching for the standard model : $t_{A,1}, t_{A,2}, \dots, t_{A,|A|}, t_{B,1}, t_{B,2}, \dots, t_{A,|A|}$.[11]

When dispatching according to the SAMC directive:

- $p_A > p_B$ – due to difference in priority all the replicas of A are dispatched before a single replica of B is dispatched. This ensures that the order of dispatching (based on the in order dispatching assumption) is: $t_{A,1}, t_{A,2}, \dots, t_{A,|A|}, t_{B,1}, t_{B,2}, \dots, t_{A,|A|}$.
- $p_A = p_B$ – For the same reason, the same order is received.

Therefore, the order of dispatching is the same for standard model and for SAMC when $p_A \geq p_B$. ■

Notations:

- SL – Schedule Length, essentially is the execution time of the program.
- ST(t) – Start Time of a replica t. Refers to the time that a replica is dispatched.
- $CT(t)$ – Completion Time of replica t. Refers to the time that a replica finishes.
- std - refers to the standard model of using only duplicable tasks.

Lemma 2:

The execution time for dispatching the replicas of A, B using SAMC is smaller or equal to that using only duplicable tasks for the cases $p_A \geq p_B$: $SL_{SACM} \leq SL_{std}$.

Proof:

Assume by contradiction that there exists a replica $t_2 \in B$ such that $CT_{std}(t_2) < CT_{SAMC}(t_2)$. As the length of t_2 is the same for both schedules, it can be inferred that $ST_{std}(t_2) < ST_{SAMC}(t_2)$. Based on the previous lemma, the order of allocable replicas is the same and it is known that the replicas of B in SAMC have an equal or less number of dependencies than its counterpart in the standard model. If a replica in the standard model can be dispatched, so can its counterpart in the SAMC directive. This is in contradiction with the assumption, which means that $ST_{std}(t) \not< ST_{SACM}(t)$ is not correct for any $t \in B$ of the standard model. This implies $CT_{std}(t) \not< CT_{SACM}(t)$, including for the last task that finishes for both approaches.

This leads to the following conclusion: $ST_{SACM}(t_2) \leq ST_{std}(t_2)$. ■

Answers to the questions raised earlier in this chapter:

- Correctness – This directive can enforce correctness as it is precedence based.
- Fairness:

[11] The order of dispatching does not refer to the time of the dispatching.

    - Dispatching – This directive cannot promise fairness in dispatching. If there are no limitations on A, then all of $A's$ replicas can be executed before any of $B's$.
    - Number of cores – Not relevant as they are not supposed to use an equal number of cores.
- Cache issues:
    - Total memory required – while dispatching of B_i might increase the total amount of memory required, as it supposedly uses the memory of all its dependencies, in actuality, this situation is preferable over the situation where all of $A's$ replicas are dispatched before $B's$ replicas. Considering the case where each A_i uses l bytes of memory, thus, a B_i will use $M \cdot l$ bytes of memory. Given C cores that execute only $A_i's$,the total memory required is $C \cdot l$, executing only $B_i's$ would require a total of $C \cdot M \cdot l$. Concurrent execution of the replicas from both duplicable task can reduce the total amount of memory required in comparison with executing each of the tasks separately as B will require the maximal amount of memory. It is worth noting that there are instances where M and l are considerably large, there might be a situation that the executed replicas of the different tasks will want to access the same cache line, which can cause memory thrashing. Even for this scenario it is likely that if the replicas of only one these task's replicas (specifically $B's$) were executed, there would still be memory thrashing.
    - Cache misses – For the instances where the total memory required by A is larger than the shared memory and M is selected appropriately, this directive can reduce the number of cache misses as data that was brought by A will be reused by B without refetching the data from the external memory.
- Scheduling efficiency –
    - Fast and efficient scheduling – for this directive, fast and efficient scheduling cannot be promised for 2 reasons:
        - Computation of the number of replicas that B can dispatch is uses a divide operation. Dividing operations are considerably slow which makes the divide operation the bottleneck for the scheduling directive.
        - It is dependent on the ea field. This is not the major bottleneck of this computation.
    - Background scheduling – This directive cannot be scheduled in the background.

Motivating benefits

- Can allow implementation of merging (control data-flow) trees. Examples of this include parallel implementation of finding the maximal number in array, parallel summation and many more.
- Expressive power.

What we won't show in this sub-section is that it is possible to place a limitation on A so that A cannot run ahead in its dispatching. This is similar to the idea behind the two types of SAC and the reader is left to fill in the missing.

5 Scheduling decision: dispatch duplicable task or regular task

This paper does not concentrate on scheduling policies, however, during our research we were confronted with the question given a long regular task and a duplicable task, which of these tasks should be dispatched first? This following section tries to answer this question for an even more general case than a duplicable task. Our results show that for some of the scenarios we can clearly state which of tasks should be dispatched first.

Given a long task called $long$ and a set of independent tasks $n_1, n_2, \ldots, n_N$ that are dependent on a fork task, called $forkN$. The combination of $forkN$ and $n_1, n_2, \ldots, n_N$ can be seen as the duplicable task. Unlike in a duplicable task, we will not assume that $w(forkN) = 0$. In this appendix, an attempt is made to show that it is better to schedule $long$ over $forkN$ whenever:

$$w(long) > w(forkN) + \sum_{i=1}^{N} w(n_i). \qquad (5.1)$$

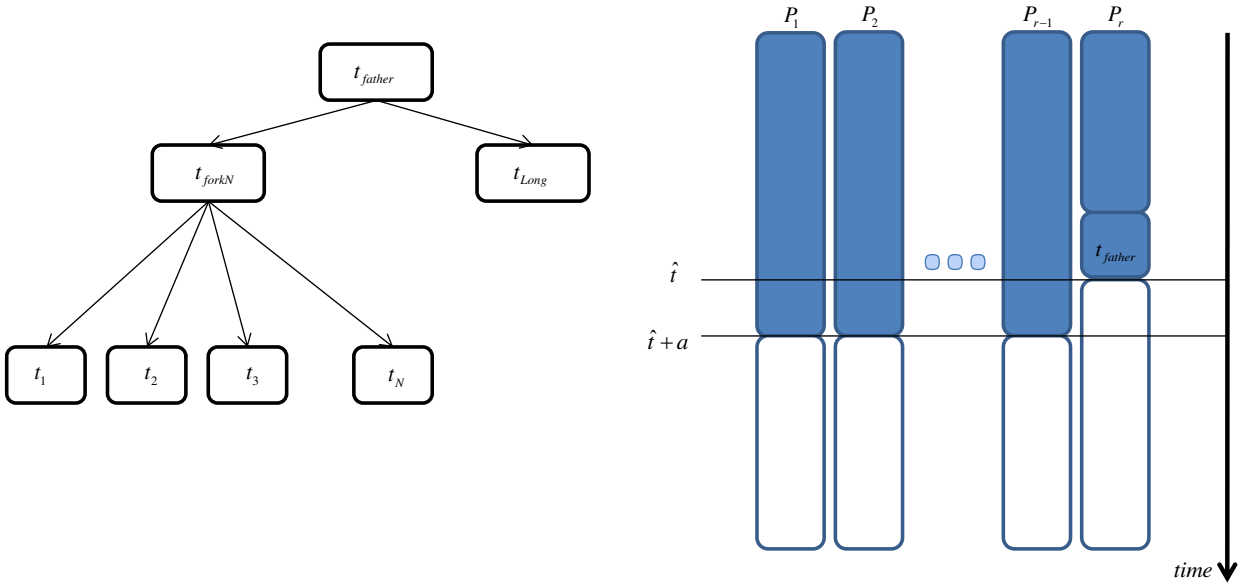


Figure 31 – (a) The task graph; (b) A possible scenario for the cores in the system. Blue cells refer to utilized cores and white cells refer to non utilized cores.

Upon the completion of task $father$ at time $\hat{t}$, both task $long$ and $forkN$ become runnable. If there is more than one idle core both $forkN$ and $long$ can be dispatched. Given only a single idle core to execute these tasks, it must be decided on which of the two tasks will be executed first. This is depicted in Figure 31 (b). In this specific scenario, at time $\hat{t} + a$, the other $r - 1$ cores become idle (having

completed work that is unrelated to the tasks at hand). Which of the 2 runnable tasks should be dispatched? The duplicable task or the regular task?

Before answering this question, a specific, example is given in order to present some intuition on the matter of selecting the proper task. Consider the following tasks lengths, given r cores.

$$\begin{aligned} &N = 3\cdot(r-1), r \geq 2 \\ &w(forkN) = a \\ &w(n_i) = a, i \in \{1,..,N\} \\ &w(long) = 3\cdot a \end{aligned} \tag{5.2}$$

This example will also show that by using critical path, CP for short, based scheduling policies, a sub-optimal schedule might be created. CP based scheduling policies dispatch tasks that are on the critical path, the longest path in the graph[18].

$$w(forkN) + w(n_i) < w(long). \tag{5.3}$$

(5.3) is correct for all $i \in \{1..N\}$. As a result a CP based scheduling policies will dispatch $long$ before $forkN$. This will result in a sub-optimal schedule as can be seen in Figure 32(a). In Figure 32 (b), the optimal scheduling is presented.

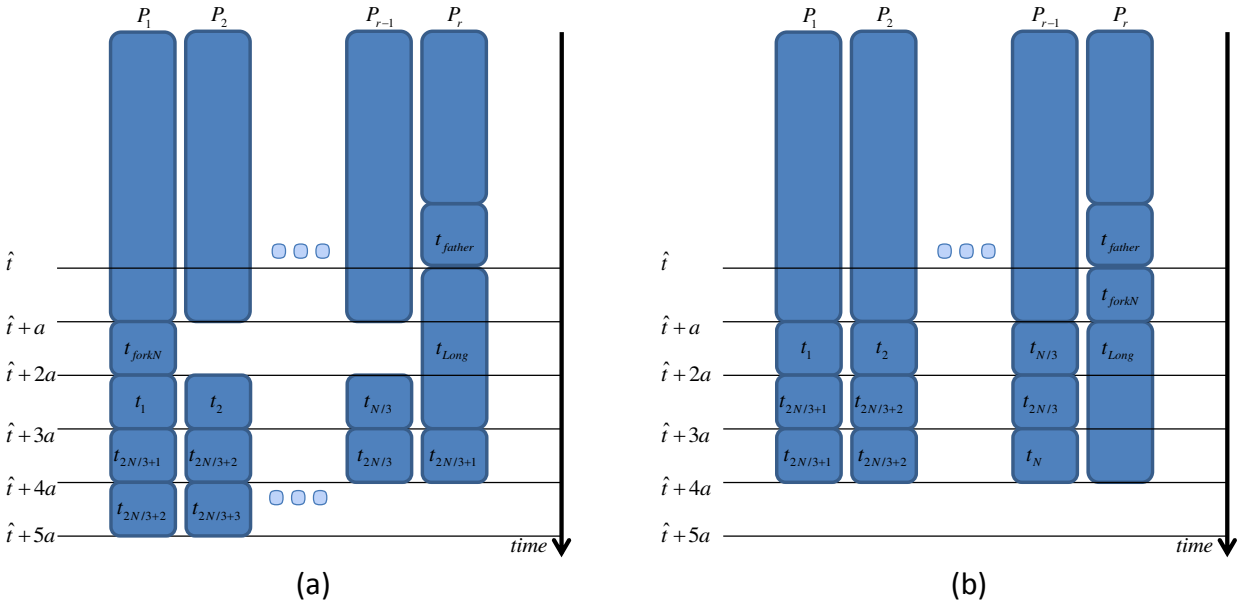


Figure 32 - Possible schedules for previous task graph. (a) Scheduling according to critical path based scheduling algorithms. This scheduling is sub-optimal. (b) Optimal scheduling.

It should be noted that $long$ does not maintain the expression in (5.1):

$$3 \cdot a = w(long) < w(forkN) + \sum_{i=1}^{N} w(n_i) = 3 \cdot r \cdot a - 2 \cdot a \underbrace{=}_{r>2} a(3 \cdot r - 2).$$

The purpose of this example was to show that CP based scheduling can ensure optimal scheduling.

In the following section we will show several scenarios where it is preferable to dispatch $long$ before $forkN$. We also show scenarios where dispatching $long$ before $forkN$ can increase the execution time.

Observation:

It should be noted that there exists $\beta > 0$ such that:

$$w(long) = w(forkN) + \sum_{i=1}^{N} w(n_i) + \beta. \quad (5.4)$$

Requirements:

1) Tasks $long, forkN, n_1, .., n_N$ maintain expression (5.1).
2) At time $\hat{t}$, out of the $r > 1$ cores there is only core that is idle.

SL_{forkN} is used to refer to schedules that $forkN$ is dispatched before $long$.

SL_{long} is used to refer to schedules that $long$ is dispatched before $forkN$.

First, we present the case that a single core executes all the tasks. After this, we will use multiple cores to execute the tasks.

Lemma:

Given (1),(2) and a single core that executes all the tasks, $SL_{long} \leq SL_{forkN}$.

Proof:

A single core is used to execute all the tasks. In this case, all tasks are executed on the same core and therefore, the order of the tasks is irrelevant as the schedule length is the same regardless of the order of execution.

As of now, multiple cores are used to execute the different tasks. This means that at some time before the execution is completed at least two cores are used concurrently, otherwise, this is a single core execution. We define the time that more than one core is used as t' and $t' > \hat{t}$. We also know that:

$$t' < SL_{\text{single}} - \min\left\{ w(long), w(forkN), \min_{i\{1..N\}} \{ w(n_i) \} \right\}. \quad (5.5)$$

Otherwise, only a single core is used.

Observation:
Now, assume by contradiction that there exists a schedule that allows for $forkN$ to be scheduled before $long$ such that SL cannot be reduced. Such a schedule would have a partial order: $forkN, n_1, n_2, \dots, long, \dots, n_{N-1}, n_N$ where $long$ can be placed anywhere between $forkN$ and n_N.

Because $w(long) > w(forkN) + \sum_{i=1}^{N} w(n_i) > w(n_i)$ for all $1 \le i \le N$, it is always preferable to schedule $long$ before all the $n_i's$. Therefore, to get the smallest SL possible a decision on who between $long$ and $forkN$ should be dispatched first needs to be made.

We partition the time range of t' $[\hat{t}, \hat{t} + SL_{single}]$. In one of these sections an additional core will become idle.

i. $\hat{t} < t' \le \hat{t} + \beta$
ii. $\hat{t} + \beta < t'$ and:
    a. $\hat{t} < t' \le \hat{t} + w(forkN)$
    b. $\hat{t} + w(forkN) < t' \le \hat{t} + w(long)$
    c. $\hat{t} + w(long) < t' \le \hat{t} + w(long) + w(forkN)$
    d. $\hat{t} + w(forkN) + w(long) < t'$

Lemma:

Given (1),(2) and that multiple cores executes all the tasks from some time $t' > \hat{t}$. $SL_{long} \le SL_{forkN}$ given a specific time partition.

We will prove the following lemma for time partitions (i), (ii.a), (ii.b), (ii.d).

Proof for (i):

In this case by switching t_{long} and t_{forkN} it is possible to reduce from $SL_{forkN} = w(long) + t'$ to $SL_{long} = w(long) + \hat{t}$. This is in contradiction with the assumption that by scheduling $forkN$ before $long$ SL cannot be reduced. ■

Proof for (ii.a):

In this case, $long$ is executed on the newly idle core during the execution of $forkN$. Completion of $long$ will be $w(long)$ time units later. The first core, the one executing $forkN$, will execute the rest of the tasks $n_1, n2, \dots n_N$ and will complete before $long$ is completed because of (5.1).

$SL_{forkN} = t' + w(long).$

$SL_{long} = max\{\hat{t} + w(long), t' + w(forkN) + \sum_{i=1}^{N} w(n_i)\}.$

The first expression refers to the time needed to schedule $long$ on the first core. The second expression refers to the time needed to schedule the remaining tasks on the newly idle core. However, both expressions are smaller than SL_{forkN}, thus, SL_{long} reduces the overall SL. ■

Proof for (ii.b)

In this case $long$ starts execution on the same core that executed $forkN$. The remaining tasks $n_1, n_2, \dots n_N$ will be executed on the new core because of (5.1).

$$SL_{forkN} = \hat{t} + w\left(t_{forkN}\right) + \max\left\{w\left(t_{long}\right), t' - \hat{t} + \sum_{i=1}^{N} w\left(t_i\right)\right\}. \quad (5.6)$$

$$SL_{long} = \max\left\{\hat{t} + w\left(t_{long}\right), t' + w\left(t_{forkN}\right) + \sum_{i=1}^{N} w\left(t_i\right)\right\}. \quad (5.7)$$

Assume by contradiction that $\hat{t} + w(long) > t' + w(forkN) + \sum_{i=1}^{N} w(n_i)$. By doing so it is possible to deduce the following:

$$\hat{t} + w\left(t_{long}\right) > t' + w\left(t_{forkN}\right) + \sum_{i=1}^{N} w\left(t_i\right)$$
$$\Rightarrow \hat{t} + w\left(t_{forkN}\right) + \sum_{i=1}^{N} w\left(t_i\right) + \beta > t' + w\left(t_{forkN}\right) + \sum_{i=1}^{N} w\left(t_i\right)$$
$$\Rightarrow \hat{t} + \beta > t'$$

This contradicts $t' > \hat{t} + \beta$. Therefore, $\hat{t} + w(long) \leq t' + w(forkN) + \sum_{i=1}^{N} w(n_i)$. Giving:

$SL_{long} = t' + w(forkN) + \sum_{i=1}^{N} w(n_i).$

Now, it will be shown that SL_{long} is smaller than both possible values of SL_{forkN} in (5.7)., thus the 2 scenarios are presented:

- $w\left(t_{long}\right) \geq t' - \hat{t} + \sum_{i=1}^{N} w\left(t_i\right)$:

From this equation, the following can be inferred:

$$(a)\ SL_{forkN} = \hat{t} + w\left(forkN\right) + w\left(long\right)$$

$$(b)\ w\left(long\right) + \hat{t} \geq t' + \sum_{i=1}^{N} w\left(n_i\right)$$

$$\Rightarrow SL_{forkN} \underset{(a)}{=} \hat{t} + w\left(long\right) + w\left(forkn\right) \underset{(b)}{\geq} t' + \sum_{i=1}^{N} w\left(n_i\right) + w\left(long\right) = SL_{long}$$

This means that by swapping $forkN$ with $long$ the tasks schedule length is reduced or left the same.

- $w\left(t_{long}\right) < t' - \hat{t} + \sum_{i=1}^{N} w\left(t_i\right)$:

  $SL_{forkN} = t' + w(forkN) + \sum_{i=1}^{N} w(n_i) = SL_{long}$.

  Which means that changing the tasks leaves the SL as it was.

■

Proof for (ii.d):

In this case, the additional core becomes available after both $long$ and $forkN$ have completed. Therefore, it doesn't matter which of them is completed first. ■

For time partition (ii.c) it was not possible to prove the lemma as a counter example exists. As a reminder $\hat{t} + w(long) < t' \leq \hat{t} + w(long) + w(forkN)$

Now consider the following parameters for tasks.

$r = 2$

$w(long) = 10$

$w(forkN) = 5$

$N = 3$

$w(n_i) = 1, \forall i \in \{1,2,3\}$

These tasks maintain equation (5.1). At time $t = \hat{t} + 11 > \hat{t} + w(long)$ an additional core becomes idle. In Figure 33 (a), $long$ is allocated before $forkN$. In Figure 33 (b), $forkN$ is allocated before $long$ and a better schedule length is achieved. Which makes that proving the Lemma for this time partition is impossible.

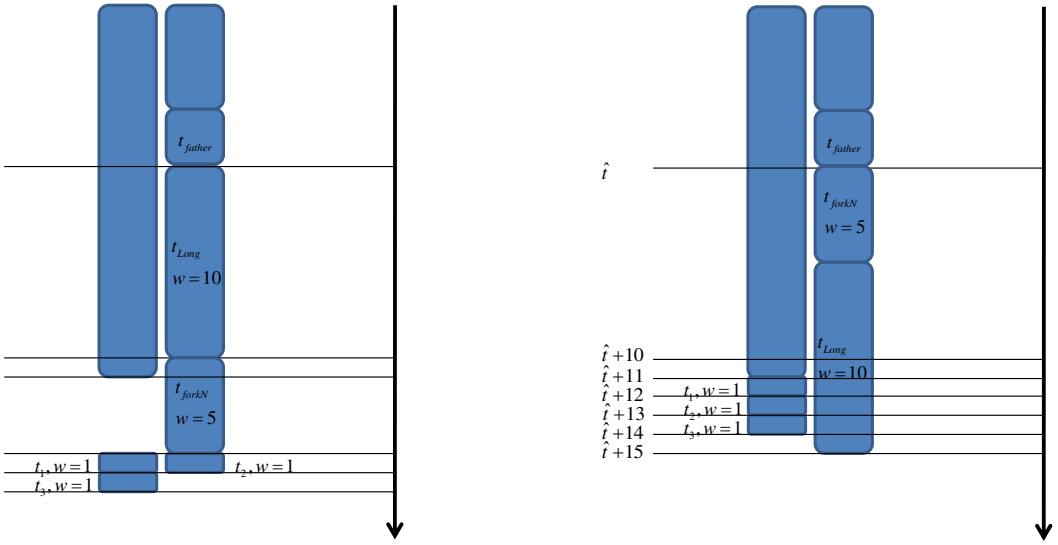


Figure 33 - Contradiction to the Lemma. (a) $Long$ is allocated before $forkN$.(b) $forkN$ is allocated before $Long$ and reduces the execution time.

6 Efficiency of cache use

The previous sections discussed scheduling directives and how they can reduce the number of cache misses for certain scenarios. These sections ignored a phenomenon that is inherent in UMA shared memory systems with no private caches. Specifically, with data parallelism multiple cores might suffer from a single cache miss.

Consider, for example, a cache line of length L bytes, and a duplicable task A whose replicas access consecutive elements of an array Arr; the size of each element of Arr is d bytes, $d < L$. Practical values for L and D are $\{32, 64\}$ and $\{1, 2, 4, 8, 16\}$ respectively. Clearly, several consecutive elements will reside within the same cache line. After multiple (consecutive) replicas are dispatched in the same cycle, their cores will try to access their appropriate cache line in the shared cache. If the cache line has already been fetched, then all the cores accessing the data will receive a cache hit. However, if the cache line has not been fetched, then multiple cores will wait on the cache miss, making this cache miss more expensive. With P waiting cores and a miss penalty of CM(Cache Miss) cycles, the number of wasted cycles is:

$$WC = P \cdot CM. \tag{6.1}$$

The upper bound on the number of wasted cycles of a specific cache miss is

$$WC = \frac{L}{d} \cdot CM, \tag{6.2}$$

where L/d refers to the maximal number of elements in the array that can fit into a single cache line. A possible solution for this example is to stagger the replica dispatching or to somehow case the relevant cache lines to be touched in ahead of time so that at most one core waits on any given cache line. The issue may be more complex. Additional parameters that affect the behavior are:

- Array dimensions
- Data granularity and the number of elements that fit into a single cache line.
- Cache
    - Size
    - Number of lines
    - Associativity
- Access pattern for a replica, for the cases that the replicas access more than a single element.

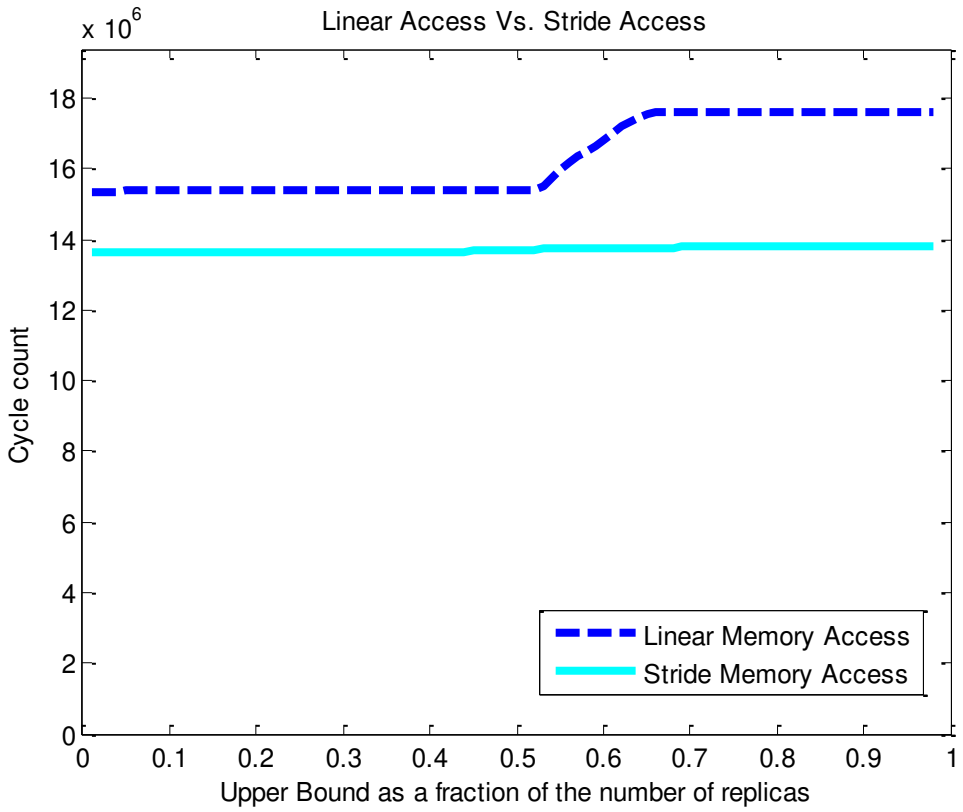


Figure 34 – Graph compares sequential memory access pattern with memory access pattern than switches between the roles of row and columns

In the following example, consider a program that processes a *2k X 2k* image (one byte per pixel) with single-pixel granularity (2^{22} replicas, each processing a single pixel). We assume a $2MB$ direct mapped cache with $L = 32$ and $d = 1$. There are thus 64K cache lines, each with data elements. The image data is arranged as a $4MB$ one dimensional array, which is larger than the cache. Rather than using the sequential access pattern that results in a large number of idle cycles, we changed the roles of the rows and the columns such that:

Sequential approach:

$$row = id \,/\, 2K$$
$$col = id \% 2K$$

New approach:

$$row = id \% 2K$$
$$col = id \,/\, 2K$$

Essentially, this enabled us to cause the sequential (ID) replicas to access data that was in different cache lines. The number of cache lines is greater than the number of rows in the matrix. Therefore, by the time that the replicas of the following column have been dispatched, all the required data is in the cache (the cache does not suffer capacity misses). We assume a 20 cycle cache miss penalty. The result is depicted in Figure 34, where the dashed curve refers to a sequential access pattern and the solid

curve refers to the new access pattern. These curves were created using the SAS directive for 2 duplicable tasks where $l_{upper} = C \cdot n$ for $0 < C < 1$ and $l_{low} = 0.2 \cdot l\_upper$. The abscissa is l_{upp}. The ordinate is the number of cycles required to complete the application. Notice that the new access pattern does not suffer as much from the redundant cache misses as SAS did. The performance gain here is in the 14%-26%.

Another way around the problem is to change the replica granularity such that all the replicas that access the same cache line will be grouped into a single replica. This approach is not preferable is portability is hurt. Nonetheless, for code that is being optimized for a specific system, this is a good work around.

It is well worth noting that by changing the memory access pattern, use of the new directives might not be possible or at the very least the input of the directives will need to be changed.

The treatment of cache related issues in this paper is only a first step. Other issues, most notably the effects of associativity and of bank-access collisions are interesting topics for further research.

7 Conclusions and future work

In conclusion, we have seen that the use of precedence constraints is not enough to enforce a desired scheduling order, offline and online. Enforcing a scheduling order offline is not simple, enforcing the order online is more difficult when maintaining fast dispatching. By using the scheduling directives that were presented in this paper, it is possible to decide on a schedule offline and have that schedule enforced online. It is the responsibility of a scheduling policy to decide on the order in which tasks are dispatched. We did not deal with scheduling policies, rather, we supplied scheduling policies a way to better express themselves and enforce an order for online scheduling.

In our work, we used Plurality's shared-memory many-core system as a reference system for the incorporation of new scheduling directives. The new scheduling directives are not only intended for Plurality's system but can be used for other systems as well.

We introduced a new scheduling directive for regular tasks, Start After Start, which:

- Increases the expressive power that the task graph designer has on the order in which tasks are allocated.
- This directive can also be applied to systems that are not shared-memory.
- This directive can be used by offline scheduling policies that can create more precise task graphs that can be enforced by online schedulers.

We also introduced several scheduling directives that deal with duplicable tasks. These directives can help reduce the number of cache misses and reduce the memory footprint of an application. We presented an example wherein the performance gain due to the decreasing in cache misses is around 15%. Some of these directives also enforce correctness while offering the aforementioned benefits. We showed how these directives could be integrated into an actual system, Plurality's system, while maintaining the low power and space envelope. We also showed a design of a "Re-Order Buffer" for duplicable tasks that is both power efficient and fast.

While we found several interesting and useful directives, there are most likely additional directives to be found. We believe that it will be possible to take several directives and create more complex directives. These directives will be dependent on the number of duplicable-task ROBs that are implemented in the hardware. We also believe that it may be possible to create 2D duplicable tasks that have some sort of internal precedence constraints (directives) by using the additional ROBs.

Although a fast and low power ROB was implemented, we believe that with a small increase in power, it might be possible to create a faster and pipelined ROB. Also, we believe that it is possible to create more efficient "Min" units. These units are connected in a certain pattern, and by using additional information on an actual system it may be possible to create better "Min" units.

We believe that some research should be done on cache line granularity for shared memory systems. We saw that a cache miss can affect multiple cores. This is not the only cache related problem we saw in this paper. It may be possible to create cache-aware scheduling directives that would reduce the number of idle cycles. This might be done using some sort of task that has one responsibility and that is to prefetch data that will be required in the near future. For short-execution replicas, a cache miss penalty may be a significant fraction of total replica execution time. Another cache related issue that we saw is the partition of a single cache line over consecutive memory banks. This issue has both pros and cons. We believe that more work should be done in defining a model that is application based.

References

[1] X. Wen, "HARDWARE DESIGN, PROTOTYPING AND STUDIES OF THE EXPLICIT MULTI-THREADING (XMT) PARADIGM," Doctor of Philosophy, Department of Electrical and Computer Engineering, University of Maryland, 2008.

[2] X. Wen and U. Vishkin, "Fpga-based prototype of a pram-on-chip processor," presented at the Proceedings of the 5th conference on Computing frontiers, Ischia, Italy, 2008.

[3] G. E. Moore, "Cramming more components onto integrated circuits (Reprinted from Electronics, pg 114-117, April 19, 1965)," *Proceedings of the Ieee,* vol. 86, pp. 82-85, Jan 1998.

[4] "HyperCore Software Developer's Handbook," ed: Plurality, 2009.

[5] D. H. Woo and H. H. S. Lee, "Extending Amdahl's Law for Energy-Efficient Computing in the Many-Core Era," *Computer,* vol. 41, pp. 24-+, Dec 2008.

[6] NVIDIA CUDA™ Programming Guide Version 3.0.

[7] NVIDIA CUDA™ Reference Manual Version 3.0.

[8] NVIDIA CUDA™ Best Practices Guide Version 3.0.

[9] G. M. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," presented at the Proceedings of the April 18-20, 1967, spring joint computer conference, Atlantic City, New Jersey, 1967.

[10] J. L. Gustafson, "Reevaluating Amdahl Law," *Communications of the Acm,* vol. 31, pp. 532-533, May 1988.

[11] M. D. Hill and M. R. Marty, "Amdahl's law in the multicore era," *Computer,* vol. 41, pp. 33-+, Jul 2008.

[12] S. Borkar, "Thousand core chips: a technology perspective," presented at the Proceedings of the 44th annual Design Automation Conference, San Diego, California, 2007.

[13] J. D. Ullman, "Polynomial complete scheduling problems," presented at the Proceedings of the fourth ACM symposium on Operating system principles, 1973.

[14] R. Graham, "Bounds for certain multiprocessing anomalies," *Bell System Technical Journal,* vol. XLV, 1966.

[15] T. L. Adam*, et al.*, "Comparison of List Schedules for Parallel Processing Systems," *Communications of the Acm,* vol. 17, pp. 685-690, 1974.

[16] I. Ahmad and Y. K. Kwok, "On exploiting task duplication in parallel program scheduling," *Ieee Transactions on Parallel and Distributed Systems,* vol. 9, pp. 872-892, Sep 1998.

[17] J. J. Hwang*, et al.*, "Scheduling Precedence Graphs in Systems with Interprocessor Communication Times," *Siam Journal on Computing,* vol. 18, pp. 244-257, Apr 1989.

[18] Y. K. Kwok and I. Ahmad, "Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors," *Ieee Transactions on Parallel and Distributed Systems,* vol. 7, pp. 506-521, May 1996.

[19] Y. K. Kwok and I. Ahmad, "Efficient scheduling of arbitrary task graphs to multiprocessors using a parallel genetic algorithm," *Journal of Parallel and Distributed Computing,* vol. 47, pp. 58-77, Nov 25 1997.

[20] Y. K. Kwok and I. Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors," *Acm Computing Surveys,* vol. 31, pp. 406-471, Dec 1999.

[21] Y. K. Kwok and I. Ahmad, "Benchmarking and comparison of the task graph scheduling algorithms," *Journal of Parallel and Distributed Computing,* vol. 59, pp. 381-422, Dec 1999.

[22] M. Maheswaran*, et al.*, "Dynamic mapping of a class of independent tasks onto heterogeneous computing systems," *Journal of Parallel and Distributed Computing,* vol. 59, pp. 107-131, Nov 1999.

[23] G. C. Sih and E. A. Lee, "A Compile-Time Scheduling Heuristic for Interconnection-Constrained Heterogeneous Processor Architectures," *Ieee Transactions on Parallel and Distributed Systems,* vol. 4, pp. 175-187, Feb 1993.

[24] A. G. T. Yang, "DSC: Scheduling Parallel Tasks on an Unbounded Number of Processors," *IEEE Transactions on Parallel and Distributed Systems,* pp. 951-967, September 1994.

[25] D. W. Gillies and J. W. S. Liu, "Scheduling Tasks with and/or Precedence Constraints," *Siam Journal on Computing,* vol. 24, pp. 797-810, Aug 1995.

[26] C. Mccreary and H. Gill, "Automatic-Determination of Grain-Size for Efficient Parallel Processing," *Communications of the Acm,* vol. 32, pp. 1073-1078, Sep 1989.

[27] P. Brucker and SpringerLink (Online service). (2007). *Scheduling algorithms (5th ed.)*.

[28] E. G. Coffman and J. L. Bruno, *Computer and job-shop scheduling theory*. New York: Wiley, 1976.

[29] O. Sinnen, *Task scheduling for parallel systems*. Hoboken, N.J.: Wiley-Interscience, 2007.

[30] R. Graham, E. Lawler, E. Lenstra, A. Rinnooy Kan, "Optimization and Approximation in Deterministic Sequencing and Scheduling: A Survey," *Annals of Discrete Mathematics,* vol. 5, pp. 287-326, 1979.

[31] T. H. Cormen*, et al.*, *Introduction to algorithms*. Cambridge, Mass.New York: MIT Press; McGraw-Hill, 1990.

[32] J. L. Hennessy*, et al.*, *Computer architecture : a quantitative approach*, 4th ed. Amsterdam ; Boston: Morgan Kaufmann, 2007.

[33] R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *Ibm Journal of Research and Development,* vol. 11, pp. 25-33, 1967.