# Decentralized in-order execution of a sequential task-based code for shared-memory architectures

Charly Castes, Emmanuel Agullo, Olivier Aumage, Emmanuelle Saillard


# Decentralized in-order execution of a sequential task-based code for shared-memory architectures

Charly Castes, Emmanuel Agullo, Olivier Aumage, Emmanuelle Saillard

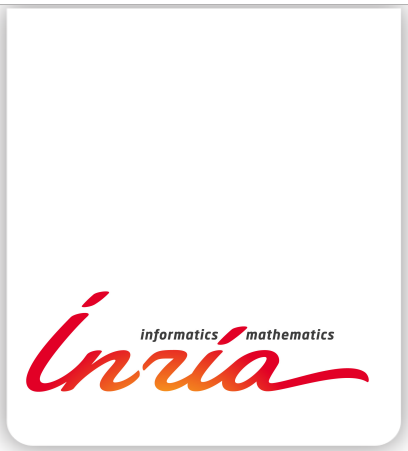
RESEARCH REPORT

N° 9450

January 2022

Project-Teams HiePACS and STORM

ISSN 0249-6399 ISRN INRIA/RR--9450--FR+ENG




# Decentralized in-order execution of a sequential task-based code for shared-memory architectures

Charly Castes, Emmanuel Agullo, Olivier Aumage, Emmanuelle Saillard

Project-Teams HiePACS and STORM

Research Report n° 9450 — January 2022 — 30 pages

**Abstract:** The hardware complexity of modern machines makes the design of adequate programming models crucial for jointly ensuring performance, portability, and productivity in high-performance computing (HPC). Sequential task-based programming models paired with advanced runtime systems allow the programmer to write a sequential algorithm independently of the hardware architecture in a productive and portable manner, and let a third party software layer —the runtime system— deal with the burden of scheduling a correct, parallel execution of that algorithm to ensure performance. Many HPC algorithms have successfully been implemented following this paradigm, as a testimony of its effectiveness.
Developing algorithms that specifically require fine-grained tasks along this model is still considered prohibitive, however, due to per-task management overhead [1], forcing the programmer to resort to a less abstract, and hence more complex "task+X" model. We thus investigate the possibility to offer a tailored execution model, trading dynamic mapping for efficiency by using a decentralized, conservative in-order execution of the task flow, while preserving the benefits of relying on the sequential task-based programming model. We propose a formal specification of the execution model as well as a prototype implementation, which we assess on a shared-memory multicore architecture with several synthetic workloads. The results show that under the condition of a proper task mapping supplied by the programmer, the pressure on the runtime system is significantly reduced and the execution of fine-grained task flows is much more efficient.

**Key-words:** Task-based programming, fine-grain, runtime system, scheduling, in-order, decentralized, shared-memory, multicore

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

# Exécution ordonnée décentralisée d'un code séquentiel à base de tâches sur une architecture à mémoire partagée

**Résumé :** La complexité matérielle des machines modernes rend la conception de modèles de programmation adéquats cruciale pour garantir à la fois la performance, la portabilité et la productivité dans le calcul haute performance (HPC). Les modèles de programmation séquentiels à base de tâches associés à des supports d'exécution avancés permettent au programmeur d'écrire un algorithme séquentiel indépendamment de l'architecture matérielle d'une manière productive et portable, et de laisser une couche logicielle tierce - le support d'exécution - s'occuper de l'ordonnancement d'une exécution parallèle de cet algorithme afin de garantir les performances. De nombreux algorithmes HPC ont ont été mis en œuvre avec succès selon ce paradigme, ce qui témoigne de son efficacité.

Cependant, le développement d'algorithmes nécessitant spécifiquement des tâches à grain fin selon ce modèle est encore considéré comme prohibitif, en raison des coûts de gestion des tâches [1], ce qui oblige le programmeur à recourir à un modèle moins abstrait et donc plus complexe de type "tâche+X". Nous étudions donc la possibilité d'utiliser modèle plus conservateur, décentralisé et traitant les tâches dans l'ordre de soumission, ainsi optimisé pour limiter le coût de gestion de tâches à grain fin quitte à perdre en réactivité par rapport aux modèles de l'état de l'art. Nous proposons une spécification formelle du modèle d'exécution ainsi qu'un prototype d'implantation, que nous évaluons sur une architecture multicœur à mémoire partagée. Les résultats montrent que sous la condition d'une association des tâches sur les unités d'exécution préalablement fournie par le programmeur, la pression sur le système d'exécution est réduite de manière significative et l'exécution d'un flux de tâches à grain fin est ainsi beaucoup plus efficace.

**Mots-clés :** Programmation à base de tâches, grain fin, support d'exécution, ordonnancement, dans l'ordre, décentralisé, mémoire partagée, multicœur

# Decentralized in-order execution of a sequential task-based code for shared-memory architectures

January 27, 2022

## 1 Introduction

Parallel computing is a requirement in HPC, to achieve the necessary level of performance. Writing correct parallel programs is a notoriously difficult task, though. Runtime systems for automatized parallelization have thus long been used as a means to offset part of this burden, and common patterns have even been standardized, see OpenMP [2]. In the last fifteen years, a new class of task-based runtime systems such as StarPU [3], PaRSEC [4], SuperGlue [5], OmpSs [6], to name a few, has been proposed to better take advantage of multicore, manycore and heterogeneous accelerated architectures. This effort resulted in a rich ecosystem of runtimes with their own different goals, guarantees, performance and programming model declinations [1]. A common trait to many of those initiatives is the ability to accept from the programmer a sequential series of tasks with implicit dependencies as the input algorithm to be parallelized. This *programming model* is sometimes referred to as *Sequential Task Flow* (STF) [7], [8]. The STF programming model is supported by OpenMP since revision 4.0 through the *task* construct and *depend* clause inspired from OmpSs, StarPU supports it, PaRSEC offers it through its *dynamic task discovery* (DTD) mode, for example.

While STF is therefore arguably a popular programming model in HPC, the per-task management overhead incurred by such runtime systems makes it prohibitive in practice to execute fine-grained tasks, as highlighted in a recent study of their performance as a function of task sizes [1]. The study estimates that on current architectures, the minimal duration of individual tasks should be on the order of $100\mu s$ for the approach to be profitable. Unfortunately, some important classes of HPC applications actually do involve tasks of small granularity. A typical example is the *High Performance Linpack Benchmark* [9] (HPL) used for establishing the TOP500 [10] supercomputer ranking. The core of the HPL algorithm is a LU matrix factorization with partial pivoting: while most operations are performed at coarse granularity, the pivoting itself requires fine-grained operations that can not be efficiently executed as tasks with such runtime systems.

Most task-based runtime systems assessed in [1] support the STF *programming model* while internally using various strategies for their *execution model*. In this paper, we further formalize the important but often implicit difference between the *programming model* and *execution model*. We highlight that most runtime systems supporting the STF programming model on shared-memory machines most often explicitly or implicitly assume a centralized, out-of-order execution model (the scheduling work possibly being decentralized, but the consistency management work remaining centralized). While this execution model may be an excellent choice for dealing with moderate or coarse grain tasks,on the contrary, this paper proposes a new lightweight execution model relying on the principles of decentralized dependency management and in-order execution, to drastically reduce per-task management overhead. We introduce a formal specification of the proposed execution model as well as a prototype implementation (for shared-memory, homogeneous multicore architectures), which we assess with synthetic workloads. The results are promising, showing that under the condition of a proper task mapping supplied by the programmer, our proposal enables a cost-effective parallel execution of algorithms with finer-grained tasks expressed in the STF programming model (the programming model itself being slightly modified because a mapping function is now additionally requested), at the expense of a potentially worse pipelining with coarser tasks. Even though we compare our model with the established centralized out-of-order paradigm our intent is not to replace the general-purpose runtimes cited earlier, but to demonstrate superior efficiency on some classes of computation involving fine granularity, and eventually enable those general purpose runtimes to delegate relevant computations to an embedded low-overhead runtime, as the one described in this paper.

Our original contributions include the execution model, its formal specification (as well as a specification of the STF programming model that the execution model must satisfy), and an analysis of the performance of a prototype implementation of that model on different synthetic benchmarks.

The paper is organized as follows. Section 2 presents some background on the STF programming model (section 2.1), typical execution models (section 2.2) employed in the HPC literature for supporting it on shared-memory machines, and our methodology to assess the efficiency of execution models (section 2.3). Section 3 introduces our proposal for a lightweight execution model implemented in our Run-in-Order (Rio) runtime system prototype, to execute sequential flows of fine-grained tasks. Section 4 presents the methodology we have employed to define the formal specification of both the STF model and the proposed execution model. Section 5 reports on experiments we conducted to assess the proposed approach. Section 6 concludes this paper.

## 2 Background

Throughout this paper we make a clear distinction between the *programming model* and the *execution model*. The programming model defines the semantic

of a program, it gives guarantees about the behavior of the program but does not specify how it is executed. Defining the precise execution of a program is the role of the execution model. It must conform to the high level semantics described by the programming model but is free to choose the underlying implementation. Decoupling the programming and execution models is important when discussing performance, because even though the programming model imposes constraints on the execution, different implementations can result in very different performance profiles.

## 2.1 The Sequential Task Flow programming model

In the STF model the programmer writes its program as a sequence of *tasks* to be executed, that we call the *task flow*. A task is a pure function in the functional programming terminology (e.g. without side effects) that can operate on some *data objects* managed by the runtime system. For each such data object, the task declares an *access mode*: read-only, write-only or read-write. The STF model gives the *sequential consistency* guarantee that the result of a valid parallel execution in this model will be the same as the result of a sequential execution of the tasks in the order given by the task flow.

The appeal of STF comes from the implicit management of data dependencies it offers: such dependencies are deduced from the access order in the task flow and the respective data access modes declared by the tasks. Sequential consistency is guaranteed by the runtime by ensuring that each read access happens after all previous write operations and that each write access happens after all previous read *and* write operations. Dependencies being implicit, writing a STF algorithm is similar to writing the sequential version of that algorithm. As a result, STF applications avoid common pitfalls of concurrent programs such as deadlocks, and data races.

## 2.2 The Execution Model

While the programming model describes the semantic of the —STF in our case— programs, the execution details within the boundaries of these semantic constraints are left for the runtime to decide. The simplest possible execution model for STF would be to execute the tasks sequentially in the order given by the task flow. While semantically correct, this execution model would make a poor usage of a parallel computer. More efficient execution models have thus been developed and are gaining momentum as an effective way to write high performance applications for supercomputers.

Multiple runtimes are compliant with the STF programming model: StarPU [3], PaRSEC [4] with Dynamic Tasks Discovery, Quark [11], SuperGlue [5], OmpSs [6] and OpenMP starting with version 4.0 [2] and the introduction of the *task* construct and *depend* clause. Within a hardware node, most STF-compliant runtimes use very similar execution models that we describe as *centralized* and *out-of-order* (OoO). We designate them as centralized because they rely on a master-worker model (especially on shared-memory architectures), in which a

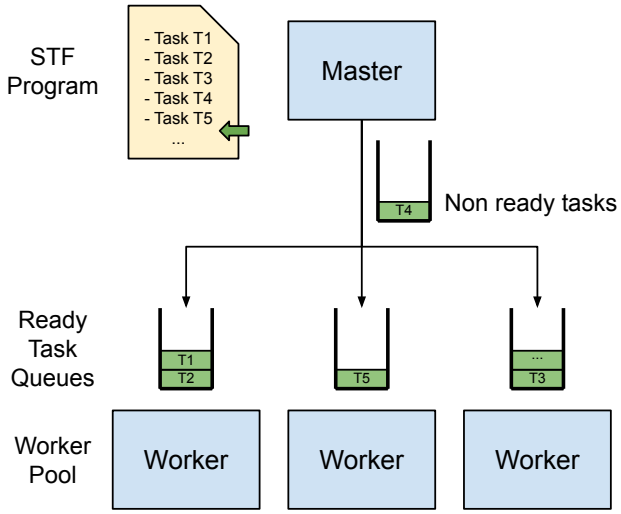



Figure 1: Illustration of a centralized out-of-order execution model. A master thread executes the STF program, producing a sequence of tasks that are dispatched to a pool of workers, using tasks queues for instance. The master thread can re-order the tasks to reduce worker idle time by taking advantage of independent task, effectively executing tasks out of their original order.

master thread unrolls the task flow to discover the tasks and dispatch them to a pool of workers (illustrated in Figure 1). In addition, the master thread (through scheduling) and/or the workers (through work stealing) can re-order the tasks to minimize idleness as long as sequential consistency is maintained. The execution is thus said to be OoO.

Centralized OoO runtimes are indeed effective. StarPU for instance is consistently achieving performance within a few percent of the best performing (possibly non STF) implementation on the Task Bench runtime survey [1]. OoO runtimes are able to make good scheduling decisions at runtime by taking into account parameters such as data locality, expected task execution time and upcoming tasks, while also dynamically balancing the workload through work stealing techniques. Those features come at the cost of higher per-task overhead, as highlighted by the Task Bench survey, which makes execution of fine-grained tasks intractable.

## 2.3 Decomposing runtime efficiency

Figure 2 shows the evolution of the execution time against the dimensions of the sub-matrices, for a matrix multiplication. It uses a state-of-the-art general matrix multiplication kernel for double precision values (DGEMM) from the Intel MKL library, together with StarPU, on a dual socket 12-core Intel Xeon E5-2680 v3 processor [12]. It illustrates the impact of granularity on the execution

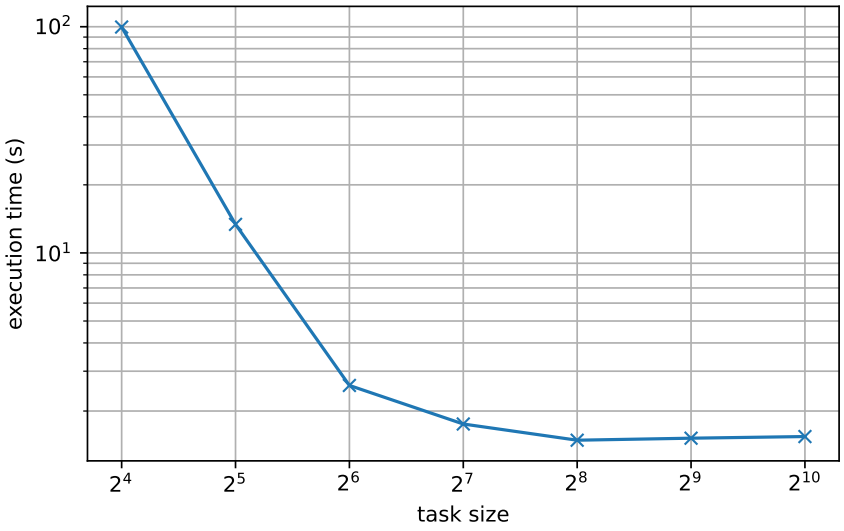



Figure 2: Execution time against task size for a 4096 by 4096 square matrix multiplication using StarPU with the Intel MKL DGEMM kernel in shared memory (24 cores). The task size corresponds to the dimensions of the square sub-matrices.

time: finer grained tasks lead to a longer execution. However, Figure 2 by itself does not explain why the efficiency decreases, which results from a combination of factors. Figure 3 shows the efficiency of the Intel MKL DGEMM routine against the matrix tile sizes when splitting the whole computation into tasks. This experiment makes it clear that the global execution time is not a good measurement of the runtime performance characteristics, since the computation kernel itself looses efficiency with smaller tasks. Matrix multiplication kernels usually exploit hardware caches efficiently on sufficiently large matrices, while dividing the computation into smaller tasks reduces opportunities for cache reuse, which in return degrades the kernel efficiency.

In this paper we investigate the impact of the runtime system on the global computation efficiency, using a methodology inspired by previous works ([13], [8] and [14]) to decompose the global efficiency into a product of efficiencies more easily attributable to specific components and properties of execution models. In the following, we use the notations:

- $t$: execution time of the fastest sequential algorithm;
- $t(g)$: execution time of the sequential algorithm when splitting the problem in tasks of granularity $g$;
- $t_p(g)$: execution time when using a runtime with $p$ threads and tasks of granularity $g$;

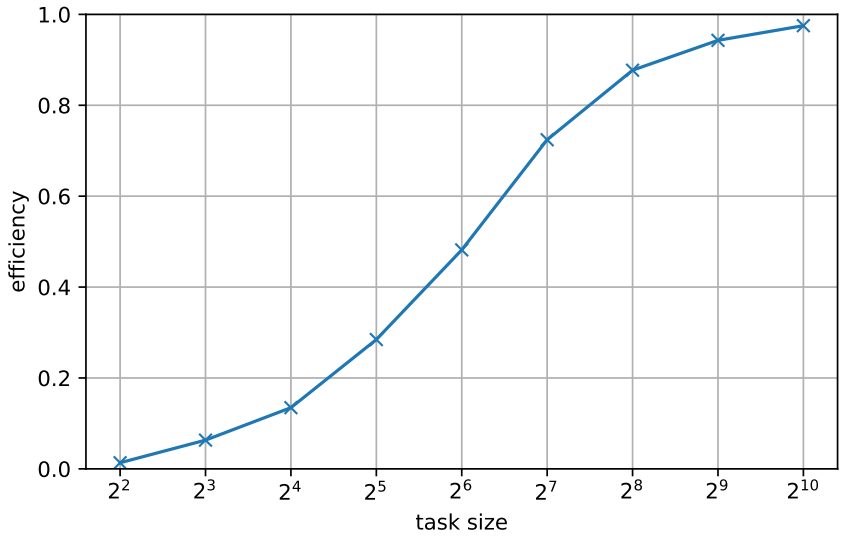



Figure 3: Sequential Intel MKL DGEMM kernel efficiency as a function of the task size, in this case the dimension of sub-matrices.

- $e$: *parallel efficiency*[15].

$$e(g) = \frac{t}{p\ t_p(g)}.$$

As discussed, the parallel efficiency encapsulates not only the cost of the runtime but also overheads such as the reduced efficiency of tasks' computation kernels at a given granularity. In our analysis, we thus want to isolate the efficiency of the computation kernel from the efficiency of the runtime itself. To that effect we further refine our notations by introducing the cumulative execution time using a runtime $\tau_p(g) = p\ t_p(g)$ and decomposing it into three parts depending on the type of event occurring at a given instant:

- $\tau_{p,t}(g)$: cumulative time spent executing tasks;
- $\tau_{p,i}(g)$: cumulative time spent idle, waiting for a dependence constraint to be resolved, for instance;
- $\tau_{p,r}(g)$: cumulative time spent in the runtime not executing a task nor idle, which corresponds to the management cost of tasks (e.g. memory allocation, scheduling).

The sum of these cumulative times corresponds to the total parallel execution time multiplied by the number of threads: $\tau_p(g) = \tau_{p,t}(g) + \tau_{p,i}(g) + \tau_{p,r}(g)$. This can be viewed as a rectangle of height $p$ and width $t_p$ being covered by

events among the above three possible types (processing tasks, idle, internal runtime management).

Using these notations we decompose the parallel efficiency $e$ into a product of four efficiencies: the granularity efficiency $e_g$ representing the efficiency of the computation kernel at a given granularity, the locality efficiency $e_l$ encapsulating the effect of locality in a multi-threaded application, the pipelining efficiency $e_p$ for the ability of the runtime to efficiently pipeline tasks execution, and the runtime efficiency $e_r$ representing the overhead of managing tasks in the runtime. Introducing $t(g)$ the sequential time when operating at granularity $g$, we can indeed write:

$$\begin{aligned} e(g) =& \frac{t}{p\, t_p(g)} \\ =& \frac{t}{t(g)} \times \frac{t(g)}{\tau_{p,t}(g)} \times \frac{\tau_{p,t}(g)}{\tau_{p,t}(g) + \tau_{p,i}(g)} \\ & \times \frac{\tau_{p,t}(g) + \tau_{p,i}(g)}{\tau_{p,t}(g) + \tau_{p,i}(p) + \tau_{p,r}(g)} \\ =& e_g(g)\, e_l(g)\, e_p(g)\, e_r(g), \end{aligned}$$

where:

$$\begin{aligned} e_g(g) &= \frac{t}{t(g)}; \\ e_l(g) &= \frac{t(g)}{\tau_{p,t}(g)}; \\ e_p(g) &= \frac{\tau_{p,t}(g)}{\tau_{p,t}(g) + \tau_{p,i}(g)}; \\ e_r(g) &= \frac{\tau_{p,t}(g) + \tau_{p,i}(g)}{\tau_{p,t}(g) + \tau_{p,i}(g) + \tau_{p,r}(g)}. \end{aligned}$$

Figure 4 shows the efficiency decomposition using StarPU for a matrix multiplication. The granularity efficiency is independent of the runtime. It corresponds to the efficiency pictured in Figure 3 when measured in isolation. We observe a small runtime overhead ($e_r < 1$) due to the StarPU execution model in which one of the thread is exclusively dedicated to the runtime. The parallel efficiency $e_p$ is maximized with middle-sized granularities: enough to expose parallelism without flooding the runtime. Finally, the locality efficiency can either slow down the computation in memory bound regime or speed it up beyond what is possible in single-threaded application ($e_l > 1$) by leveraging multiple caches. We use this decomposition in Section 5 to analyse the performance of different execution models for several granularities.

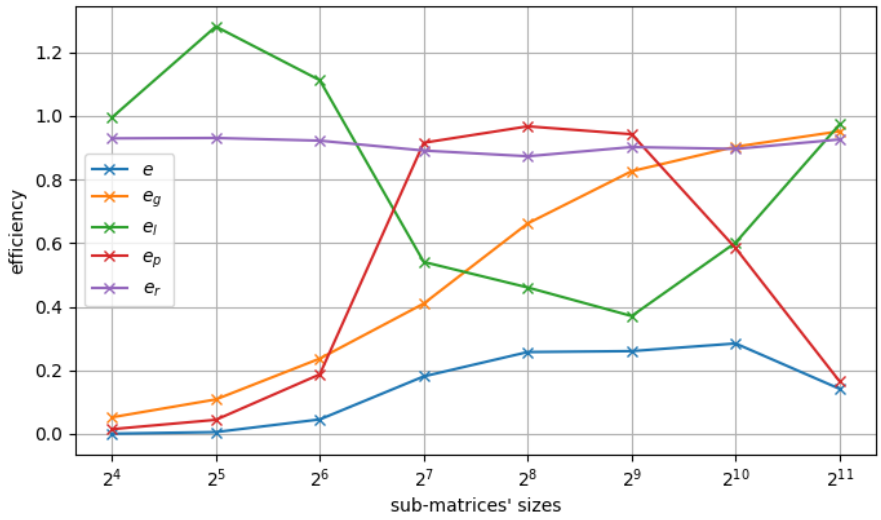



Figure 4: Efficiency decomposition on a 4096 by 4096 square matrix multiplication with StarPU (24 threads).

# 3 A lightweight execution model

Runtime systems such as StarPU are designed for the execution of "reasonably" coarse tasks. They are built around a rich centralized OoO execution model using advanced heuristics for dynamic decisions. This model achieves good pipelining efficiency as long as the per-task overhead is negligible compared to the cost of executing the task. This assumption no longer holds with smaller tasks, however. In this section, we propose an alternative decentralized in-order execution model optimized for small granularity, for which we will assess a minimal implementation in section 5.

## 3.1 In-order execution

In OoO execution models, tasks can be freely re-ordered as long as sequential consistency holds. A smart OoO scheduler can take advantage of that to yield better computation overlapping and reduce idle time. The gains from OoO scheduling come from the ability to execute ready tasks while other tasks are waiting for a dependency, which can produce efficient execution even if the order of task submissions in the task flow is not optimal. The overhead of OoO execution is due to both the need for good (hence expensive) heuristics and the necessary data structures used to store pending tasks, whose space requirement is linear in the number of tasks.

To handle a high volume of fine-grained tasks, we propose to use an in-order execution model rather than traditional OoO. An in-order execution model removes the need for scheduling heuristics and task storage, drastically reducing

the per-task overhead at the cost of a much higher sensitivity to task submission order. The scheduler in OoO models is also responsible for resources allocation, and often aims at maximizing data locality. In our proposed in-order execution model there is no dynamic scheduler; the assignment of tasks to resources must thus be done through other means.

## 3.2 Task mapping

We propose to rely on a static mapping of the tasks to do so. For some classes of computation, including most popular numerical algorithms, there has been extensive research on efficient static scheduling, such as 2d-block cyclic mapping in dense linear algebra [16] or proportional mapping in sparse linear algebra [17], [18], which can be leveraged to write efficient task mapping and discovery order. Static mappings have also been used in a distributed-memory task-based context [7], [19]. Although such mappings have been much more often considered for designing distributed-memory algorithms, nothing prevents one to translate them to the shared-memory case. It is to be noted that in the case the mapping is collected from the application, it also slightly changes the programming model, as an additional information (the mapping) is requested to write the algorithm. However, the automatic computation of static mappings has also been considered [20]. We advocate that, although less convenient than the original STF model relying on dynamic scheduling, the additional constraint of providing (or computing) a task mapping may be viewed as reasonable in HPC where there is already a well established expertise of optimizing mappings in a distributed context. In any case, this is the assumption we assess in this paper. In our prototype, we use *parametric resources allocation*: we ask the programmer for an explicit mapping from tasks to compute units in the form of a closure of type $\mathit{TaskID} \rightarrow \mathit{WorkerID}$. This enables taking advantage of application knowledge at no runtime cost.

## 3.3 Decentralized task management

Centralized runtimes rely on a master-workers model in which a single thread is responsible for unrolling the task flow and managing dependencies, while delegating task execution to a worker pool. This model makes sense when the task execution time is much greater than the unrolling and management cost, but the master thread can become a bottleneck with smaller tasks. The total execution time $t_p(g)$ can be modelled, in first approximation, as a function of the time spent in the runtime *per task* $t_r$ and the task execution time $t_t(g)$:

$$t_{p,centralized} = \max\left(n\, t_{r,centralized}, \frac{n\, t_t(g)}{w}\right), \tag{1}$$

where $n$ is the number of tasks to execute and $w$ the number of worker threads. With coarse tasks the application is limited by the speed at which the workers execute the tasks, but at smaller granularity the master thread may become the bottleneck.

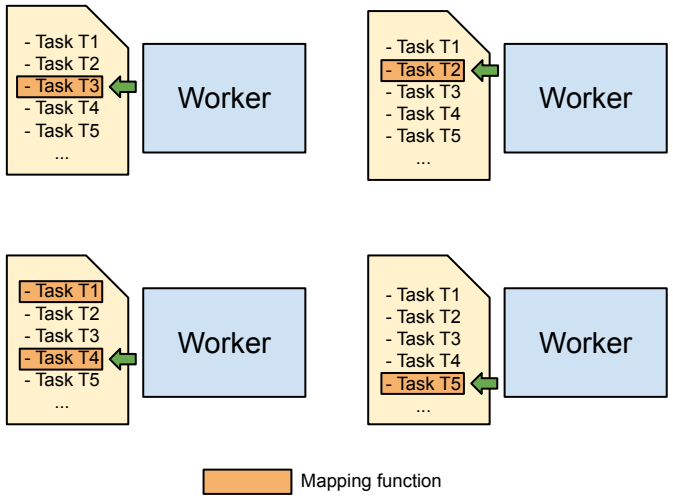



Figure 5: Illustration of a decentralized in-order execution model. All the workers execute the STF program to produce the sequence of tasks but only execute the tasks attributed to them by a deterministic mapping function. The workers can make progress independently, synchronization is only needed when there is a dependency between tasks executed by different workers.

We propose to use a decentralized execution model instead: all the threads have symmetric roles, they all unroll the whole task flow, while only executing tasks assigned to them through the mapping function (see section 3.2). The model is illustrated in Figure 5. We also present an algorithm for cheap decentralized data synchronization in section 3.4. With this model the total execution cost can be modelled as:

$$t_{p,decentralized} = n\, t_{r,decentralized} + \frac{n\, t_t(g)}{w} \tag{2}$$

Cost model (2) is obviously worse than model (1), all things being equal. In practice the runtime cost per task $t_r$ is different for the two execution models: in a centralized runtime the master thread has to perform expensive operations for each task, including updating data structures, scheduling and dispatching tasks, whereas a worker in decentralized model can simply skip over the tasks executed by other workers, leading to a much lower runtime cost. In the algorithm we present in section 3.4, the runtime cost of a task not assigned to the thread boils down to one or two writes in private (non shared) memory per dependency, depending on the access modes. The decentralized execution model combined with cheaper management costs is not affected by the bottleneck effect introduced by the master thread. Figure 6 illustrates this behavior by reporting the execution times of a minimalist program (executing a fixed number of tasks with no dependencies consisting in incrementing a counter),

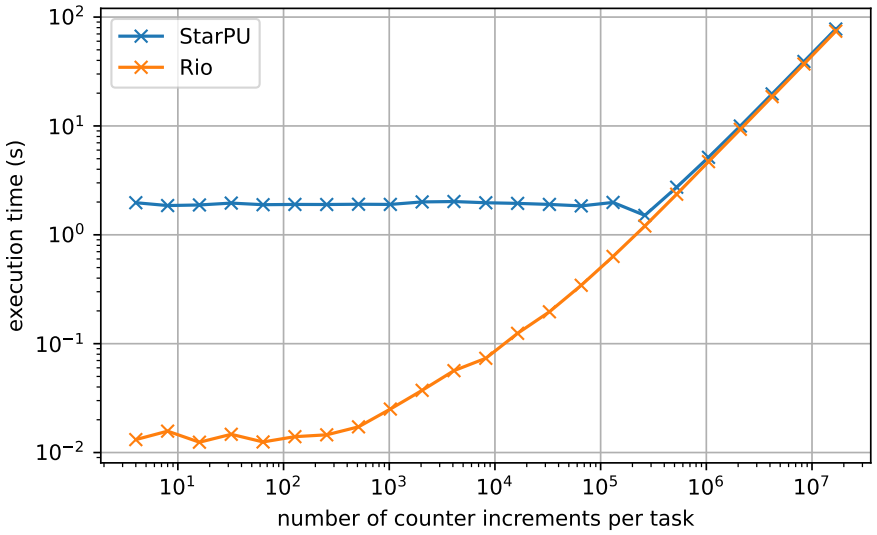



Figure 6: Execution time of a program executing a fixed number of tasks with no dependencies consisting in incrementing a counter, with the centralized runtime StarPU, and with our minimal decentralized runtime Rio.

first with StarPU (a centralized runtime) and then with Rio, our minimal decentralized runtime, for different task sizes. The cost of runtime management quickly dominates in StarPU, for which the centralized cost model (1) is accurate in the prediction of a bottleneck for small granularities. We discuss possible improvements to mitigate the worse theoretical complexity of the decentralized model in section 3.5.

## 3.4 Decentralized data synchronization

Without a master thread to coordinate workers, a new protocol is needed to ensure data accesses are properly synchronized and respect the sequential consistency ordering imposed by the STF model. Such a distributed protocol is actually commonly used by task-based runtime systems (including StarPU) on distributed-memory machines [7], [19], where there is typically one master thread per hardware node: each master thread delegates the handling of tasks mapped on the node to the node workers, but the master threads of all the nodes have to coordinate with each other. We adapt this approach into a shared-memory algorithm defining a light-weight protocol for synchronizing data accesses in a decentralized in-order execution model. We present this approach in algorithm 1, which we further introduce in the remaining of this section.

We make the following assumptions:

1. Tasks are numbered in the order in which they appear in the control flow, that number is called the *Task ID*.

2. All the threads discover the *same* sequence of tasks, i.e. the tasks have the same ID and dependencies and are delivered in the task flow order for all threads.

3. All the threads have access to a *mapping function* that deterministically associates a *Task ID* to a unique thread.

A shared-memory region is managed by a *data* object, composed of both a thread-local and a shared state. Accesses to the latter must be properly synchronized. To keep pseudocode concise, algorithm 1 supposes there is a single data object. The local state contains two integer values: `local.nb_reads_since_write` corresponding to the number of read operations encountered by the thread on this shared-memory region (but maybe not yet executed) since the last write, and `local.last_registered_write` which is the Task ID of the last write operation encountered on this memory region. The shared state also contains two integers: `shared.nb_reads_since_write` holding the number of reads *performed* on the shared-memory region since last write, and `shared.last_executed_write` containing the Task ID of the last write operation *performed* on the memory region.

Finally we define a set of routines for the *data* object that manipulates the local and shared states. Each routine exists in two versions: read or write. The appropriate version must be called depending on the access mode requested by the task (lines 4 & 12 in algorithm 1). We replace *read* or *write* by *op* in the following routines (detailed in algorithm 2):

- `declare_op`: declare an operation in `op` mode but *does not* execute it on the current thread. This only requires to modify the local state.

- `get_op`: return a pointer to the data for use in `op` mode. This operation might be blocking: it can only return once all dependencies have been resolved, which may require reading the shared state and potentially waiting for other threads.

- `terminate_op`: declare that an operation in `op` mode has been executed. This modifies the shared state.

Given these definitions, to synchronize accesses to a shared-memory location through a data object all the threads must iterate over the list of tasks. For each task in which the memory location is involved, the thread calls the *mapping* function (line 3 in algorithm 1) to get the identifier of the thread responsible for that task. If the thread is assigned to the task it calls `get_op` (lines 6 & 14) to get access to the memory location, performs the task and then releases the memory location with `terminate_op` (lines 8 & 16). If the thread is not responsible for the task, it updates its local state by calling the `declare_op` function (lines 10 & 18).

A read-only operation can be executed if `local.last_registered_write` is equal to `shared.last_executed_write` of the data object (algorithm 2, lines

**Algorithm 1:** Decentralized Data Synchronization

```
 1: for all threads do
 2:    for all task in TaskFlow do
 3:       executor ← mapping(task)
 4:       if task has read dependency then
 5:          if executor = self then
 6:             data ← get_read()
 7:             /* data can be used in read mode here */
 8:             terminate_read()
 9:          else
10:             declare_read()
11:          end if
12:       else if task has write dependency then
13:          if executor = self then
14:             data ← get_write()
15:             /* data can be used in write mode here */
16:             terminate_write(task.id)
17:          else
18:             declare_write(task.id)
19:          end if
20:       end if
21:    end for
22: end for
```

12 & 13), this ensures that all the required writes have been performed on the data. A write operation has to check that `local.last_registered_write` and `shared.last_executed_write` are the same and the number of reads since that write in the local and shared `nb_reads_since_write` variables are equal (algorithm 2, lines 17 to 20). This asserts that all the previous reads and writes have been performed on the data.

A notable property of algorithm 1 is its small runtime overhead, both in time and space. A data object requires 2 integers in the shared state plus 2 integers per worker for synchronization, independently from the number of tasks. In contrast with centralized execution models, threads can make progress independently as long as they are not blocked waiting for a dependency. Coupled with very small per-task overhead when the thread is not responsible for executing the task (a single write in private memory per data object for a read operation, two writes in private memory for a write operation), the decentralized model is not subject to the bottleneck observed with centralized runtimes (section 3.3) caused by workers waiting for the master thread to dispatch the tasks.

An extended variant of this algorithm is used for dependence management in the centralized, OoO task-based runtime system SuperGlue [5]. It introduces the notion of data versioning [21], where a new *version* of a piece of data is created upon a write by a task, and lets task dependencies be expressed as references to specific versions of some pieces of data. It enables expressing additional constructs beyond the strict sequential consistency of STF, such as reductions.

**Algorithm 2:** Decentralized Data Synchronization Routines

```
 1: function declare_read() do
 2:     local.nb_reads_since_write ←
 3:         local.nb_reads_since_write + 1
 4: end function
 5:
 6: function declare_write(task_id) do
 7:     local.nb_reads_since_write ← 0
 8:     local.last_registered_write ← task_id
 9: end function
10:
11: function get_read() do
12:     wait for local.last_registered_write =
13:         shared.last_executed_write
14: end function
15:
16: function get_write() do
17:     wait for local.last_registered_write =
18:         shared.last_executed_write
19:     wait for local.nb_reads_since_write =
20:         shared.nb_reads_since_write
21: end function
22:
23: function terminate_read() do
24:     shared.nb_reads_since_write ←
25:         shared.nb_reads_since_write + 1
26:     declare_read()
27: end function
28:
29: function terminate_write(task_id) do
30:     shared.nb_reads_since_write ← 0
31:     shared.last_executed_write ← task_id
32:     declare_write(task_id)
33: end function
```

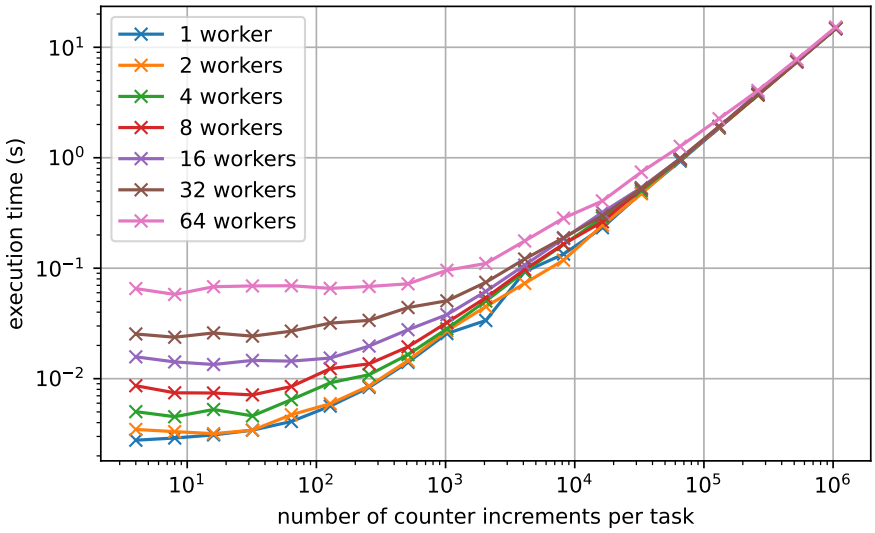



Figure 7: Total execution time of $2^{15} \approx 32000$ independent tasks per worker consisting in incrementing counters.

## 3.5 Task pruning

The main drawback of the decentralized model is that the work of unrolling the task flow is duplicated on all the workers. Scaling the number of tasks with the number of workers increases the overhead, because each worker has to process the tasks of all workers. Figure 7 illustrates this behavior. It reports the total execution time of $2^{15}$ independent tasks per worker consisting in incrementing counters, on a 64 cores AMD EPYC 7702 chip. Since all workers discover all the tasks, more tasks to execute translates into more time spent by workers in managing tasks and dependencies. Depending on the number of workers and task granularity, the overhead incurred might be negligible, as might be the case in a hypothetical a centralized OoO model runtime delegating fine-grained tasks to an embedded decentralized in-order runtime on a subset of workers.

In case the runtime overhead becomes intractable because of a high volume of extremely fine-grained tasks, an application-specific solution is to use *task pruning*. Task pruning for STF has been successful in distributed-memory settings [7]. It consists in having each entity (worker or master depending on the execution model) unrolling only the relevant part of the task flow. The effectiveness of task pruning depends on the application and the density of the dependency graph, but for common and well known applications such as dense linear algebra the gains can be substantial.

# 4 Formal specification

In addition to the algorithm described in section 3 and a concrete implementation of the decentralized in-order execution model, we propose a formal specification of the model in $TLA^+$ [22]. This formalism allows us to precisely (1) distinguish the programming model from the execution model and (2) define the proposed model in terms independent from the proposed implementation. In addition, although model checking is subject to combinatorial explosion and is intrinsically limited to the assessment of very small test cases, it may still provide further confidence on the assessed model (as a complement to the — necessarily non exhaustive — at scale actual experiments we will discuss later on in section 5).

The specification consists in two modules: a specification of the STF model and a specification of our Run-In-Order execution model which must comply with this STF specification. For a matter of conciseness, we only present here the methodology we have followed together with the illustration of a particular property, and we report to appendix B for an exhaustive specification.

The STF module describes all the possible executions of a STF program for a given set of workers, data, tasks and task flow. By giving concrete values to these variables, tools such as the $TLA^+$ model checker, $TLC$ [23], can be used to verify that some properties hold for any possible execution. We illustrate it with the termination property. In the STF specification, termination is defined as any state in which the union of active tasks (tasks that a worker is actively executing) and pending tasks (tasks not yet executed or being executed by a worker) is empty.

$$Terminated \triangleq pendingTasks \cup activeTasks = \{\}$$

The STF specification also defines a data-race freedom property that is satisfied as long as long as no pair of workers are executing tasks with a dependency on the same data and one of the tasks performing a write to that data. There is no property enforcing the sequential consistency in the STF specification. Instead, it is encoded in the state transition by exclusively allowing states to be reached for which sequential consistency holds. We report to appendix B.1 for an exhaustive specification of the STF model.

The Run-In-Order module describes all possible execution for the in-order execution model presented in this paper. In addition to the workers, data, tasks and task flow variables, an additional mapping variable is used to attribute tasks to workers. The state transition is further restricted to prevent workers from reordering their tasks. The only property checked against the Run-In-Order model is that it implements the STF specification, that is the set of executions allowed by the Run-In-Order model is a subset of all possible STF executions. Because the STF model is checked to verify termination and data-race freedom and ensures sequential consistency, checking the Run-In-Order model also ensures those properties. Appendix B.2 gives the full specification of the execution model.

Table 1: Number of states found and execution time of TLC to check the STF and Run-In-Order models on the LU factorization algorithm with different matrix sizes (number of row × column blocks).

| | **STF** | | | **Run-In-Order** | | |
|---|---|---|---|---|---|---|
| Size | Generated States | Distinct States | Time | Generated States | Distinct States | Time |
| $2 \times 2$ | 445 | 23 | 1s | 2322 | 11 | 1s |
| $3 \times 2$ | 54 481 | 94 | 11s | 1 847 877 | 29 | 56s |
| $3 \times 3$ | 542 753 065 | 655 | 22h27min | - | - | >48h |

Using the *TLC* model checker we checked the correctness of the STF and Run-In-Order specifications by emulating a tiled LU matrix factorization using two workers. The results for different sizes using *TLC* are reported in table 1. The exponentially growing number of tasks only allows us to assess very small test cases. We nonetheless found no errors during model checking and obtained a low state collision probability of at most $1.9 \times 10^{-8}$, giving us some confidence in the correctness of the proposed models.

# 5 Performance evaluation

For evaluating the ability of a decentralized in-order runtime to efficiently execute tasks of fine granularity, we have implemented the specifications proposed in section 3 within our new Rio runtime. We compare it against StarPU, a state-of-the-art runtime whose default execution model within a node is centralized OoO. The experiments have been conducted on a dual socket 12 cores Haswell Intel Xeon E5-2680 v3 [12].

## 5.1 Methodology

We consider four test cases to assess our method:

- Experiment 1 (Fig. 8, row 1) uses in independent tasks;
- Experiment 2 (Fig. 8, row 2) uses random read and write dependencies (128 data objects with 2 random read and 1 random write dependencies per task);
- Experiment 3 (Fig. 8, row 3) uses the matrix multiplication dependency graph; and
- Experiment 4 (Fig. 8, row 4) uses the dependency graph of a LU factorization without pivoting.

As illustrated above with the matrix multiplication (Figure 3), the efficiency of the considered kernels executed by the tasks may be sensitive to the effects of

the granularity and of the locality, which are orthogonal to the issues we focus on in the present study. When operating at low granularity, the dumping of the traces notifying all the events which would allow one to remove such effects in post-processing would have a non negligible impact on the overall performance. Instead, we chose to substitute each actual task with a synthetic task, common for all the tasks of the four experiments. This common synthetic task consists in incrementing a counter:

```
volatile uint64_t counter = 0;
for (uint64_t i = 0; i < N; i++)
  counter = i;
```

Using this kernel, we get a granularity efficiency $e_g(g) = 1$ as incrementing a single counter up to $N$ is almost exactly as long as incrementing $n$ counters up to $N/n$. Also, because the only relevant memory location lives on the thread's stack, the locality efficiency also becomes irrelevant: $e_l(g) = 1$.

With this kernel the experiments become sensitive only to the two remaining efficiencies, $e_p(g)$ and $e_r(g)$, the ones of interest for our study. They depend on the cumulative time spent executing tasks $\tau_{p,t}(g)$, idle $\tau_{p,i}(g)$ and the total cumulative execution time $\tau_p(g)$. Because there is no locality effect, $\tau_{p,t}(g)$ is equal to the execution time for the same sequence of tasks on a single computation unit without runtime, $t(g)$, and the total measured execution time $t_p(g)$ can be used to trivially derive $\tau_p(g)$. As Rio uses mutexes for synchronization, the idle time can be obtained with non-intrusive measurements from the CPU time share, while StarPU offers lightweight built-in online performance monitoring tools for measuring idle time that does not require to dump a trace. Measurements in StarPU are intrusive and do incur a small overhead, but because StarPU has a parallel efficiency close to zero due to the bottleneck effect with fine granularity, that overhead is negligible in our experiments.

All in all, the four experiments we conducted therefore correspond to the actual task graphs of the considered test cases but the tasks themselves are synthetically generated.

## 5.2 Results

The results of the four experiments are shown in Figure 8. Centralized OoO and decentralized in-order execution models indeed exhibit very different performance profiles: StarPU demonstrates very good and consistent performance for coarse tasks on all four experiments while Rio is much more sensitive to the dependency graph, especially when no appropriate mapping and task ordering can be given, as with random dependencies (experiment 2).

The runtime overhead of StarPU is almost independent from task sizes and explained by the fact that one of the thread is completely dedicated to task management, capping the maximal theoretical runtime efficiency to $\frac{p-1}{p}$ when running on p threads. When tasks get small, between $10^5$ and $10^6$ instructions for StarPU, the centralized model starts struggling to handle all the tasks: the master thread is not able to produce enough tasks to feed all the workers, who

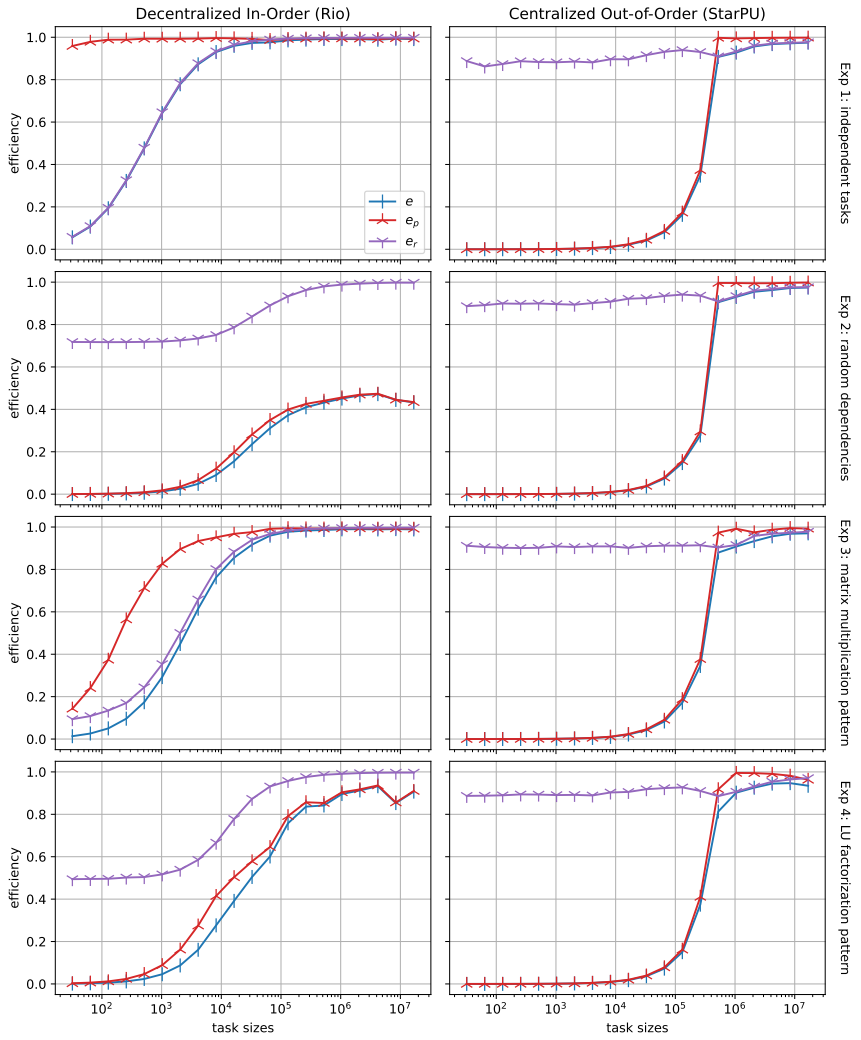



Figure 8: Efficiency decomposition as a function of task sizes for a decentralized in-order runtime (Rio) and a centralized OoO runtime (StarPU) on different task graphs.

are then forced to enter idle mode leading to the observed drop in pipelining efficiency. Decentralized models do not have this weakness because the workers independently process the task flow. With Rio, we observe that the execution becomes limited either by the pipelining or by the runtime efficiency depending on the task graph. If the number of synchronizations required is low or mainly for read operations (experiments 1 and 3), the time spent by the runtime for processing the task flow is the main source of slowdown, but thanks to the efficient in-order execution, the overhead is still reasonable even for very fine tasks of $10^3$ to $10^4$ operations. When more synchronization are needed (experiments 2 and 4), the time spent waiting for dependencies becomes the main source of total execution time.

# 6 Conclusion

While most modern STF runtimes rely on centralized OoO execution models when dealing with shared-memory machines, other models are possible. In particular, inefficiency in handling fine-grained tasks was previously considered as a limitation of the STF programming model itself, while we showed it can in fact be attributed to the centralized execution model used de facto in current implementations. We have proposed and assessed an alternative decentralized in-order execution model, on top of an enriched (with the additional requirement to provide a static mapping) STF model. This execution model achieves a higher level of performance in the special case of fine-grained tasks, thanks to lower runtime overhead and independent task flow unrolling. By drawing a distinction between the programming and execution models, we demonstrate that the case of small granularity is not an intrinsic limitation of the STF model itself and we hope that the present study might motivate future work combining both execution models (and thus requiring only partial mappings) for enabling efficient and portable implementations of wider classes of algorithms within the STF programming model.

# References

[1] E. Slaughter, W. Wu, Y. Fu, L. Brandenburg, N. Garcia, W. Kautz, E. Marx, K. S. Morris, Q. Cao, G. Bosilca, *et al.*, "Task bench: A parameterized benchmark for evaluating parallel runtime performance," in *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2020.

[2] A. OpenMP, "Openmp application program interface version 4.0," in *The OpenMP Forum, Tech. Rep*, 2013.

[3] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "Starpu: A unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience*, vol. 23, no. 2, 2011.

[4] C. Cao, T. Herault, G. Bosilca, and J. Dongarra, "Design for a soft error resilient dynamic task-based runtime," in *International Parallel and Distributed Processing Symposium*, IEEE, 2015.

[5] M. Tillenius, "Scientific computing on multicore architectures," Ph.D. dissertation, Uppsala Universiteit, 2014.

[6] A. Duran, E. Ayguadé, R. M. Badia, J. Labarta, L. Martinell, X. Martorell, and J. Planas, "Ompss: A proposal for programming heterogeneous multi-core architectures," *Parallel processing letters*, vol. 21, no. 02, 2011.

[7] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[8] E. Agullo, A. Buttari, A. Guermouche, and F. Lopez, "Implementing multifrontal sparse solvers for multicore architectures with sequential task flow runtime systems," *Acm transactions on mathematical software (toms)*, vol. 43, no. 2, 2016.

[9] J. J. Dongarra, P. Luszczek, and A. Petitet, "The linpack benchmark: Past, present and future," *Concurrency and Computation: practice and experience*, vol. 15, no. 9, 2003.

[10] J. J. Dongarra, H. W. Meuer, E. Strohmaier, *et al.*, "Top500 supercomputer sites," *Supercomputer*, vol. 13, 1997.

[11] A. YarKhan, J. Kurzak, and J. Dongarra, "Quark users' guide: Queueing and runtime for kernels," *University of Tennessee Innovative Computing Laboratory Technical Report ICL-UT-11-02*, 2011.

[12] *Haswell intel® xeon® e5-2680 v3 @ 2,5 ghz*, `https://ark.intel.com/content/www/us/en/ark/products/81908/intel-xeon-processor-e5-2680-v3-30m-cache-2-50-ghz.html`, Accessed: 2021-09-06.

[13] S. Nakov, "On the design of sparse hybrid linear solvers for modern parallel architectures," Ph.D. dissertation, Université de Bordeaux, 2015.

[14] E. Agullo, O. Aumage, B. Bramas, O. Coulaud, and S. Pitoiset, "Bridging the gap between openmp 4.0 and native runtime systems for the fast multipole method," Inria, Tech. Rep., 2016.

[15] H. Casanova, A. Legrand, and Y. Robert, *Parallel Algorithms*, ser. Chapman & Hall/CRC Numerical Analysis and Scientific Computing Series. CRC Press, 2008, ISBN: 9781584889465. [Online]. Available: `https://books.google.fr/books?id=7Z7SBQAAQBAJ`.

[16] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, *et al.*, *ScaLAPACK users' guide*. SIAM, 1997.

[17] A. George, J. W. Liu, and E. Ng, "Communication results for parallel sparse cholesky factorization on a hypercube," *Parallel Computing*, vol. 10, no. 3, 1989.

[18] A. Pothen and C. Sun, "A mapping algorithm for parallel sparse cholesky factorization," *SIAM Journal on Scientific Computing*, vol. 14, no. 5, 1993.

[19] J. Lee and M. Sato, "Implementation and performance evaluation of xcalablemp: A parallel programming language for distributed memory systems," in *International Conference on Parallel Processing Workshops*, 2010.

[20] E. Agullo, O. Beaumont, L. Eyraud-Dubois, and S. Kumar, "Are static schedules so bad? a case study on cholesky factorization," in *2016 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2016.

[21] A. Zafari, M. Tillenius, and E. Larsson, "Programming models based on data versioning for dependency-aware task-based parallelisation," in *International Conference on Computational Science and Engineering*, 2012.

[22] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 16, no. 3, 1994.

[23] ——, *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, Jun. 2002. [Online]. Available: `https://www.microsoft.com/en-us/research/publication/specifying-systems-the-tla-language-and-tools-for-hardware-and-software-engineers/`.

# A Artifact description

## A.1 Software availability and dependencies

The source code of Rio and all experiments are available on Github at `https://github.com/CharlyCst/rio`. The dependencies necessary for reproducing the results described in this paper are listed in table 2.

| **Tool** | **Version** |
|---|---|
| GCC | 10.2.0 |
| Rust toolchain | 1.51 |
| StarPU | 1.3.8 |
| hwloc | 2.4.1 |
| python | 3.9.6 |
| matplotlib | 3.4.2 |

Table 2: Software dependencies.

## A.2 Benchmarking and reproducing figures

The repository contains a *scripts* folder containing python scripts for benchmarking and displaying the collected data. All the scripts accept a *-h* or *–help* argument that can be used to learn about configuration options.

The following commands reproduce the efficiency graphs:

```
$ python scrips/benchmark_efficiency.py \
  -f efficiency.json --timeout 100      \
  --average-on 3 --nb-threads 24        \
  --experiment counter_deps --verbose
$ python scripts/plot_efficiency.py     \
  -f efficiency.json
```

To plot the execution time for different number of workers (on a 64 cores machine), run:

```
$ python scripts/benchmark_workers.py   \
  -f workers.json --timeout 100         \
  --average-on 3 --nb-threads 6         \
  --size 22 --task-size 20              \
$ python scripts/plot_workers.py        \
  -f workers.json
```

To obtain the results presented in this paper, we ran the efficiency experiments on a dual 12 cores Haswell Intel Xeon E5-2680 v3 node and the workers benchmark on a 64 cores AMD EPYC 7702 chip.

# B Formal Specification

The formal specifications discussed in section 4 are located in the *specs* folder of the repository and are reproduced in this appendix. In addition to the specifications, two models are available for checking the specification of both the STF and Run-In-Order on a STF program emulating a LU matrix factorization for different matrix sizes.

Running the *TLC* model checking on these models requires a Java installation (we used Java 11.0). *TLC* can be downloaded using the scripts in the *specs* folder:

```
$ ./download_tla.sh
```

Then the STF model can be checked by running:

```
$ ./check_stf2x2.sh
```

And the Run-In-Order can be checked by running:

```
$ ./check_rio2x2.sh
```

Where $2 \times 2$ represents the problem size. Other available sizes are $3 \times 2$ and $3 \times 3$, checking the bigger sizes may take dozens of hours.

## B.1 STF specification

MODULE *STFSpec*

EXTENDS *Integers*, *TLC*

An STF program is represented as a Task Flow, that is a set of tasks on which a sequential order is defined. Tasks are represented as functions mapping data items to a kind of dependency. Dependency kinds are read-only ("R"), write ("W") or none ("None"). Finally, a set of workers can execute the tasks concurrently in any order that respects sequential consistency, as defined in this module.

CONSTANTS *Data*,
*Tasks*,
*Workers*,
*TaskFlow*,
*Idle*

VARIABLES *pendingTasks*,
*workerStates*

To make the specification simpler, we define the set of active tasks as the set of tasks being actively processes by all the workers.

$activeWorkers \triangleq \{w \in Workers : workerStates[w] \neq Idle\}$

$activeTasks \triangleq \{workerStates[w] : w \in activeWorkers\}$

Initialy all the workers are idle and no task is being executed

$$\begin{aligned} Init \triangleq\ & \wedge pendingTasks = TaskFlow \\ & \wedge workerStates = [w \in Workers \mapsto Idle] \end{aligned}$$

This predicate can be used to verify the correctness of the model parameters and variables.

$$\begin{aligned} TypeOK \triangleq\ & \wedge \forall\, t \in \text{DOMAIN } Tasks : Tasks[t] \in [Data \to \{\text{“R”}, \text{“W”}, \text{“None”}\}] \\ & \wedge TaskFlow \subseteq (\text{DOMAIN } Tasks) \times Int \\ & \wedge activeTasks \subseteq TaskFlow \\ & \wedge pendingTasks \subseteq TaskFlow \end{aligned}$$

An execution is said to be data-race free if there is no concurrent executions of two tasks such that one task has a write dependency on a data while the other has either a read or write dependency on the same data.

$$\begin{aligned} DataRaceFreedom \triangleq\ & \forall \langle task1, task2 \rangle \in activeTasks \times activeTasks : \\ & \quad \forall\, d \in Data : \\ & \qquad Tasks[task1[1]][d] = \text{“W”} \implies \vee\, task1[2] = task2[2] \\ & \qquad\qquad\qquad\qquad\quad \vee\, Tasks[task2[1]][d] = \text{“None”} \end{aligned}$$

Sequential consistency is ensured by allowing execution of a task only if for each data on which this task has a read or write dependency, all previous tasks (in the sequential order) that have a write dependency on that same data have already been executed.

$$\begin{aligned} ReadReady(d, tid) \triangleq\ & \forall \langle t, otherTid \rangle \in pendingTasks \cup activeTasks : \\ & \quad \vee\, Tasks[t][d] = \text{“None”} \\ & \quad \vee\, Tasks[t][d] = \text{“R”} \\ & \quad \vee\, otherTid \geq tid \end{aligned}$$

$$\begin{aligned} WriteReady(d, tid) \triangleq\ & \forall \langle t, otherTid \rangle \in pendingTasks \cup activeTasks : \\ & \quad \vee\, Tasks[t][d] = \text{“None”} \\ & \quad \vee\, otherTid \geq tid \end{aligned}$$

$$\begin{aligned} TaskReady(t, tid) \triangleq\ & \forall\, d \in Data : \\ & \quad \vee\, Tasks[t][d] = \text{“None”} \\ & \quad \vee\, Tasks[t][d] = \text{“R”} \ \wedge ReadReady(d, tid) \\ & \quad \vee\, Tasks[t][d] = \text{“W”} \wedge WriteReady(d, tid) \end{aligned}$$

A step in an STF execution consists in either an idle worker starting to execute a task that is marked a ready, or a busy worker terminating the execution of a task it started earlier.

$$ExecuteTask(w, t, tid) \triangleq\ \wedge TaskReady(t, tid)$$

$$\begin{aligned} &\wedge\ workerStates[w] = Idle \\ &\wedge\ workerStates' = [workerStates \text{ EXCEPT } ![w] = \langle t,\ tid\rangle] \\ &\wedge\ pendingTasks' = pendingTasks \setminus \{\langle t,\ tid\rangle\} \end{aligned}$$

$$\begin{aligned} TerminateTask(w) \triangleq\ &\wedge\ workerStates[w] \in (\text{DOMAIN } Tasks) \times Int \\ &\wedge\ workerStates' = [workerStates \text{ EXCEPT } ![w] = Idle] \\ &\wedge\ \text{UNCHANGED } pendingTasks \end{aligned}$$

$$\begin{aligned} Next \triangleq\ \exists\, w \in Workers :\ &\vee \exists\, \langle t,\ tid\rangle \in pendingTasks : ExecuteTask(w,\ t,\ tid) \\ &\vee\ TerminateTask(w) \end{aligned}$$

The Next predicate ensures sequential consistency by selecting tasks to be executed among tasks whose dependencies have already been executed. But we are also interested in two other properties: data-race freedom and termination. The following theorem asserts that the STF specification indeed ensures those properties hold.

$$Terminated \triangleq pendingTasks \cup activeTasks = \{\}$$

$$\begin{aligned} Spec \triangleq\ &\wedge\ Init \\ &\wedge\ \Box[Next]_{\langle pendingTasks,\ workerStates\rangle} \\ &\wedge\ \mathrm{WF}_{\langle pendingTasks,\ workerStates\rangle}(Next) \end{aligned}$$

THEOREM $Spec \implies \Box DataRaceFreedom \wedge \Diamond Terminated$

## B.2 Run-In-Order model specification

MODULE *RunInOrder*

EXTENDS *Integers*

The STF specification imposes very few constraints on the execution order beyond sequential consistency, this module defines a more restrictive "in-order" model that adds two constraints: the worker responsible for a task is deterministically choosen by a Mapping function and each worker executes its attributed tasks in the sequential order. By implementing the STF specification, this modules shows that the "in-order" model satisfy the same three properties: sequential consistency, data-race freedom and termination.

CONSTANTS *Data*,
*Tasks*,
*Workers*,
*TaskFlow*,
*Mapping*,
*Idle*

VARIABLES *workerPendingTasks*,
*terminatedTasks*,
*workerStates*

To make following statements simpler, we define the pending and active tasks in terms of the worker task queues and worker states.

$$pendingTasks \triangleq \text{UNION } \{workerPendingTasks[w] : w \in Workers\}$$

$$activeWorkers \triangleq \{w \in Workers : workerStates[w] \neq Idle\}$$

$$activeTasks \triangleq \{workerStates[w] : w \in activeWorkers\}$$

---

Initially all the workers are idle and the tasks are attributed to the workers using the Mapping function.

$$\begin{aligned} Init \triangleq\ & \land\ terminatedTasks = \{\} \\ & \land\ workerStates = [w \in Workers \mapsto Idle] \\ & \land\ workerPendingTasks = \\ & \qquad [w \in Workers \mapsto \{\langle t,\ tid\rangle \in TaskFlow : Mapping[tid] = w\}] \end{aligned}$$

---

This proposition can be used to verify that constants and variables hold sensible values.

$$\begin{aligned} TypeOK \triangleq\ & \land\ TaskFlow \subseteq (\text{DOMAIN } Tasks) \times Int \\ & \land\ \forall\, t \in \text{DOMAIN } Tasks : Tasks[t] \in [Data \to \{\text{“R”},\ \text{“W”},\ \text{“None”}\}] \\ & \land\ pendingTasks \cup activeTasks \cup terminatedTasks = TaskFlow \end{aligned}$$

---

The two main differences in the way the next tasks are choosen compared to the STF specification is that tasks are choosen from the pool of tasks assigned to a given worker and that only the first (in sequential order) of that pool is considered for execution.

$$\begin{aligned} ReadReady(d,\ tid) \triangleq\ & \forall\, \langle t,\ other\_tid\rangle \in TaskFlow : \\ & \lor\ Tasks[t][d] = \text{“None”} \\ & \lor\ Tasks[t][d] = \text{“R”} \\ & \lor\ tid \leq other\_tid \\ & \lor\ \langle t,\ other\_tid\rangle \in terminatedTasks \end{aligned}$$

$$\begin{aligned} WriteReady(d,\ tid) \triangleq\ & \forall\, \langle t,\ other\_tid\rangle \in TaskFlow : \\ & \lor\ Tasks[t][d] = \text{“None”} \\ & \lor\ tid \leq other\_tid \\ & \lor\ \langle t,\ other\_tid\rangle \in terminatedTasks \end{aligned}$$

$$\begin{aligned} TaskReady(t,\ tid) \triangleq\ & \forall\, d \in Data : \\ & \lor\ Tasks[t][d] = \text{“None”} \\ & \lor\ Tasks[t][d] = \text{“R”}\ \land ReadReady(d,\ tid) \\ & \lor\ Tasks[t][d] = \text{“W”} \land WriteReady(d,\ tid) \end{aligned}$$

$$\begin{aligned} ExecuteTask(w) \triangleq\ & workerStates[w] = Idle \land \exists\, \langle t,\ tid\rangle \in workerPendingTasks[w] : \\ & \land\ \forall\, \langle other\_t,\ other\_tid\rangle \in workerPendingTasks[w] : tid \leq other\_tid \end{aligned}$$

$$
\begin{array}{ll}
 & \land\ TaskReady(t,\ tid) \\
 & \land\ workerPendingTasks' = \\
 & \quad [workerPendingTasks \text{ EXCEPT } ![w] = workerPendingTasks[w] \setminus \{\langle t,\ tid\rangle\}] \\
 & \land\ workerStates' = [workerStates \text{ EXCEPT } ![w] = \langle t,\ tid\rangle] \\
 & \land\ \text{UNCHANGED } terminatedTasks
\end{array}
$$

$$
\begin{array}{lll}
TerminateTask(w) & \triangleq & \land\ workerStates[w] \in (\text{DOMAIN } Tasks) \times Int \\
 & & \land\ workerStates' = [workerStates \text{ EXCEPT } ![w] = Idle] \\
 & & \land\ terminatedTasks' = terminatedTasks \cup \{workerStates[w]\} \\
 & & \land\ \text{UNCHANGED } workerPendingTasks
\end{array}
$$

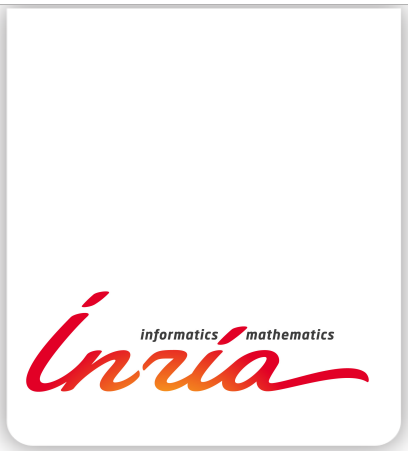
$$Next \triangleq \exists\, w \in Workers : ExecuteTask(w) \lor TerminateTask(w)$$

The following theorem asserts that the "in-order" model implements the STF specification, and thus ensures sequential consistency, data-race freedom and termination.

$$
\begin{array}{lll}
Spec & \triangleq & \land\ Init \\
 & & \land\ \Box[Next]_{\langle workerPendingTasks,\ terminatedTasks,\ workerStates\rangle} \\
 & & \land\ \mathrm{WF}_{\langle workerPendingTasks,\ terminatedTasks,\ workerStates\rangle}(Next)
\end{array}
$$

$$STF \triangleq \text{INSTANCE } STFSpec$$

$$\text{THEOREM } Spec \implies STF\,!\,Spec$$




RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399