Cycle-Stealing in Load-Imbalanced HPC Applications

Po Hao Chen*‡, Akshaya Bali*‡, Shining Yang*, Pouya Haghi*, Carlton Knox*,
Benjamin Li*, Amr Akmal Abouelmagd†, Anthony Skjellum†, and Martin Herbordt*
*ECE Department, Boston University †Department of Computer Science, Tennessee Technical University

***Abstract*—It is practical to steal cycles when Message Passing Interface (MPI) programs are load imbalanced, either from natural algorithmic load, because of collective communication and/or process skew, and/or because of variable message loads and needs for message progress within the MPI implementation. We introduce TIMELORD, a runtime library to provide fungibility for compute-cycles without significantly degrading the nominal performance of the parallel application. We propose three means to exploit wasted cycles in LAMMPS, miniFE, and miniAMR. The results show, on average, 40% of the runtime can be used to execute *extra* computations and that runtime overheads are less than 4%. We envisage that these extra computations be related to the target application; here, for simplicity and proof-of-concept, we assume arbitrary commodity applications. Additionally, TIMELORD uncovers inefficiencies in the MPI progress engine and improves baseline performance. We explore some fundamental issues of load imbalance and its interaction with system software, including the predictability of extra time and the relative benefits of prediction versus preemption. TIMELORD requires no changes to the MPI application, MPI implementation, or other system components.**

***Index Terms*—Workload Imbalance, High-Performance Computing, MPI, Progress Engine**

I. INTRODUCTION

As parallel computing advances beyond the exascale milestone, increases in heterogeneity and processor counts emphasize the utmost importance of resource utilization. The new flagship system, Aurora, has over 1.1 million CPU cores [1], while another exascale system, Frontier, has over 600,000 cores [2]. This new era of computing implies a new set of challenges in designing scalable applications.

In parallel processing, the decomposition of the problem and assigning tasks to processes has been fundamental to delivering performance, and the set of methods to distribute the algorithmic load is known as load balancing. In most cases, application programmers implement at least some internal mechanism to support the partitioning. In other cases, external software (e.g., [3]) is used. The quality of load balance, however, is ultimately constrained by the inherent limitations of the problem. Another constraint is the programmer effort to maintain load balance as applications increase in complexity over lifetimes of 10+ years, and as they are ported to ever more complex and heterogeneous architectures.

These challenges in load balancing, coupled with variance in the cost incurred for synchronization and communication across many processes, result in a significant portion (§IV-D) of the runtime being spent busy-waiting for the laggard; that is, the slowest, process to complete its assignment [4]–[6]. Moreover, as applications suffer from process skews, performance loss is further exacerbated by external noise sources such as networking, the operating system, and more [7]–[9]. The standard everyday approach to improve resource utilization is time-slicing: idle resources are released to other processes. However, time-slicing is not well-defined for High-Performance Computing (HPC) workloads, since they are compute-intensive and resource-hungry. Also, typically each entire compute node is occupied only with a single job.

Motivated by this problem, prior work [10], [11] has proposed to minimize waiting costs or to slow down the lead process. Another solution is to overload a CPU core with multiple threads [12], [13]. These approaches pertain to reducing energy consumption, whereas our work exploits a fixed cost to run the program.

In this paper, we propose TIMELORD: an untapped, yet straightforward, method to improve efficiency by claiming the wasted cycles (*extra time*) to enable execution of *extra work*. We define *extra work* as a secondary program that can be executed along with the primary MPI application. In practice, extra work can be any user-selected function. This spans from scientific computation, such as Folding@Home [14], to utility functions, video encoding, or homomorphic encryption. We recognize that the most valuable secondary program of all may be work usable by the primary program itself.

We postulate that the standard synchronous communication model has concealed benefits yet to be extracted. The focus of this study is any MPI application with a single communicator. We offer an application-agnostic runtime library that applies a transformation to any blocking MPI collective communication to provide fungibility to the *trapped* CPU cycles. We examine methods of joining existing features in the MPI implementation with OS scheduling, as well as designing an adaptive prediction scheme to estimate the wait time. A key feature of the approach is that it is completely realistic and transparent. TIMELORD only needs to be linked to the application. It is run transparently within current production HPC environments including MPI, Scheduler, and OS.

We summarize our approach as follows. The starting point is the fact that, despite massive efforts, most processes in HPC MPI applications continue to have significant idle (*extra*) time. We assert that doing something useful with this idle time is better than doing nothing. We suggest various forms that this

‡Authors contributed equally

extra work can take, but also claim that the overall result does not depend on any of these particular *extra work* applications being viable. Finally, we arrive at the core of this work: a study of various approaches, collectively referred to as TIMELORD, to exploit this *extra time* through the execution of *extra work*. Among the questions asked are: (i) How predictable is the *extra time*? (ii) What are the new overheads of the different options? and (iii) Which class of approaches are more efficient, those based on prediction or on preemption? A further goal of this work is provide a framework within which to better tune and structure HPC applications.

We evaluate TIMELORD with respect to a set of applications on the TACC Stampede II cluster [15], a modern HPC system representative of widely available systems on different scales worldwide. We summarize our contributions as the following:

- Propose a runtime library to enable cycle-stealing with zero application-level instrumentation and that is completely transparent to the application, MPI implementation, and scheduler scripting, while requiring no changes to system components (e.g., slurm or Linux).
- Demonstrate that execution of *extra work* has minimal performance overheads in LAMMPS, miniFE, and miniAMR. We show, on average, 40% of the runtime can be exploited at less than 4% runtime overheads.
- Design of an Adaptive Deadline Estimation (ADE) algorithm to improve preemption quality for waiting processes.

The remainder of this paper is organized as follows. Section II describes the background and challenges for this work. Section III describes the system design and mechanisms used in cycle stealing across MPI applications. Section IV describes applications of the framework for cycle stealing. Section V describes an evaluation of the effectiveness of the methods offered. Related work in is Section VI, discussion in Section VII, and conclusions and future work in Section VIII.

II. BACKGROUND

This background section first considers MPI collectives and then discusses kernel preemption.

A. MPI Collectives

In the most prevalent model of parallel computation, bulk synchronous parallel (BSP), processes execute in iterations at the end of which they are synchronized at a barrier. For example, in classical Molecular Dynamics, an iteration corresponds to a timestep. *Extra time*, it would seem, should be found in non-laggard processes idling at that barrier waiting for the laggard process. In reality, however, processes *sync up* frequently during iterations, often in conjunction with MPI collective operations (collectives). Thus the *extra time* is distributed over a large number of small quanta.

Collectives are a set of communication primitives involving a group of processes within the communicator group. They are ubiquitous to HPC applications and have often been the subject of architectural support [16]–[18]. As prior work [19]–[21] reveals, certain collectives are often paired together locally.

In this work we therefore focus on these operations. This is effective because collectives are themselves often barriers (e.g., `Barrier`, `Allreduce`), or behave as such. In any case, we find that there is a high likelihood of finding *extra time* during which *extra work* can execute.

Collectives come in blocking and non-blocking versions. As an alternative to blocking (aka completing) collective communication, their non-blocking (asynchronous) counterparts return control to the application upon initiating the primitive. We exploit this characteristic in TIMELORD. Although we cannot access the message buffer before it is received, asynchrony allows programmers to execute instructions without data dependencies and so overlap computation and communication. This technique still requires some expertise from application programmers while assuming there even exists useful work that can be executed. Rather than improving the application's baseline performance, we exploit this window to gain programmability and utility. Figure 1 illustrates a possible solution to this problem, in which the *extra work* is overlapped with asynchronous communication.

B. Kernel Preemption

Resource management in the kernel is implemented as a scheduler that orchestrates CPU ownership and assigns states to processes, such as *running* or *ready*, based on a given policy. The Linux kernel supports several scheduling policies to allot CPU resources. In this work, we explore the `SCHED_IDLE` [22] policy as a candidate for extra work allocation, as it assigns the lowest priority to the process, forcing tasks to execute only during idle time. This mechanism from the operating system, however suitable, does not integrate well with MPI programs.

In general, MPI blocking communication often employs a busy-polling mechanism during synchronization to avoid high latency and prevent the execution of background processes. Moreover, MPI ranks are typically mapped per core to minimize resource-sharing between multiple threads (e.g., oversubscription). Certain MPI implementations, such as Open MPI and Intel MPI, can be configured to disable polling during blocking communication. When this option is set, MPI invokes the system call `sched_yield` to idle the process, a technique often employed in oversubscribed environments to prevent contention among multiple ranks.

Polling is not always beneficial to overall application performance. We benchmark a set of unmodified (vanilla) applications against a version in which the blocking collective communication is replaced with their non-blocking counterparts (which are then synchronized afterward). In Figure 2, we report a percentage relative to its vanilla runtime for the applications when (1) `sched_yield` is enabled, and (2) a non-blocking call is initiated and waits for completion. We observe some improvements: 3% and 7% runtime reductions for asynchronous mode in miniAMR and LAMMPS, respectively.

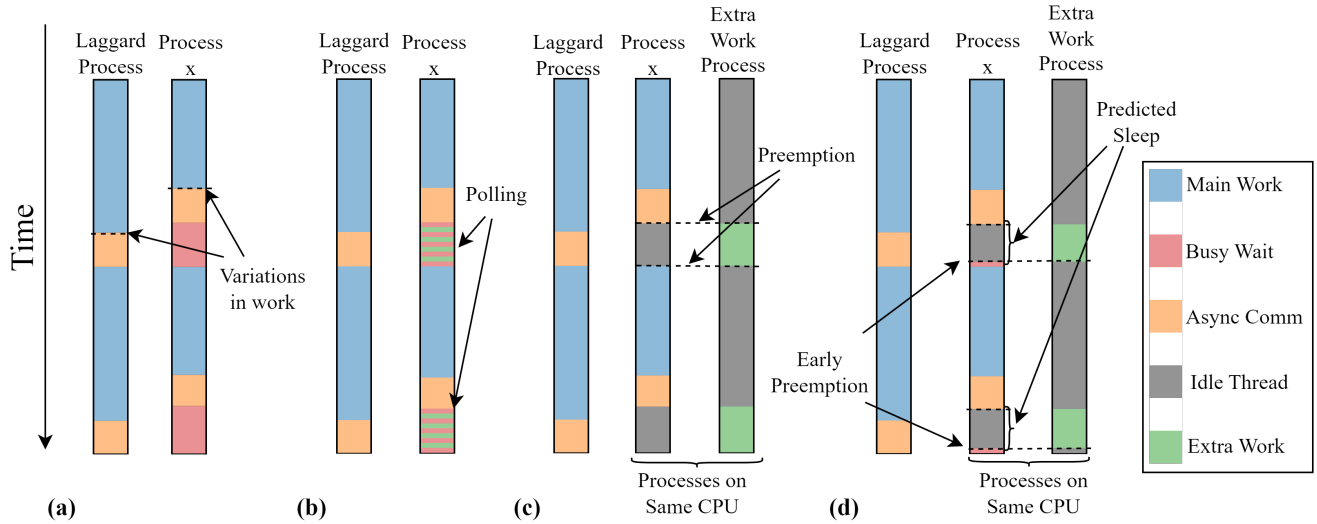

Fig. 1: We define *Comm* as any MPI communication in progress. (a) Problem: longer wait time spent because of variation in the thread. Possible solutions: (b) filling polling time with *extra work* (c) OS time-slicing (d) early preemption of *extra work* to minimize time-slicing overhead.

TABLE I: Summary of TIMELORD methods to schedule *extra work*

Methods	**Use slack**	**Predict / Preempt**	**Blocking**	**Context Switch**
Baseline	–	–	✓	✓
TL-Func (Function/Polling)	all	–	–	–
TL-Proc (OS Preemption)	partial	preempt	✓	✓
TL-Hybrid (Prediction/Preemption)	partial	predict	–	✓

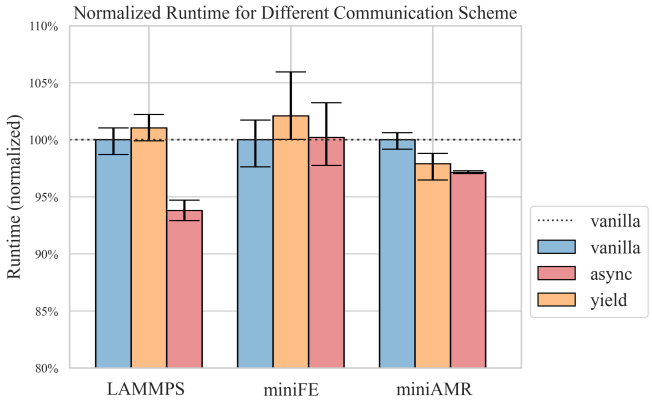

Fig. 2: We normalize the measured runtime for yield (blocking and yield) and async (non-blocking and polling) to its unmodified vanilla (blocking and polling) application performance.

III. FRAMEWORK

TIMELORD is a runtime library that intercepts and manages MPI collective communication procedures. The framework supports the capability to execute a user-defined function or a spawned secondary program during communication or synchronization slack, providing a window for extra computation before the completion of the laggard process. We present three means to enable cycle-stealing: TL-FUNC, TL-PROC, and TL-HYBRID. Table I summarizes their differences.

A. Abstraction

Our prior work [6] depended on careful annotation of the application's Bulk-Synchronous Parallel (BSP) region. Although the user maintains greater control over the *extra work* execution, it can be error-prone and difficult to navigate the complexity of an HPC code base. Consequentially, our design principle focuses on automatic instrumentation: TIMELORD is implemented as a shim that interfaces directly with the collective communication offered by the standard MPI libraries via drop-in replacement and is enabled by using `LD_PRELOAD`. We treat `MPI_*` as a weak symbols that point to the TIMELORD definition, which internally calls its equivalent subroutine with the `PMPI_` prefix. In principle, any program may be considered fungible and executed along with the main application as *extra work*. The user specifies the path to their program by exporting the execution path as an environment variable.

B. TL-Func

This methodology executes *extra work* as a user-level function, which does not require any system calls. Figure 3 illustrates the workflow of performing *extra work* with asynchronous communication and polling. However, this comes with the downside of not being able to save the context; that is, we must explicitly return control at the end of execution. This requires the *extra work* to be a series of iterations with "checkpoints," and limits the possible *extra work* applications.

```
// Polling -- TL-Func

int MPI_Allreduce (...) {
    // In addition to global BARRIER
    // at the end of BSP super cycle,
    // we do this for all collectives
     MPI_Request request;   // Declare request
     MPI_Request status;   // Declare status

     int done;
     int ret = PMPI_Iallreduce (...  &request);
     while not done:
          do_extrawork();
          MPI_Test(&request, &done, &status);
     // EW execution as user-level function,
     // its duration specified by timer.
     return ret;
}
```

Fig. 3: Illustration of TL-Func. While there are no system calls and, therefore, lower overheads, there is also no preemption. We cannot save context; *extra work* must reach a stopping point and explicitly return control at the end of execution.

Therefore, TL-FUNC is suitable for simple tasks with a fixed set of operations.

For this approach, the API exposes two functions for the user to define, `setup_extrawork` and `do_extrawork`, that initializes their *extra work* and performs its execution. As illustrated in Figure 3, the `do_extrawork` function and `PMPI_test` is invoked in a loop until `PMPI_test` detects completion of the collective communication. Consequently, the worst-case overhead for TL-FUNC is the length of `do_extrawork` because of its atomicity.

C. TL-Proc

To exploit the time-slicing feature from the operating system (§II-B), we must disable polling for blocking collectives (§II-A). In fact, this is supported in mainstream MPI distributions. For example, Open MPI accepts `mpi_yield_when_idle` as a launch option, and Intel MPI has `I_MPI_THREAD_YIELD`. Figure 4 illustrates the necessary steps to set up TL-PROC. This method implements a straightforward augmentation to the `MPI_Init` and `MPI_Finalize` procedures. At initialization, we call `MPI_Comm_spawn` to spawn an *extra work* process on each of the ranks. The over-subscription ratio is 2:1, such that the number of spawned processes matches the initial number of MPI ranks (equivalent to the number of launched CPUs in `mpiexec`). The *extra work* must explicitly use the `SCHED_IDLE` policy for scheduling to become a background process. Additionally, it should also handle the `SIGINT` signal to exit gracefully. At program termination, we propagate the `SIGINT` signal to kill the child processes.

In contrast to TL-FUNC, TL-FUNC is capable of context-saving. As a result, it is able to support a more diverse set of complex programs. Moreover, as a side effect, by allowing the OS to handle the extra-work scheduling we also utilize time slices outside the communication layer.

```
    // Yield & Preemption -- OS

    export I_MPI_THREAD_YIELD=2
    // set Intel MPI to yield control to OS.
```

Fig. 4: Alternative to TIMELORD. Intel MPI can yield control to the OS during internal wait when the environmental variable `I_MPI_THREAD_YIELD`=2 is set. *Extra work* is scheduled by the OS and executed until the waiting ends. OS preempts the *extra work*, and returns control of the MPI process. We can resume at any time since the context is saved.

```
// Prediction & Preemption -- Hybrid TimeLord

int MPI_Allreduce (...) {
    MPI_Request request;
    MPI_Status status;
    int id = get_call_id ();
    // described by its return address

    int done;
    int ret = PMPI_Iallreduce (...) ;
    auto x_ns = ADE[id].predict(η, ϵ);

    nanosleep(x_ns);
    // EW execution duration is specified by
    // timer - which is then preempted

    while not done:
        MPI_Test(&request, &done, &status);
    return ret;
}
```

Fig. 5: Illustration of executing *extra work* for a predicted amount of time, as calculated by ADE. The input ψ is a tunable parameter for step size, and ϵ is a threshold to determine if the computation is beneficial. The process releases the resource for `x_ns` nanoseconds. The *extra work* context is saved, and we can stop and start at any time.

D. TL-Hybrid

TL-HYBRID inherits concepts from TL-PROC. As illustrated in Figure 5, we enable asynchronous communication mode in order to estimate the waiting time and preemptively relinquish the cores. The time taken by MPI communication primitives can be divided into two parts: the skew between the initial senders spent waiting, and the duration for messages to be received. This approach minimizes the former, using the system call `nanosleep`, in order to preempt the extra work process before the message is completely received, effectively reducing the preemption overhead towards the application. To achieve this, we employ a prediction scheme, *adaptive deadline estimation* (ADE), that requires only a small number of cycles relative to the potential benefits.

Each collective call at a unique location of the application code is independent of other instances in the same program. Therefore, the skew between processes originates from the inherent communication patterns and external sources of noise. This requires TIMELORD to provide each collective with its own predictor object. To achieve zero application-level

instrumentation, we use the function return address as its unique identifier.

1) Adaptive Deadline Estimation: The prediction scheme is defined in Algorithm 1. ADE monitors the skew duration from the previous iteration while maintaining a *sleep time* variable that adapts to the performance. This variable is initialized to zero nanoseconds. If the skew decreases, as a result of `nanosleep`, we enter the *first increment phase*, in which the sleep time is doubled. Otherwise, we enter the *first decrement phase*, which halves the estimated *sleep time*. In subsequent iterations that trigger either condition, we increment or decrement the value by η nanoseconds . Also, an ϵ value is used as a threshold to filter out any noise in the measurements where $\epsilon << \eta$.

2) Behavior: The predicted *sleep time* pessimistically chases optimality and continues to oscillate under it. The benefits of the estimation are determined by the median of the skew between the current process and the laggard; if the current process is the laggard, there is no skew. The worst case overhead is characterized by the quantity $N(\eta/2) \times \frac{1}{t}$, where N is the total number of iterations the primitive is invoked and t is the program execution time. In general, this is sufficiently small to have minimal impact on the performance, similar to what we expect in TL-PROC. If $\eta < 1/2m$, where m is the median of the skew, then η has been is appropriately chosen.

Algorithm 1 ADE (η: step size, ϵ: threshold)

```
if Wait Time decreases by ε then
    if First Increment Phase then
        sleep_time ← sleep_time × 2
    end
    else
        sleep_time ← sleep_time + η
    end
end
else
    if First Decrement Phase then
        sleep_time ← sleep_time ÷ 2
    end
    else
        sleep_time ← sleep_time − η
    end
end
```

IV. APPLICATIONS

We evaluate TIMELORD against a set of HPC applications on a production supercomputer (Section V). Applicatoins were chosen to represent a range of communication patterns and workloads, and because they have significant collective communication (Section III).

A. LAMMPS–Molecular Dynamics

LAMMPS is a classical MD code [23] with substantial fine-grained communication. Collectives are used to distribute particle dynamics and collect, sum, and apply forces.

B. MiniAMR–Structured Grid

MiniAMR performs a 3D-stencil calculation with Adaptive Mesh Refinement (AMR) [24]. It applies stencil computations on a cubic domain which is divided into blocks. More specifically, MINIAMR uses 3D meshes with an octree structure. Each block communicates ghost values with neighboring blocks. Refining a block involves splitting it into eight cubic sub-blocks. When coarsening a block, the eight adjacent sub-blocks are replaced with the parent block [25]. The major phases are refinement/coarsening, load balancing (repartitioning after refinement/coarsening), and a 7-point stencil.

C. MiniFE–Unstructured Grid

MiniFE is a proxy application for unstructured implicit finite element methods. The finite element generation, assembly, and solution are performed on a 3D-box physical domain [26]. The important MINIFE kernels are element operator computation (diffusion matrix and source vector), assembly (scattering element operators into sparse matrix and vector), sparse matrix-vector multiplication (the main kernel of a conjugate gradient solver) [27], and miscellaneous vector operations including dot-product and norm.

D. Potential Idle Time

To understand the potential benefit, we measured the total time spent in MPI collective communication for the described applications with respect to process count. As shown in Figure 6, load-imbalanced programs have a higher wait time to benefit from the cycle-stealing approach, while load-balanced configurations have significantly fewer wasted cycles. Also, the wait ratio, i.e. the percentage of the runtime spent waiting, gets amplified as communication complexity increases. Here, miniFE has two scenarios, where the $*$-suffix denotes the problem with equal dimensions. Results from the latter shows lower wait-ratio. We observe that application configuration may drastically change the potential slack. For the following experiments, we do not optimize the input settings shown in Table II.

V. EVALUATION

A. Experimental Setup

The experiments presented in this section were conducted on the TACC Stampede II cluster. The system is configured with Intel Xeon Platinum 8160 CPUs, based on the Skylake (SKX) architecture, with 48 cores (2 sockets) per node and 100 GB/s Intel Omni-Path Architecture (OPA) fabric for supporting inter-node communications. We use Intel libraries and compilers (version 18.0.2), e.g., MPI or Math Kernel Libraries (MKL), for all applications. To minimize noise for the scope of this work, we set the number of OpenMP threads to one per MPI process. Our evaluations are done on exclusive hardware resources, using the maximum number (48) of available physical cores in the CPU, and the number of *extra work* processes matches this amount.

TABLE II: Experimental configurations for the applications. The suffix $*$ implies the workload is well-balanced.

Application	# Processes			
	48	192	768	1536
LAMMPS-lj	500000 atoms	2048000 atoms	8788000 atoms	16384000 atoms
miniFE	$512 \times 512 \times 192$	$1024 \times 1024 \times 192$	$2048 \times 2048 \times 192$	$4096 \times 2048 \times 192$
miniFE $*$	$364 \times 364 \times 364$	$577 \times 577 \times 577$	$916 \times 916 \times 916$	$1154 \times 1154 \times 1154$
miniAMR	$6 \times 4 \times 2$	$8 \times 6 \times 4$	$16 \times 8 \times 6$	$16 \times 12 \times 8$

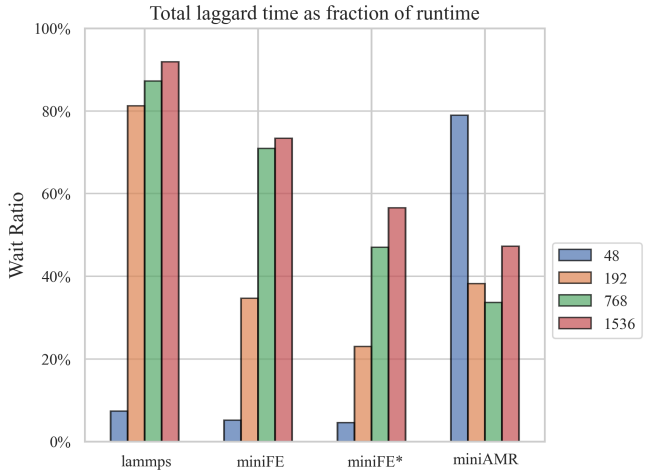

Fig. 6: We measure the elapsed time once asynchronous communication begins to progress until all processes synchronize. We report the program's total percentage (average of three runs) that are spent waiting for laggard completion.

B. Extra work

While different forms of *extra work* demand varying amounts of resources and CPU time, the scope of this paper considers compute-bound workloads that have low memory requirements in order to minimize memory overhead and focus on CPU resource sharing. Cryptographic hash computations showcase high CPU utilization while being less memory intensive. In our *extra work*, we settled on AES encryption, which runs on each MPI rank and repeatedly computes subsequent encryptions on a single 128-bit block cipher. To standardize our approach, we use the popular AES implementation from OpenSSL [28] and kept count of how many times the encryption operation took place. This is then combined for each MPI process that performed the *extra work* to obtain a cumulative total. From this point, we assume any reference to *extra work* as the AES encryption.

C. Overheads & How much extra work?

Communication patterns vary among applications and platforms. Therefore, to compare the *extra work* performance with various applications, a baseline was obtained by executing the *extra work* as a standalone baseline for a fixed time period and counting the number of iterations to determine a standard value of *extra work* iterations per unit time for our machine configuration. We assume that the amount of time taken for each iteration of a CPU-bound *extra work* amortizes over the course of any program's execution. It is then appropriate to normalize the number of *extra work* iterations by this baseline to obtain a percentage of its runtime.

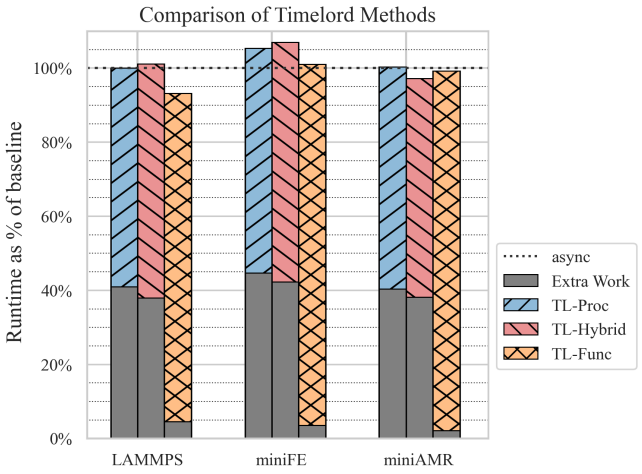

Fig. 7: Comparison of different TIMELORD methods. The experiment is configured with 1,536 processes using the respective input values in Table II. We represent the *extra work* in gray as a percentage of the baseline and show the overhead added to the runtime of each program.

We report the geometric mean of three trials in Figure 7. In terms of overheads, we observe TL-FUNC has a nominal impact on the overall performance because of their user-level invocation. As for TL-PROC (§III-C) and TL-HYBRID (§III-D, both methods rely on system calls; that is, `sched_yield` or `nanosleep` that has a higher context-switching cost to the OS. Both LAMMPS and miniAMR maintain a comparable execution time to the baseline. We observe higher overheads for TL-PROC and TL-HYBRID in memory-bounded applications such as miniFE. This is because of inteference caused by multiple processes sharing cache (e.g, SRAM and TLB). As a consequence, we can infer a positive correlation between their memory intensity and resource sharing.

In terms of utilization, we observe that both TL-PROC and TL-HYBRID are able to perform *extra work* in at least 30% of the application time. On the other hand, TL-FUNC suffers from polling; it is only able to extract a small fraction of the workload imbalance. Although the benefit from TL-FUNC seems little, in the grand scale of HPC workloads, the low-overhead cost for a fraction of a large number of cores may redeem the performance.

VI. Related Work

Goldrush [29] is another system that seeks to harvest idle cycles from HPC programs. Their emphasis is on the large time quanta available during I/O and on a particular form of (what we here call) extra work: data analytics. Our focus is on general methods, on much more fine-grained quanta (through collectives), and on investigating basic mechanisms such as preemption and prediction.

Hoefler and Lumsdaine [30], among others [31], discuss the strategies for asynchronous communication in MPI implementations. Since `MPI_Test` loop methods have a lower chance of being late to receive the message, it has been used in applications to synchronize non-blocking communication. However, in the oversubscribed environment, it forces user threads to share resources with the progression threads; as a result, it can be harmful to overall efficiency. Therefore, preemption is an undervalued technique for the focus of our study. This point was reinforced by the findings of Bierbaum et al. [12], they describe an event-driven model for communication, in which `sched_yield` is used in the progress loop to release resources. Their work shows that using interrupts during non-blocking calls incurs latency costs for a balanced workload; however, it is helpful to load-imbalanced applications and reduces energy consumption.

We are also aware of work, such as [10], [32], that reduces power consumption originating from wasted resources because of skew in communication. That work focuses on minimizing energy spent during the time CPU when is waiting for communication completion; and achieves this by dynamically reducing the CPU frequency and increasing it back to normal soon as messages are received. However suitable, the lower bound of the CPU frequency for stable operation, without incurring high overheads, limits the advantage of voltage scaling. In addition, it can also be affected by hardware variations such as the unavailability of per-core voltage scaling in multi-core CPUs.

VII. Discussion

In Figure 7, we observe TL-Func for LAMMPS and TL-Hybrid for miniAMR have lower runtimes than the baseline. One explanation is that this is the result of inefficient implementation of the MPI progress engine. We show in Figure 6 that non-blocking communication allows TimeLord to claim the hidden available slack time. This means we not only improve over the baseline but also get more utilization. Another possibility is that there is Dynamic Voltage Scaling (DVS) in response to less idle time in the CPU. These observations warrant future studies.

Progress that is purely polling is assuredly not supporting efficient overlap of communication and computation. Comparatively, blocking completion steps ought to do a reasonable amount of polling before descheduling user processes in order to balance arrival rates of messages vs. freeing resources from excessive polling. These types of tradeoffs are well known, but have been hard to exploit in Open MPI, MPICH, and Intel MPI, prior to availability of descheduling services. Other MPI implementations, such as MPI/Pro and ExaMPI [31], [33], [34], have supported strong progress all along, and have shown the ability to overlap communication and computation.

VIII. Conclusion and Future Work

Several areas of future work appear promising. First Co-scheduling: It is possible there is an additional global skew within the *extra work*. We speculate that this can be addressed with multiple levels of *extra work* recursively, where the subsequent skew can be filled by *extra work* from the level below. However, the potential benefit diminishes. Formally, we characterize co-scheduling as a knapsack optimization problem. Given C units of idle time, suppose there are N *extra work* timeslices, and the i-th slice has t_i amount of cost.

$$\arg\max_{S} \sum_{i \in S} t_i \leq C \tag{1}$$

Second, we would like to implement TimeLord in conjunction with the ExaMPI [34] implementation of MPI; this is because of ExaMPI's properties with respect to strong progress.

And third, we are currently rerunning all of the experiments on Stampede 3 with larger node allocations. We are also adding applications (e.g., HPCCG).

To conclude, this work shows that it is practical to steal cycles for HPC workloads. We propose three methods to accomplish this by joining existing features of the kernel and MPI. On average, for three applications, we found 40% of the runtime was used to run *extra work* with only 4% of the overhead cost. This demonstrates that the latency incurred for system calls required for time-slicing (`sched_yield` and `sleep`), have minimal runtime overheads in load-imbalanced HPC applications. In particular, TimeLord effectively unleashes the cycles that have been trapped in the MPI communication layer, and improves resource utilization for modern HPC clusters. Finally, our results highlight potential inefficiencies of MPI progress engines. Nevertheless, we are able to exploit this defect to increase benefit per cost.

Acknowledgments

We thank the anonymous reviewers for their helpful comments. This work was supported, in part, by the NSF through awards CCF 1919130, CCF 2151021, and CCF 2405142; and by a grant from Red Hat.

References

[1] "Aurora — Argonne Leadership Computing Facility — alcf.anl.gov," https://www.alcf.anl.gov/aurora, 2023.

[2] "Frontier — olcf.ornl.gov," https://www.olcf.ornl.gov/frontier/#4, 2023.

[3] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*, 1993, p. 91–108.

[4] P. Haghi, T. Geng, A. Guo, T. Wang, and M. Herbordt, "Reconfigurable Compute-in-the-Network FPGA Assistant for High-Level Collective Support with Distributed Matrix Multiply Case Study," in *IEEE Conference on Field Programmable Technology*, 2020.

[5] P. Haghi, A. Guo, T. Geng, A. Skjellum, and M. Herbordt, "Workload Imbalance in HPC Applications: Effect on Performance of In-Network Processing," in *IEEE High Performance Extreme Computing Conference*, 2021, doi: 10.1109/HPEC49654.2021.9622847.

[6] P. Chen, P. Haghi, J. Chung, T. Geng, R. West, A. Skjellum, and M. Herbordt, "The Viability of Using Online Prediction to Perform Extra Work while Executing BSP Applications," in *IEEE High Performance Extreme Computing Conference*, 2022, doi: 10.1109/HPEC55821.2022.9926405.

[7] K. B. Ferreira, P. Bridges, and R. Brightwell, "Characterizing application sensitivity to os interference using kernel-level noise injection," in *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, 2008, pp. 1–12.

[8] F. Petrini, D. Kerbyson, and S. Pakin, "The case of the missing supercomputer performance: Achieving optimal performance on the 8,192 processors of asci q," in *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, 2003, pp. 55–55.

[9] P. Widener, K. B. Ferreira, S. Levy, and T. Hoefler, "Exploring the effect of noise on the performance benefit of nonblocking Allreduce," in *Proceedings of the 21st European MPI Users' Group Meeting*, ser. EuroMPI/ASIA '14, 2014, p. 77–82.

[10] D. Cesarini, A. Bartolini, P. Bonfà, C. Cavazzoni, and L. Benini, "COUNTDOWN: A Run-Time Library for Performance-Neutral Energy Saving in MPI Applications," *IEEE Transactions on Computers*, vol. 70, no. 5, pp. 682–695, 2021.

[11] N. Kappiah, V. W. Freeh, and D. K. Lowenthal, "Just In Time Dynamic Voltage Scaling: Exploiting Inter-Node Slack to Save Energy in MPI Programs," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005.

[12] J. Bierbaum, M. Planeta, and H. Härtig, "Towards Efficient Oversubscription: On the Cost and Benefit of Event-Based Communication in MPI," in *2022 IEEE/ACM International Workshop on Runtime and Operating Systems for Supercomputers (ROSS)*, 2022, pp. 1–10.

[13] Q. Xiong, E. Ates, M. Herbordt, and A. Coskun, "Tangram: Colocating HPC Applications with Oversubscription," in *IEEE High Performance Extreme Computing Conference*, 2018.

[14] S. M. Larson, C. D. Snow, M. Shirts, and V. S. Pande, "Folding@home and genome@home: Using distributed computing to tackle previously intractable problems in computational biology," 2009, available at https://arxiv.org/abs/0901.0866.

[15] D. Stanzione, B. Barth, N. Gaffney, K. Gaither, C. Hempel, T. Minyard, S. Mehringer, E. Wernert, H. Tufo, D. Panda, and P. Teller, "Stampede 2: The evolution of an xsede supercomputer," in *Proceedings of the Practice and Experience in Advanced Research Computing 2017 on Sustainability, Success and Impact*, 2017.

[16] J. Sheng, Q. Xiong, C. Yang, and M. Herbordt, "Collective Communication on FPGA Clusters with Static Scheduling," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 4, 2017, doi: 10.1145/3039902.3039904.

[17] P. Haghi, T. Geng, A. Guo, T. Wang, and M. Herbordt, "FP-AMG: FPGA-Based Acceleration Framework for Algebraic Multigrid Solvers," in *28th IEEE International Symposium on Field-Programmable Custom Computing Machines*, 2020, dOI: 10.1109/ FCCM48280.2020.00028.

[18] P. Haghi, A. Guo, Q. Xiong, C. Yang, T. Geng, J. Broaddus, R. Marshall, D. Schafer, A. Skjellum, and M. Herbordt, "Reconfigurable switches for high performance and flexible MPI collectives," *Concurrency and Computation: Practice and Experience*, vol. 34, no. 2, 2022, doi: 10.1002/cpe.6769.

[19] P. Haghi, W. Krska, C. Tan, T. Geng, P. Chen, C. Greenwood, A. Guo, T. Hines, C. Wu, A. Li, A. Skjellum, and M. Herbordt, "FLASH: FPGA-Accelerated Smart Switches with GCN Case Study," in *37th ACM International Conference on Supercomputing (ICS)*, 2023, dOI = 10.1145/3577193.3593739.

[20] P. Haghi, R. Marshall, A. Skjellum, and M. Herbordt, "A Survey of Potential MPI Complex Collectives: Large-Scale Mining and Analysis of HPC Applications," 2023, arXiv eprint 2305.19946.

[21] P. Haghi, C. Tan, A. Guo, C. Wu, D. Liu, A. Li, A. Skjellum, T. Geng, and M. Herbordt, "SmartFuse: Reconfigurable Smart Switches to Accelerate Fused Collectives in HPC Applications," in *38th ACM International Conference on Supercomputing (ICS)*, 2024, dOI: 10.1145/3650200.3656616.

[22] "sched(7) - Linux manual page — man7.org," 2023. [Online]. Available: https://man7.org/linux/man-pages/man7/sched.7.html

[23] A. P. Thompson, H. M. Aktulga, R. Berger, D. S. Bolintineanu, W. M. Brown, P. S. Crozier, P. J. in 't Veld, A. Kohlmeyer, S. G. Moore, T. D. Nguyen, R. Shan, M. J. Stevens, J. Tranchida, C. Trott, and S. J. Plimpton, "LAMMPS - a flexible simulation tool for particle-based materials modeling at the atomic, meso, and continuum scales," *Computer Physics Communications*, vol. 271, p. 108171, 2022.

[24] M. Berger and P. Colella, "Local adaptive mesh refinement for shock hydrodynamics," *Journal of Computational Physics*, vol. 82, no. 1, pp. 64–84, 1989.

[25] C. T. Vaughan, "Miniamr adaptive mesh refinement (amr) mini-app," https://github.com/Mantevo/miniAMR, [Online].

[26] P. S. Crozier, H. K. Thornquist, R. W. Numrich, A. B. Williams, H. C. Edwards, E. R. Keiter, M. Rajan, J. M. Willenbring, D. W. Doerfler, and M. A. Heroux, "Improving performance via mini-applications," Sandia National Laboratories, United States, Technical Report SAND2009-5574, 9 2009. [Online]. Available: https://www.osti.gov/biblio/993908

[27] O. Axelsson and P. S. Vassilevski, "A black box generalized conjugate gradient solver with inner iterations and variable-step preconditioning," *SIAM Journal on Matrix Analysis and Applications*, vol. 12, no. 4, pp. 625–644, 1991.

[28] "Openssl project," 2023. [Online]. Available: https://www.openssl.org

[29] F. Zheng, H. Yu, C. Hantas, M. Wolf, G. Eisenhauer, K. Schwan, H. Abbasi, and S. Klasky, "Goldrush: resource efficient in situ scientific data analytics using fine-grained interference aware execution," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, 2013.

[30] T. Hoefler and A. Lumsdaine, "Message progression in parallel computing - to thread or not to thread?" in *2008 IEEE International Conference on Cluster Computing*, 2008, pp. 213–222.

[31] R. Dimitrov and A. Skjellum, "Software architecture and performance comparison of MPI/Pro and MPICH," in *International Conference on Computational Science*. Springer Berlin Heidelberg Berlin, Heidelberg, 2003, pp. 307–315.

[32] B. Rountree, D. K. Lownenthal, B. R. de Supinski, M. Schulz, V. W. Freeh, and T. Bletsch, "Adagio: Making DVS Practical for Complex HPC Applications," in *Proceedings of the 23rd International Conference on Supercomputing*, 2009, p. 460–469.

[33] D. Schafer, T. Hines, E. D. Suggs, M. Rüfenacht, and A. Skjellum, "Overlapping Communication and Computation with ExaMPI's Strong Progress and Modern C++ Design," in *2021 Workshop on Exascale MPI (ExaMPI)*, 2021, pp. 18–26.

[34] A. Skjellum, M. Rüfenacht, N. Sultana, D. Schafer, I. Laguna, and K. Mohror, "ExaMPI: A modern design and implementation to accelerate Message Passing Interface innovation," in *High Performance Computing: 6th Latin American Conference, CARLA 2019, Revised Selected Papers 6*. Springer International Publishing, 2020, pp. 153–169.