Measuring and Interpreting Dependent Task-based Applications Performances

Romain Pereira, Thierry Gautier, Adrien Roussel, Patrick Carribault

▶ **To cite this version:**

Romain Pereira, Thierry Gautier, Adrien Roussel, Patrick Carribault. Measuring and Interpreting Dependent Task-based Applications Performances. 15th International Conference on Parallel Processing & Applied Mathematics, Sep 2024, Ostrava, Czech Republic. hal-04767262

HAL Id: hal-04767262
https://hal.science/hal-04767262v1

Submitted on 5 Nov 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Measuring and Interpreting Dependent Task-based Applications Performances

Romain Pereira[1], Thierry Gautier[1], Adrien Roussel[2], and Patrick Carribault[2]

[1] Avalon, LIP, ENS, Inria, Lyon, France
[2] LIHPC, CEA, DAM, DIF, F-91297, Arpajon, France

Abstract. Breaking down the parallel time into work, idleness, and overheads is crucial for assessing the performance of HPC applications, but difficult to measure in asynchronous dependent tasking runtime systems. No existing tools allow its measurement portably and accurately. This paper introduces POT: a tool-suite for dependent task-based applications performance measurement. We focus on its low-disturbance methodology consisting of task modeling, discrete-event tracing, and post-mortem simulation-based analysis. It supports the OMPT standard OpenMP specifications. The paper evaluates the precision of POT's parallel time breakdown analysis on LLVM and MPC implementations and shows that measurement bias may be neglected above $16\mu s$ workload per task, portably across two architectures and OpenMP runtime systems

Keywords: Performances · Tasks · Time Breakdown · OpenMP

1 Introduction

The desire for an exaflopian machine consuming no more than $20MW$ has led modern supercomputers to adopt massively parallel and heterogeneous architectures [30]. Each processing unit is capable of asynchronous execution: some logic may run on CPUs while computation is running on GPUs, and data are transiting on the network. Mixing programming models to exploit the underlying asynchronous hardware entirely became a necessity. The mixed-use through dependent tasks is a promising solution, allowing fine control of workload synchronizations. Since 2013, the OpenMP standard specifications define a dependent task-based programming model, but its use remains niche with few applications using it: one of the main issue is the lack of tool for comprehensive performance analysis of task-based execution.

An approach for initiating performance profiling consists in breaking down the time into the useful *work* provided by programmers with the *non-work*. Without fine comprehension of the executing environment - including the operating system, compilers, runtimes, and libraries - it can be challenging to explain *non-work*. It is usually separated into *overheads* and *idleness* as shown in Fig. 1. N.R. Tallent et al. [28] defines overheads as the additional

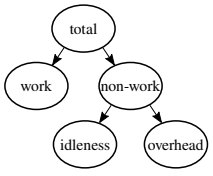


Fig. 1: Breakdown

inherent work of the parallel system for orchestrating tasks over processing units, and idleness as a lack of work to feed each one of them. Breaking down non-work time is fundamental: authors proposed the Table 1 to determine whether an execution is effectively parallel, confronting idleness and overheads to work time.

	Idleness	Overheads	Interpretation
LL	low	low	effectively parallel; focus on serial performance
LH	low	high	coarsen concurrency granularity
HL	high	low	refine concurrency granularity
HH	high	high	switch parallelization strategies

Table 1: N.R. Tallent et al. methodology on parallel execution effectiveness

N.R. Talent et al. (2008) focused on *independent* tasks with Cilk [13]. S.L. Olivier et al. [23] (2013) used such an approach with HPCToolkit [3] to measure work-time inflation induced by Non-Uniform Memory Accesses (data) latencies for independent OpenMP tasks. Breaking down non-work time into idleness and overheads was the starting point for identifying the task dependence graph discovery as a performance-limiting factor [24] (2023). No tools existed for quantifying parallel idleness and overheads of HPC programs using *dependent* tasks, which lead us to implement our own measuring within the Multi-Processing Computing (MPC) OpenMP runtime. One source of difficulties were measurements bias [22] that can lead the analyst to draw wrong conclusions. This paper introduces a low-disturbance measuring methodology for attributing parallel idleness and overheads in dependent task-based programming models. Our contributions are:

- a formal definition of parallel idleness and overhead metrics for dependent task-based programs,
- the modeling of OpenMP 5.2 task specifications to a transition system, to fallback onto formal definitions and measure metrics,
- the introduction of POT: a tool-suite for task-based programming model analysis relying on discrete event tracing and post-mortem simulation to reduce disturbance, with OMPT support for standard OpenMP measurements.

The tool should contribute to improving performance evaluations of applications using dependent tasks. This paper is organized as follows. Section 2 defines parallel overhead and idleness for dependent task-based program and the modeling of OpenMP tasks. Section 3 presents POT and Section 4 evaluates its measurement accuracy. Section 5 review related works before concluding in Section 6.

2 Definitions for Idleness and Overheads

In practice, parallel idleness and overheads metrics are whether not quantified explicitly [2, 21], not considering dependences [16, 23, 26], or measured on a specific runtime system [24]. It causes issue with performance interpretation and comparison between programming models and implementations. This section

formalize metrics definition to any task-based models including dependences with a case of study on OpenMP. Next sections introduces a measurement implementation, discuss its portability, and evaluate its accuracy.

2.1 Formal Definitions

Let $\mathcal{P} = \{1, 2, ..., n\}$ with $n \in \mathbb{N}^*$ a set of processing units, and (V, E) an application represented as directed acyclic graph with V representing tasks as nodes, and E precedence constraints as edges: $(u, v) \in E$ means that task u precedes v, for instance, due to dependency. An *observation* is a triplet (σ, a, d)

- $\sigma : V \mapsto \mathbb{R}^+$ mapping tasks to their starting time,
- $a : V \mapsto \mathcal{P}$ mapping tasks to processors,
- $d : V \mapsto \mathbb{R}$ mapping nodes to their work time,

A *correct* observation verifies the *precedence* (1) and the *processor* (2) constraints:

$$\forall e = (u, v) \in E,\ \sigma(u) + d(u) \leq \sigma(v) \tag{1}$$

$$\forall (u, v) \in V^2,\ a(u) = a(v) \Rightarrow \begin{cases} & \sigma(u) + d(u) \leq \sigma(v) \\ \textbf{or} & \sigma(v) + d(v) \leq \sigma(u) \end{cases} \tag{2}$$

A task $v \in V$ is *ready* at time t if $t < \sigma(v)$ and $\forall e = (u, v) \in E,\ \sigma(u) + d(u) \leq t$. Let $\delta_r : \mathbb{R}^+ \mapsto \{0, 1\}$

$$\delta_r(t) = \begin{cases} 1 & \text{if any task is ready at time } t \\ 0 & \text{otherwise} \end{cases}$$

A processing unit p is *working* at time t if $\exists v \in V$ so that $a(v) = p$ and $\sigma(v) \leq t < \sigma(v) + d(v)$. Let $\delta_w : \mathcal{P} \times \mathbb{R}^+ \mapsto \{0, 1\}$ be

$$\delta_w(p, t) = \begin{cases} 1 & \text{if } p \text{ is working at time } t \\ 0 & \text{otherwise} \end{cases}$$

Note that δ_r is an information global to all processing units while δ_w is local. Given a processing unit $p \in \mathcal{P}$ and a time interval $I = [t_0, t_f]$, we define *overheads* (3) as the non-working time while there are tasks ready. Symmetrically, we define *idleness* (4) as the non-working time with no ready-tasks.

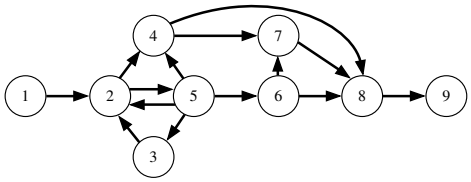
$$overheads(p, I) = \int_{t_0}^{t_f} \delta_r(t)\ (1 - \delta_w(p, t))\ \mathrm{d}t \tag{3}$$

$$idleness(p, I) = \int_{t_0}^{t_f} (1 - \delta_r(t))\ (1 - \delta_w(p, t))\ \mathrm{d}t \tag{4}$$

Note that δ_r, δ_w and their product are piecewise constant functions. Therefore overheads/idleness can be computed as a discrete sum over constant intervals. Hence, measurement difficulties stand in defining δ_r and δ_w for a given execution using a task-based programming model.

Task Transition System

1 - initialized
2 - ready
3 - suspended
4 - cancelled
5 - running
6 - executed
7 - detached
8 - completed
9 - deleted



Transition	Occurs	Related OpenMP callbacks
① → ②	- during a `task` or a `target nowait` construct - or after a predecessor task completed	- `task_create`, `implicit_task` - `dependences`, `task_schedule`
② → ④	- on a cancel taskgroup construct	- `cancel`
② → ⑤	- on a scheduling point	- `task_schedule`
③ → ②	- after a taskwait completion - or after an undeferred child task executed	- `sync_region` - `task_schedule`
④ → ⑦	- the discarded task has a detach clause with an unfulfilled event	- *none*
④ → ⑧	- the discarded task has a detach clause with a fulfilled event, or no detach clause	- *none*
⑤ → ②	- on any task scheduling point	- `task_schedule`
⑤ → ③	- on a `taskwait` construct - or on an undeferred child `task` construct	- `sync_region` - `task_create`
⑤ → ④	- on a task cancellation point	- `cancel`
⑤ → ⑥	- after the task structured block executed	- `task_schedule`
⑥ → ⑦	- after executing a detachable task with an unfulfilled event	- `task_schedule`
⑥ → ⑧	- after executing an undetachable task, or a detachable task with a fulfilled event	- `task_schedule`
⑦ → ⑧	- after a detached task event is fulfilled	- `task_schedule`
⑧ → ⑨	- whenever the task is deleted	- *none*

Fig. 2: Modeling OpenMP Specifications 5.2 Tasks as a Transition System

2.2 Task-based Programming Model: case of study of OpenMP

In order to define δ_r and δ_w for a task based program, we propose to model tasks as a *transition system* [18]. We illustrate on OpenMP. Since the introduction of explicit tasks [4], the OpenMP standard kept evolving towards a general purpose task-based programming model. Our reading of the standard specification and experience as implementers lead us to the modeling depicted Fig. 2.

The table links OMPT callbacks to task transitions. For instance, ① → ② may occur whether (a) on a `task` construct if the task has no dependence or if its predecessors had already completed; or (b) after its last predecessor's completion. Using the transition system, δ_r and δ_w can be trivially defined for OpenMP : Computing δ_r and δ_w constant intervals is therefore deferred to a problem of bookkeeping task states over an execution: this is the purpose of POT.

$$\delta_r(t) = \begin{cases} 1 & \text{if there is any task in state } ② \text{ at time } t \\ 0 & \text{otherwise} \end{cases}$$

$$\delta_w(p,t) = \begin{cases} 1 & \text{if there is any task in state } ⑤ \text{ on } p \text{ at time } t \\ 0 & \text{otherwise} \end{cases}$$

It must be noted that the formal model Section 2.1 allows only a single schedule per task, while *OpenMP tasks* can have multiple schedules via the cycle (2) → (5) → (3). However, falling back to the formal model is straightforward subdividing OpenMP tasks into logically-sequential segments interpreting a task suspension as a task completion and a new continuation task creation as per H. S. Matar [20].

3 POT - A Tool-suite For Dependent Task-based Programming Model Performance Analysis

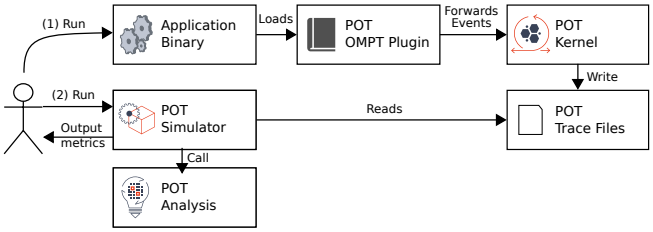


Fig. 3: POT Workflow

The primary motivation of POT tool-suite is the computation of δ_r and δ_w functions with low distortion of the execution; else-way, measurements bias [22] could lead the analyst to draw wrong conclusions. For that purpose, POT workflow consists of a two-step measurement method as shown in Fig. 3. First, the analyst must execute an instrumented version of its program to record and timestamp events when δ_r and δ_w may vary. Then, a post-mortem simulation of the execution replays events to construct δ_r and δ_w tracking task states changes. The following sections present our implementation that is publicly available [3] .

3.1 Instrumentation using OMPT

The POT instrumentation is composed of two modules : (1) the OMPT plugin converts and forwards OpenMP callbacks as events to the POT-Kernel, which is responsible for recording. The OMPT plugin currently includes the 8 events depicted on Fig. 2 out of the 37 available in the standard specifications.

- `thread_{begin|end}` records available CPUs assuming no over-subscription.
- `task_create` and `implicit_task` records task initialization (1).
- `task_schedule` is used for several transitions and needed to track transition to state (5) and compute δ_w.
- `dependences` controls whether a task became ready after a predecessor completion: a key transition (1) → (2) for computing of δ_r
- `sync_region_wait` reports the start and completion of a `taskwait` construct, hence controls task suspension and readiness on the cycle: (3) → (2) → (5): transitions needed to compute δ_r.

[3] `https://gitlab.inria.fr/ropereir/pot`

– `sync_region` and `cancel` can be used to detect a `taskgroup`, its cancellation, and discarded tasks for transitioning (2),(5) → (4).

OpenMP `task_dependence` callback reports dependency edges between tasks generated by the runtime system. However, it may not report all edges as the runtime itself may skip some, for instance, when predecessors are completed before the successor creation. This callback is insufficient to reconstruct the task dependency graph entirely, and using the `dependence` callback is required. To limit distortion on the execution, the OMPT plugin follows two principles:

– (1) Limit events callback to recording only
– (2) Perform only thread-local operations

On (1), POT-Kernel pre-allocates memory buffers to record events at run-time and flushes to disk on program termination. Recording an event includes time-stamping and writing from 8 to 64 bytes of data to the memory buffer, depending on the traced event. Using the `OSJitter`[4] micro-benchmark, we measured $16ns$ overheads for time-stamping with `clock_gettime` on Intel(R) Xeon(R) Gold 6130 CPU. It corresponds to the cost of the function call reading the Time Stamp Counter (TSC). On (2), memory buffers are pre-allocated per-thread, and the tool identifies tasks with a primary key $(thread\text{-}id, task\text{-}id)$ so each thread can generate them independently from one another. Evaluations Section 4.1 quantify distortions on a set of benchmarks.

3.2 Simulating Dependent Tasks

Algorithm 1 Simulator - In-order discrete event simulation

In: $\mathcal{E}$ a set of events
1: Initialize simulator S
2: $E := \{\}$
3: **while** $E \neq \mathcal{E}$ **do**
4: $\quad e := \min\ \mathcal{E} \setminus E$ ▷ with $e_1 < e_2 \Leftrightarrow e_1.time < e_2.time$
5: $\quad d_t := e.time - \max\{x.time \mid x \in E\}$ ▷ time distance to last event
6: $\quad$ Analysis(S, e, d_t)
7: $\quad$ Update(S, e) ▷ Update the simulator accordingly to the event e
8: $\quad E := E \cup \{e\}$
9: **end while**

The second step of POT measurements is a post-mortem simulation of the execution presented on Algorithm 1. The simulator reconstructs task states along the recorded execution so analysis can compute and integrate (δ_r, δ_w). It reads recorded events sequentially and in-order, to maintain coherent tasks states over time through the `Update` routine. It is illustrated on Algorithm 2 on a few OMPT events. For instance, on a `task_create` event, the simulator allocates a new task to state `1`, and may transition it to (2) if its dependences are fulfilled. Users can enable analysis passes alongside the simulation. Our *time-breakdown* analysis is presented in Algorithm 3. It integrates δ_r and δ_w provided by the simulator computing overheads and idleness as a discrete sum.

[4] `https://github.com/gsauthof/osjitter`

Algorithm 2 Simulator - Update

```
In: e an OpenMP event record
switch e do
    case task_create with task
        Transition task to (1)
        if task has no dependences or task' predecessors completed then
            Transition task to (2)
        end if
    case task_schedule with (prior_task, prio_task_status, next_task)
        Transition next_task to (5)
        if prev_status is executed then
            Transition prior_task to (6)
        end if
        ...
    case dependences with (task, dependences)
        Insert in TDG with respect to dependences                ▷ in, out, ...
        if task' predecessors completed then                     ▷ i.e. they are in state (8)
            Transition task to (2)
        end if
    case ...
```

Algorithm 3 Simulator - Analysis - Computing the Parallel Time Breakdown

```
In: e current event; d_t time until last event
Inout: a parallel time breakdown (work, overheads, idleness)
for each Processing unit p ∈ P do
    if there is a task running on p then                         ▷ ⇔ δ_w(p, t) = 1
        work := work + d_t
    else
        if there is any task ready then                          ▷ ⇔ δ_r(t) = 1
            overheads := overheads + dt
        else
            idleness := idleness + d_t
        end if
    end if
end for
```

3.3 Discussion: Portability of POT

This paper primarily focused on bridging POT with OpenMP through OMPT. As shown by the transition system Fig. 2, the OpenMP standard task programming model had become very generic with many task states and transitions. Shared-memory task-based programming models such as Cilk [13], Kaapi [15], Kokkos [12], OmpSs-2 [2] or StarPU [1] could most likely be mapped to the presented transition system. Hence, for a given programming model, a straightforward bridging to POT is through the support of the OMPT event callbacks listed on Fig. 2. Nevertheless, because all the above cited runtimes have their own tracing tools, it could be most straightforward to rewrite the `Update` Algorithm 2 that reads the recorded trace and generates the corresponding transitions.

However, any task-based programming model may not be mapped to the presented transition system. It is the case for distributed task-based programming models [6, 8, 16, 19] where tasks migrate between distributed processes. The POT-Kernel and Simulator would need extensions to support new events and their

interpretation into states and transitions. In addition, time-stamping distributed using different hardware clock references (*i.e.* TSC registers) introduces a new measurement bias. D. Becker et al. mention that "*the analysis of such traces may yield wrong quantitative results and confuse the user*" [5]. Solutions exist, including post-mortem corrections which POT could take advantage of.

4 Evaluation

We conduct several experiments to assess the precision of POT measurements. In our evaluations, we are using the following environment:

- Debian 11 installed via Kadeploy on Grid5000 [9],
- Intel(R) Xeon(R) Gold 6130 16-Core processor,
- Compute nodes with two Intel processors,
- LLVM 19.x compiler and runtime (commit `c40146c`),
- Compactly binding threads to cores.

4.1 Slowdown induced by tracing tools

Our first experiment evaluates the slowdown induced by several tools, all based on OMPTS, compared to the non-instrumented execution. We use two complementary suites of benchmarks: the BOTS (b) [11] for independent tasking, and the KaStORS (k) [29] for dependent tasking. We measure the slowdown over non-instrumented executions of several tools:

- *no-op* is an OMPT tool with empty callbacks for every event,
- TiKKi [10], OMPTrace [5], OmpTracing [25], HPCToolkit [3] are OMPT tools.

Results are presented on Table 2 as the slowdown of the 10-runs average on each configuration. Note that a few measurements did not complete due to too important slowdown (>50) or error during the execution. The penultimate and last line, respectively shows the average slowdown excluding and including them.

The *no-op* results show that enabling OMPT introduces less than 3% of slowdown over non-instrumented execution (`fib.manual` and `uts`). HPCToolkit has the least slowdown on three microbenchmarks (fft, health, sort), which is mostly explained because it only records the `task_create` and `implicit_task` tasking events: the post-mortem analysis lacks information to compute the time breakdown metric in such a case[6]. As shown on the average slowdown, POT-OMPT yields the least slowdown.

4.2 Measurements Precision

The slowdown over the non-instrumented execution metric is not sufficient to determine whether the execution had been distorted. This second experiment compares POT breakdown measurement over theoretical ideal values.

[5] `https://github.com/passlab/omptrace`

[6] Removing the `task_schedule` event recording from POT-OMPT on (fft, health, sort) leads to a (1.05, 1.00, 1.01) slowdown, similar to HPCToolkit.

Benchmark	Parameter	no-op	POT	TiKKi	OMPTrace	OmpTracing	HPCToolkit
(b) alignment	prot.100.aa	1.00	**1.00**	1.00	1.00	1.03	1.01
(b) fft(2^{26})	$n = 2^{26}$	1.01	1.22	1.90	3.18	45.1	**1.04**
(b) fib.manual	$n = 36$	1.02	**1.06**	1.23	1.06	3.37	2.08
(b) health	medium	1.00	1.03	1.09	1.08	7.67	**1.02**
(b) nqueens	$n = 12$	1.00	1.02	1.03	**1.00**	1.17	1.25
(b) sort	$n = 2^{26}$	1.00	1.04	1.31	1.22	11.3	**1.03**
(b) sparselu	$(n, m) = (2^5, 2^8)$	1.00	**1.00**	1.00	1.00	1.01	1.01
(b) strassen	$n = 2^{12}$	1.00	0.99	1.03	1.00	**0.98**	1.00
(b) uts	tiny	1.03	**1.93**	2.64	*err*	*err*	*err*
(k) jacobi	$(n, b) = (2^{15}, 2^8)$	1.00	1.00	1.01	*err*	*err*	1.00
(k) sparselu	$(n, m) = (2^7, 2^6)$	1.00	**0.99**	1.00	1.00	44.3	1.02
(k) strassen	$n = 2^{13}$	1.00	1.00	1.00	**0.99**	1.10	1.01
average of all	w/o uts, jacobi	1.00	**1.04**	1.16	1.25	11.7	1.15
average of all	with uts, jacobi	1.01	**1.11**	1.27	*err*	*err*	*err*

Table 2: Slowdown using OMPT against Non-instrumented Execution

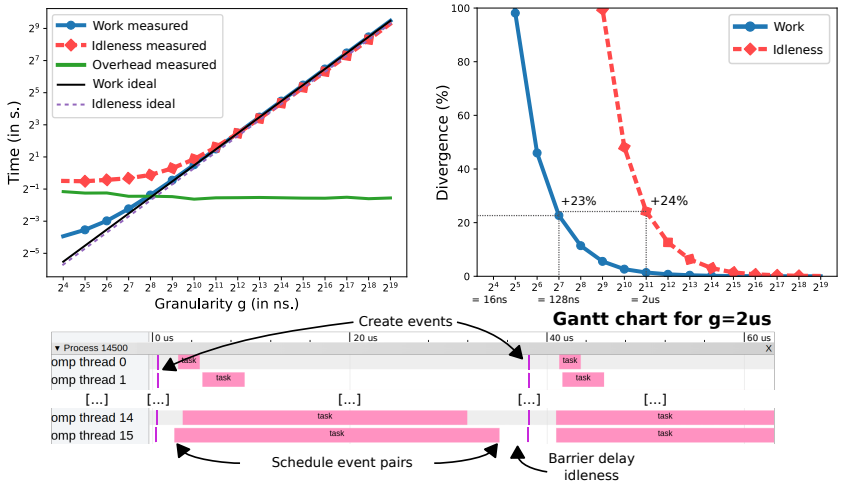


Fig. 4: POT accuracy on LLVM 19.x OpenMP and Intel Cores

Load Imbalance Given a g granularity parameter, each thread $t \in [1, n]$ of the `load-imbalance` micro-benchmark repetitively (I times) creates one task running for $t.g$ μs with $g \in \mathbb{R}^+$ and waits for their completion. On an ideal execution of a single-iteration with no overheads (events recording, task creation, scheduling, threads synchronizations...) we expect :

$$(\text{work},\ \text{idleness},\ \text{overhead}) = \left(g\ \frac{n.(n+1)}{2},\quad g\ \frac{(n-1).n}{2},\quad 0 \right)$$

Results Fig. 4 presents measurements for $I = 10,000$ iterations. X-axes represent the g parameter, which is the finest task grain. The left-side figure y-axis reports measured and ideal time accumulated on threads (in s.) for the entire execution. The right-side figure y-axis reports the divergence (in %), that is, the relative distance between measurement to ideal values for both work and idle times.

Refining g bellow $2\mu s$ for idle time and $128ns$ for work time, both lead to more than 20% inflated measurement reported by POT.

For $g > 16\mu s$, measurements are accurate with less than 3% divergence on work and idleness reported. At finer grain, we find several explanations for the observed divergence inflation. First, each task actively polls its execution time using `clock_gettime` to stop after $g.t$ seconds. Hence, each task may run a few more nanoseconds than ideal (reminder: `clock_gettime` takes $16ns$). A second part stems in the POT-OMPT instrumentation. OpenMP runtime raises event callbacks, which are recorded by POT-OMPT adding a few function calls and from 16 to 64 bytes of memory written. For instance, a `task_schedule` event is recorded after the execution of the task structured block, therefore, the event recording is counted as part of the task *work*, leading to slight work inflation.

On idleness, the ideal value neglects that thread may idle a bit until tasks get created. Using the EPCC microbenchmark [7] `PARALLEL_TASK`, we measured $250 \pm 56ns$ for a MPC-OMP task creation. Additionally, the barrier delay on each iteration introduces idleness time that can be observed on the Gantt chart.

Intel(R) Xeon(R) Gold 6130 16-Core

Runtime	Work Div(<20%)	Idleness Div(<20%)	Overhead average
LLVM	$g \geq 128ns.$	$g \geq 2\mu s$	$0.36s$
MPC-OMP	$g \geq 128ns.$	$g \geq 4\mu s$	$2.64s$

AMD EPYC 7352 24-Core

Runtime	Work Div(<20%)	Idleness Div(<20%)	Overhead average
LLVM	$g \geq 128ns.$	$g \geq 2\mu s$	$1.80s$
MPC-OMP	$g \geq 128ns.$	$g \geq 4\mu s$	$6.69s$

Table 3: Load Imbalance benchmark results portability

Portability We reproduced the experiment on the same machine using the Multi-Processor Computing OpenMP (MPC-OMP) runtime with the LLVM compiler; and on AMD EPYC 7352 24-Core Processor with both LLVM and MPC-OMP runtimes. Table 3 reports results on Intel and AMD CPUs. The first column is the runtime, the second and third columns are grain parameters g for which measurements diverge by more than 20%, and the last column is the average overhead reported for all grains. Both processors have similar grain thresholds for work and idleness divergence. Runtimes have similar results on the minimal grain, although MPC-OMP introduces more overheads than LLVM on average.

4.3 Case of Study : LULESH

Fig. 5 depicts metrics obtained with POT on a dependent task-based version of the Livermore Unstructured Lagrangian Explicit Shock Hydrodynamics (LULESH [7]) proxy-applications. In this experiment, we fixed parameters $(size, iterations)$ to $(80, 128)$ varying the number of threads from 1 to 32 (x-axis) and the number of tasks per loop from 1 to 256 with a step of 8. Heatmap cell values report metrics measured for a given instance of execution. The uppermost figures report the

[7] `https://github.com/rpereira-dev/LULESH`

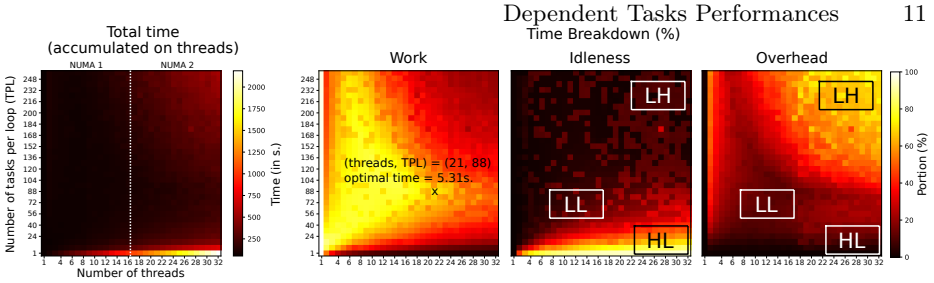


Fig. 5: LULESH time breakdown varying threads and tasks on LLVM 19.x

parallel time accumulated over threads (left) with its time breakdown (right). Note that this experiment exhaustively tests a wide range of threads and task grains. In practice, POT users would compute these metrics on a single point of interest to assess the application performances.

We observe the three LL, HL, and LH configurations respecting Table 1 interpretation. The best performances are reached for 21 threads and 88 tasks per loop in a LL configuration. We observed a similar time-breakdown using MPC-OMP. Refining tasks grain further-more produces HH case-scenario, that had been previously attributed to the task dependency graph discovery on LULESH and optimized with *persistent* tasks to reduce both idleness and overheads [24]. Regarding distortion, we measured an average of 1.3% and 0.8% slowdown with LLVM and MPC-OMP over non-instrumented execution. However, we also observed up to 49% slowdown *and* 42% speed-up in LH configurations where the average task grain varies from 25 to 45μs.

5 Related Works

The HPX [16] runtime system embs counters for performance metrics. In particular, it provides *HPX-Task Overheads* as $\frac{\text{non-work time}}{\text{n° of tasks}}$ and *HPX Thread Idle-rate* as $\frac{\text{non-work time}}{\text{total time}}$. While their name suggests to be related to the time breakdown understudy, these metrics only consider *non-work* time with no distinction on idleness/overheads as per POT (see Fig. 1). It significantly impacts performance interpretation: users would observe high *HPX - Task Overheads* in both LH and HL setup, while they suggest opposite optimizations directions.

M. Roth et al [26] proposed a breakdown of parallel performance factors that can be seen as a refinement of N.R. Tallent [28] breakdown of parallel time. As opposed to POT, their measurement requires a reference sequential execution and instrumentation of each entry point of the parallel runtime. In addition, it may produce negative values when scaling super-linearly; it corresponds to work time deflation between two execution instances with POT.

TAU [27] and HPCToolkit [3] are tool-suites for HPC applications performance analysis. None of them performs a bookkeeping of ready-tasks with respect of their dependences to compute a time breakdown. Extending them with POT-alike simulator could be achieved to take advantage of their existing performance profiling infrastructure.

V. Garcia et al. introduced a performance analysis framework for distributed task-based programs [14] illustrated on StarPU. Their analysis also relies on discrete-event tracing through runtime system instrumentation. However, they do not breakdown non-work time and idleness/overheads interpretation are mainly visual making it subjective to the analyst: POT-like simulation-based metrics could enlight interpretations, in particular with the presence of visual artifacts.

Cilkview [17] is a scalability analyzer for Cilk++ programs via Intel Pin instrumentation. Authors report an average slowdown of 2x to 10x. Integrating Cilkview algorithm in POT simulator as a new analysis would be straightforward, similarly allowing the detection of scalability issues for OpenMP programs.

6 Conclusion

HPC applications are evolving towards more asynchrony with the mixed-use of programming models through dependent tasks. Performance profiler must take into consideration the presence of task dependences.

This paper provides a formal definition of parallel time breakdown with support for dependent tasks. We illustrate on OpenMP with quantification through POT: a tool-suite for dependent task-based applications performance profiling. POT aims at low impact on the execution for accurate and meaningful measurement, by modeling parallel objects as transition systems, discrete-event tracing and post-mortem simulation. While experiments suggests low distortion above $16\mu s$ per tasks on two different CPUs and runtime systems, analyst should always compare their measurement with non-instrumented execution to avoid misinterpreting due to the presented measurement bias. POT-alike analysis had already been used in LULESH detecting HH configuration.

Measuring in presence of GPUs is kept as future work; a straightforward approach could be offering the possibility to choose *processing units* hierarchically on the hardware (GPU, SM, warps, or threads). We would also like to extend POT to mixed-programming model simulation: knowledge of pending MPI requests or Cuda streams alongside the simulation could enlighten time breakdown reports. Hardware-counters could also enlighten reports, for instance to detect limiting performances factor of critical tasks or to attribute energy costs.

Acknowledgements. Parts of this work was financed as part of a Ph.D. at the Commissariat à l'énergie atomique et aux énergies alternatives (CEA). Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] Agullo, E., et al.: Bridging the Gap Between OpenMP and Task-Based Runtime Systems for the Fast Multipole Method. IEEE Transactions on Parallel and Distributed Systems (2017)

[2] Álvarez, D., et al.: Advanced synchronization techniques for task-based runtime systems. In: Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (2021)

[3] Anderson, J., et al.: Preparing for Performance Analysis at Exascale. In: Proceedings of the 36th ACM International Conference on Supercomputing. ICS '22, Association for Computing Machinery, New York, NY, USA (2022)

[4] Ayguade, E., et al.: The Design of OpenMP Tasks. IEEE Transactions on Parallel and Distributed Systems **20**(3), 404–418 (2009)

[5] Becker, D., et al.: Replay-Based Synchronization of Timestamps in Event Traces of Massively Parallel Applications. In: International Conference on Parallel Processing - Workshops (2008)

[6] Bosilca, G., et al.: DAGuE: A Generic Distributed DAG Engine for High Performance Computing. In: IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (2011)

[7] Bull, J.M., et al.: A Microbenchmark Suite for OpenMP Tasks. In: OpenMP in a Heterogeneous World. Springer Berlin Heidelberg (2012)

[8] Callahan, D., et al.: The cascade high productivity language. In: Ninth International Workshop on High-Level Parallel Programming Models and Supportive Environments (2004)

[9] Cappello, F., et al.: Grid'5000: a large scale and highly reconfigurable grid experimental testbed. In: The 6th IEEE/ACM International Workshop on Grid Computing, 2005. (2005)

[10] Daoudi, I., et al.: sOMP: Simulating OpenMP Task-Based Applications with NUMA Effects. In: OpenMP: Portable Multi-Level Parallelism on Modern Systems. Springer International Publishing (2020)

[11] Duran, A., et al.: Barcelona OpenMP Tasks Suite: A Set of Benchmarks Targeting the Exploitation of Task Parallelism in OpenMP. In: 2009 International Conference on Parallel Processing. pp. 124–131 (2009)

[12] Edwards, H.C., Ibanez, D.A.: Kokkos' Task DAG Capabilities. Tech. rep., Sandia National Lab.(SNL-NM), Albuquerque, NM (United States) (2017)

[13] Frigo, M., et al.: The Implementation of the Cilk-5 Multithreaded Language. In: Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation. p. 212–223. PLDI '98, Association for Computing Machinery, New York, NY, USA (1998)

[14] Garcia Pinto, V., , et al.: A Visual Performance Analysis Framework for Task-based Parallel Applications running on Hybrid Clusters. Concurrency and Computation: Practice and Experience (18), 1–31 (2018)

[15] Gautier, T., et al.: XKaapi: A Runtime System for Data-Flow Task Programming on Heterogeneous Architectures. In: 27th IEEE International Parallel &

Distributed Processing Symposium (IPDPS). Boston, Massachusetts, United States (May 2013), https://inria.hal.science/hal-00799904
[16] Grubel, P., et al.: The Performance Implication of Task Size for Applications on the HPX Runtime System. In: IEEE International Conference on Cluster Computing (2015)
[17] He, Y., et al.: The cilkview scalability analyzer. In: Proceedings of the Twenty-Second Annual ACM Symposium on Parallelism in Algorithms and Architectures. SPAA '10, Association for Computing Machinery, New York, NY, USA (2010)
[18] Keller, R.: A fundamental theorem of asynchronous parallel computation. LNCS **24** (1975)
[19] Klinkenberg, J., et al.: Reactive Task Migration for Hybrid MPI+OpenMP Applications. In: Parallel Processing and Applied Mathematics (2020)
[20] Matar, H.S., et al.: Runtime Determinacy Race Detection for OpenMP Tasks. In: Euro-Par: Parallel Processing. Springer International Publishing (2018)
[21] Myllykoski, M.: A Task-Based Algorithm for Reordering the Eigenvalues of a Matrix in Real Schur Form. In: Parallel Processing and Applied Mathematics (2018)
[22] Mytkowicz, T., et al.: Producing wrong data without doing anything obviously wrong! SIGPLAN Not. **44**(3) (mar 2009)
[23] Olivier, S.L., et al.: Characterizing and mitigating work time inflation in task parallel programs. In: SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis
[24] Pereira, R., et al.: Investigating Dependency Graph Discovery Impact on Task-based MPI+OpenMP Applications Performances. In: Proceedings of the 52nd International Conference on Parallel Processing. p. 163–172. ICPP '23, Association for Computing Machinery, New York, NY, USA (2023)
[25] Pinho, V., et al.: OmpTracing: Easy Profiling of OpenMP Programs (09 2020)
[26] Roth, M., et al.: Deconstructing the overhead in parallel applications (2012)
[27] Shende, S.S., Malony, A.D.: The Tau Parallel Performance System. Int. J. High Perform. Comput. Appl. **20**(2), 287–311 (may 2006)
[28] Tallent, N.R., et al.: Effective performance measurement and analysis of multithreaded applications. SIGPLAN Not. **44**(4), 229–240 (feb 2009)
[29] Virouleau, P., et al.: Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite. In: Using and Improving OpenMP for Devices, Tasks, and More. Springer International Publishing (2014)
[30] Whitt, J.L.: Frontier Overview - OLCF Annual User Meeting (October 2022), https://www.olcf.ornl.gov/wp-content/uploads/2022-OLCF-User-Meetig-Overview-of-Frontier-Whitt.pdf