The Viability of Using Online Prediction to Perform Extra Work while Executing BSP Applications

Po Hao Chen*§ Pouya Haghi*§ Jae Yoon Chung* Tong Geng‡
Richard West* Anthony Skjellum† Martin C. Herbordt*
*Department of Electrical and Computer Engineering, Boston University
†SimCenter & Department of Computer Science and Engineering, University of Tennessee at Chattanooga
‡Department of Electrical and Computer Engineering, University of Rochester

***Abstract*—A fundamental problem in parallel processing is efficiently partitioning work; often much of a parallel program's execution time is often spent idle or performing overhead operations. We propose to improve the efficiency of system resource utilization by having idle processes execute extra work. We develop a method whereby the execution of extra work is optimized through performance prediction and the setting of limits (a deadline) on the duration of the extra work execution. In our preliminary experiments of proxy BSP applications on a production supercomputer we find that this approach is promising with all five applications benefiting from this approach.**

***Index Terms*—Workload Imbalance, High Performance Computing, BSP, Online Prediction**

I. Introduction

The fundamental problem of parallel processing is determining which process should do what work. The most basic method of partitioning (i.e., decomposition into tasks and assignment of tasks to processes [1]) is the set of methods known as load balancing. Depending on the application, partitioning may be obvious, as in, say, many BSP applications; or it may require some run-time updates (semistatic and dynamic methods). In all cases, however, there are limits to the quality of partitioning with the result that, frequently, much of a parallel program's execution time is spent idle or performing overhead operations [2]–[4]. Workload imbalance can originate from different sources. One type is inherent to the application itself (*i.e.* some processes have more work to do). Another type is due to external noise coming, e.g., from the operating system, network, checkpointing, or mapping [5]–[7].

One possible solution is to spend some of this idle time performing ever more frequent and complex dynamic load balancing; the gain, however, is limited by new overhead for extra work, synchronization, communication, and scheduling. An alternative is to reduce the cost of the idle compute resources—the slack time of the leader processes waiting for laggards—e.g., by slowing down the leader processes [8], [9]. Another alternative is for the leader processes to do extra work while waiting for the laggards. The most general form here is simply time sharing: the leader processes block while waiting and turn over their resources to the operating system. This is generally not useful in HPC, however, due both to the immense amount of OS overhead and the way that processes are scheduled. A different way to execute extra work, and the focus of our study here, is to allocate the excess computing resources to predefined, user-space tasks. For now we leave the *extra work* undefined, but postulate either work that is useful for the current program, or some generic or fungible other work (e.g., SETI at home or bitcoin mining). The value of the extra work is likely to be less than that of the *real work*; the ratio of their values can be a parameter used in optimization.

Determining whether the extra work method (EWM) is viable is the focus of this study and requires several research questions to be answered. First, is there sufficient slack time to make EWM worth the overhead? Second, is it predictable that certain processes will be leaders or laggards? If so, then traditional semistatic load balancing might be preferable. Third, which method should be used to implement EWM? There are at least two possibilities. In *asynchronous* EWM, extra work is performed until the final laggard process completes (reaches the soft barrier) and messages sent to the other processes. In *synchronous* EWM, the execution time of the laggard process is predicted. Then, when any process completes its real work, it checks whether it has exceeded the predicted time and, if not, executes extra work until then. Questions arising in synchronous EWM include: how variable is the execution time (is prediction viable)? How can it be predicted? How accurate are the predictions? What is the cost/benefit of missing a prediction? Should the prediction be guaranteed? How often should the prediction be updated? For simplicity, in this preliminary study, we concentrate on Bulk Synchronous Parallel (BSP) applications, which often consist of a series of similar iterations.

To obtain results with respect to several of these questions we use Stampede 2 cluster [10] with up to 1536 processes (32 nodes) and a set of five BSP proxy applications from Exascale Computing Project (ECP) [11]. We find that two of these applications benefit significantly from EWM and that this benefit is likely to increase with production applications on realistic problem sizes. We summarize our contributions:

- We propose a new method of improving the efficiency of BSP programs in HPC through the use of extra work;
- We demonstrate an optimization method for EWM based on predicting the time interval during which the extra

§The first two authors have equal contribution to the paper.

work should be executed;
- We evaluate the workload imbalance of time intervals for BSP supersteps on Stampede2 compute cluster in different HPC applications;
- We present experimental results showing predictability, the characteristics of the extra work deadline, and the overall benefit of the approach for a preliminary set of BSP proxy HPC applications.

II. Application Space

A. The Bulk Synchronous Parallel Model

The Bulk Synchronous Parallel (BSP) model [12] was developed as a theoretical augmentation of the PRAM model, but is now commonly used to refer to parallel programs that execute in well-defined iterations (supersteps), each terminated by a global barrier. In each superstep, processes perform local computation, communication, and, finally, synchronization. A large fraction of HPC applications fall generally into this model. For instance, in stencil computations, processes communicate with neighbors and perform computation; this happens for a number of iterations [13]. Iterative solvers operate by having each process perform part of the computation; processes are synchronized and convergence checked at the end of each superstep [14].

B. Applications

Our study adopted five selected applications from the ECP Proxy Applications Suite [11]. The *proxies* are designed to be lightweight and simplified while encapsulating the essence of large applications. The simplicity offers the ideal subjects for our evaluation. We described the following applications that implement BSP algorithms.

MiniVite is an iterative graph proxy implementation based on MPI+OpenMP. It performs the first phase of Louvain's method, a community detection algorithm [15]. In real-world large networks, the vertices exhibit densely connected clusters (communities) with considerably more edges than the connections outside. The notion of *modularity* serves as a metric to reflect the density of the edges within the community relative to the ones outside, and it is optimized as the iterations progress. The algorithm converges in a non-deterministic number of iterations.

MiniFE is a proxy application that approximates unstructured implicit finite element by solving a sparse linear system of equations [16]. The kernels exhibit patterns that are similar to many HPC applications. Specifically, the element-operator computations (diffusion matrix and vector); communications via scattering to sparse matrix and vector; sparse linear algebra operations (conjugate gradient solver); and scalar-vector operations.

HPCCG is a conjugate gradient solver for a 3D chimney domain [17]. Similar to MiniFE, the main kernels involve generating finite difference matrix, sparse linear algebra operation (matrix-vector multiplication) and scalar-vector operations. In addition, the implementation allows for configurable sub-block size for each processor.

CoMD is a proxy application for molecular dynamics simulations. The performance depends on the efficiency of evaluating of all forces between atom pairs within some short distance [18]. The program can be broken down into three main stages: inter-node computation, each node computes forces and evolving positions of the atoms within its domain; intra-node computation, assignment of atoms to link cells and checking for interactions; inter-node communication, exchanging information of kinetic and potential energy.

LULESH is a proxy application that approximates hydrodynamics equations by partitioning the Sedov blasts problem into a collection of volumetric elements simulated by a mesh [19]. In addition to following BSP model, the program allows for control over load balance in each set of elements.

III. The Synchronous Extra Work Method

A. Overview of the EW method

The purpose of EWM is to increase the utilization of compute resources during execution of HPC applications. The basic idea is that, rather than idling, waiting processes perform extra work (EW). In this study for exploring the viability of the EWM we make certain assumptions.

Assumption 1: There is a global distributed clock sufficiently accurate so that the skew is small in comparison to the time scales critical to EWM, say, on the order of microseconds. Although current leadership class systems do not have such a capability [20], [21], there are no technological hurdles to their introduction (see, e.g., [22]).

Assumption 2: EW exists that is useful and whose execution is predictable, i.e., it can be *tokenized*. Ideally the EW is useful to the running application, but we do not want to restrict EW to just that. There are many fungible applications whose executions have non-zero value.

Assumption 3: EW is not as useful as the application work. Further, the user (or system administrator) can specify the utility of the EW as some fraction of the utility of the application work.

Assumption 4: There does *not* exist an efficient mechanism to preempt a process. While these mechanisms can be implemented, they are not a standard feature of current large scale systems. We therefore concentrate on *synchronous* EWM and leave *asynchronous* EWM to a further study.

Assumption 5: The cost of starting up and tearing down EW is small. Since we are constraining EW this is reasonable for at least some types of EW, but will be investigated further in another study.

We now sketch EWM for a generic BSP application.

$*$ There is a start-up phase of some number of iterations during which data is collected so that a prediction can be made as to the iteration time and also to determine the distribution, over the processes executing the application, of the amount of application work (and idling) performed by the processes.

$*$ This prediction is used to create a deadline, which is used to optimize EWM. Note that this deadline is unrelated to the use of this term in various types of real time systems. Also, that the prediction is completely empirical (*a posteriori*) and not related to any "hard deadline" of the type found in real-time operating system (RTOS) and often determined *a priori*. In particular, violation is only an inconvenience and not comparable to a violation in a hard RTOS.

$*$ Using this deadline we have the following pseudocode for each process in EWM.

Algorithm 1: Extra Work Method (EWM)

```
1 do
2     DO ApplicationWork        // for this iteration
3     if $T < T_{DL}$ then
4         // beat deadline?
5         do
6             ExtraWork
7         while $T$ != $T_{DL}$;
8     BARRIER()
9 while Termination Condition = False;
```

B. Definitions

Before continuing with details of EWM we summarize the definitions in use.

Iteration := a.k.a. superstep. This is the fundamental unit of BSP and the unit of prediction in this study.

AW and *EW* are the application and extra work, respectively.

T_{global} := or simply T, is the current time on the (postulated) accurate distributed clock.

T_{lag} := is the time it takes the last (*laggard*) process to finish its work during an iteration and defines the iteration time in non-EWM.

T_{AW} := for each process, the amount of time during an iteration spent working on the application.

$T_{slack} := T_{lag} - T_{AW}$. Without EW, this is the amount of *slack* time the process spends idling while waiting for the laggard process.

T_{DL} := A timestamp (roughly, a soft deadline) from the beginning of the iteration. It is used to optimize EWM. Like T, this is a global value.

$T_{EW} := max(0, T_{DL} - T_{AW})$. EW is executed in the interval between the time when the application work is completed and the deadline. If the process misses the deadline, then there is no EW.

T_{idle} := Without EW, this is T_{slack}. With EW, there can also be idle time, but only if the laggard process misses the deadline ($T_{lag} > T_{DL}$). In that case, for processes that meet the deadline ($T_{AW} < T_{DL}$), $T_{idle} = T_{lag} - T_{DL}$. For non-laggard processes that *also* miss the deadline, $T_{idle} = T_{lag} - T_{AW}$.

$T_{iter} := max(T_{lag}, T_{DL})$ is the execution time for a particular iteration (superstep). See Figure 1: Since in EWM T_{DL} can be longer than T_{lag}, T_{iter} is the greater of the two.

U := The coefficient of utility of the EW. This is the assigned fraction of utility of EW with respect to AW.

C. Scenarios

We assume a timestamp mechanism based on the predicted time per iteration (T_{lag}). The timestamp, or *deadline* (as we call it here), can be either *safe* or not. By safe we mean that all processes are guaranteed (with high probability) to complete by the deadline T_{DL}. As we see in the next section, there is substantial variation in iteration time T_{iter} which would make the safe very inefficient. In this study we assume that T_{DL} is set to maximize overall performance. In this version, the laggard process, which sets the non-EWM iteration time, can finish its AW either before the deadline ($T_{lag} < T_{DL}$) or after ($T_{lag} > T_{DL}$); the implications of both are discussed in Section IV.

For non-laggard process execution there are three scenarios as shown in Figure 1; execution of the laggard process follows immediately. In (a), $T_{lag} < T_{DL}$. All processes finish their AW before the deadline and execute EW until the deadline. In (b) and (c), $T_{lag} > T_{DL}$. In (b), the process finishes its AW before the deadline, executes EW until the deadline, but then is idle until T_{lag}. In (c), the process executes AW beyond the deadline and then is (again) idle until T_{lag}.

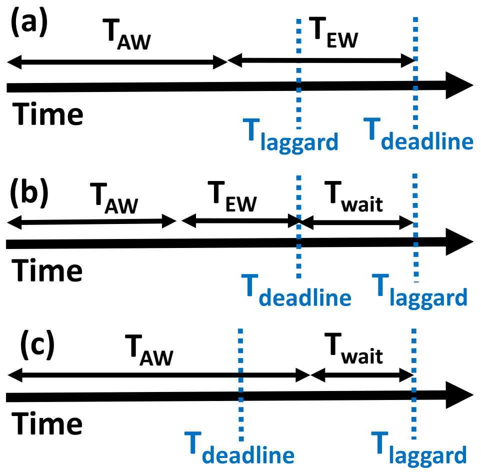

Fig. 1: The three cases for non-laggard process execution.

IV. Evaluation

A. Experimental Setup

For these preliminary experimental results, we executed the applications described in Section II on the Stampede 2 cluster [10]. We picked five BSP-based proxy applications inspired from the work [23] with the configuration shown in Table I. We used up to 32 nodes each with 48 processes (1536 processes in total) but we used fewer processes for some of

TABLE I: Experimental Configuration for Proxy Applications

Application	Small Input	Medium Input	Large Input	# Processes
miniVite	**-p 3 -l -n 131072**	-p 3 -l -n 262144	-p 3 -l -n 524288	128
miniFE	-nx 20 -ny 20 -nz 20	-nx 40 -ny 40 -nz 40	**-nx 60 -ny 60 -nz 60**	1536
HPCCG	64 64 64	**100 100 100**	200 200 200	1536
CoMD	-e -i 16 -j 8 -k 8 -x 128 -y 128 -z 128	-e -i 16 -j 8 -k 8 -x 256 -y 256 -z 256	**-e -i 16 -j 8 -k 8 -x 512 -y 512 -z 512**	1024
LULESH	-s 30 -p -i 500	-s 40 -p -i 500	**-s 50 -p -i 500**	512

the applications due to their requirements. Figure 2 shows an example (LULESH) of our measurement. The figure shows T_{AW} for laggard and leader processes in addition to their deviation across the iterations. As evident from the figure, the deviation is comparable to T_{AW} itself. Due to limited space, we show a sample of results selected from configurations per application (bolded in Table I), which covers small, medium, and large inputs. This sample is largest possible inputs for which runs completed, which is also the most commonly run in HPC environments.

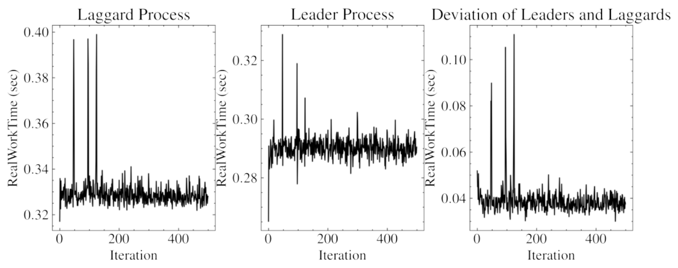

Fig. 2: time measurement (application work) of LULESH per iteration for laggard and leader processes and the deviation.

B. *Are Processes Systematically Laggard?*

If we are able to identify processes as being *systematically laggard*, then the issue may be originated from the underlying hardware (e.g., slow clock) or mapping (e.g., remote node). In that case, non-EWM methods are preferred.

Figure 3 is a series of histograms, one for each application. The bins are the processes sorted by frequency of being laggard. The counts are normalized with the expected frequency of laggardness (#-of-iterations/#-of-processes) being set to 1.

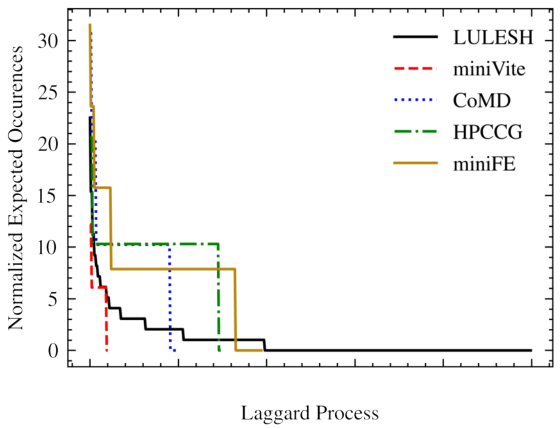

Fig. 3: Laggard process occurrences sorted by the frequency.

If the laggard processes were chosen by chance we would expect a Poisson distribution and a maximum value of roughly the log of the # of processes. This is what we observe in Figure 3 indicating that—for these runs—the selection of the process that is laggard is indistinguishable from random selection.

C. *What is the potential benefit of EWM?*

We find here whether EWM is sufficiently promising to warrant further investigation. The maximum potential benefit of EWM is shown by the behavior of applications under normal execution (without executing EW), i.e., through the proportion of slack time with respect to total time (T_{slack} / T_{iter}) for all processes over all iterations. Table II shows, for each application, the proportion of idle time. Also shown is the average wall-clock idle time per process per iteration.

TABLE II: Fraction of execution time that a process is idle and average idle time per iteration in miliseconds.

Application	Avg Idle Ratio	Avg Idle Time
miniVite	0.25%	0.03
HPCCG	16.93%	526.72
CoMD	0.01%	0.06
LULESH	5.92%	19.58
miniFE	0.17%	1.4

We observe that the potential benefit is there is a bimodal distribution with two applications, LULESH and HPCCG, indicating substantial potential benefit and the others little. We observe further that overhead time, e.g., to initiate EW, is likely to be small in comparison to the average idle time per process per iteration especially for LULESH and HPCCG.

D. *Is optimizing T_{DL} beneficial?*

We begin this subsection by continuing our examination of the various EWM scenarios (e.g., Figure 1) with the goal of finding whether it is beneficial to predict a deadline with the potential to be violated. If so, then the further question is how T_{DL} should be optimized.

We note that EW is not as valuable as AW. Without knowing the details of EW we postulate that, for any run, it is worth some fixed fraction of AW which we denote as U for *coefficient of utility*. We also note that idle/waiting time has $U = 0$. We therefore create as our metric *Average Work Rate* AWR:

$$AWR = \frac{T_{aw} + U * T_{ew}}{T_{iter}} \quad (1)$$

The iteration time is divided into three parts: AW, EW, and idling. AW is weighted at 1, EW at U, and idling at 0. Returning to Figure 1 we see that there are a number of cases.

Baseline: In the baseline case, with no EW, AWR is simply T_{aw}/T_{iter}

EWM: In this case, there are three possibilities:

a): In the case where $T_{lag} < T_{DL}$, all of the idle time has been replaced EW. We observe that the interval T_{EW} during which EW executes has two parts, before and after T_{lag}. The EW during the first interval is entirely positive ($U > 0$), while the *excess* T_{EW}, which continues execution after the normal termination of the iteration, is negative ($U < 1$).

b): Where $T_{DL} < T_{lag}$, in the first case the particular process itself has "beat" the deadline, although some other processes have not. In this case its EW is purely beneficial in that it replaces idling ($U > 0$), but its idling is not because it replaces EW ($0 < U$).

c): Where $T_{DL} \leq T_{lag}$, in the second case the particular process itself has *not* beat the deadline. For these processes there is no EW and the AWR is the same as that of the baseline.

To summarize, setting the deadline either too safely or too aggressively both have their detriments. If the deadline is too safe, then T_{iter} is extended unnecessarily and some AW is replaced with EW. If the deadline is too aggressive, then some EW is unnecessarily replaced with idling.

We predict an optimal deadline based on some number of iterations during which T_{AW} (for each process) and T_{lag} (for the entire iteration) are recorded (see Figures 2 and 5). Since the T_{lag} probability distribution function (Figure 5) may not lend themselves to a simple expression we estimate the optimal T_{DL} as the following.

Algorithm 2: Parameter Search to optimize T_{DL}

```
   Input: U ← Utility Coefficient
1  MaxIterTime = max($T_{iter}$)
2  for $T \in [0, MaxIterTime]$ do
3      $T_{EW}$ = max$(0, T - T_{AW})$
4      foreach iteration i do
5          foreach process p do
6              WorkRate$_{i,p}$ = $(T_{AW} + U * T_{EW})/T_{iter}$
7          end
8      end
9      AWR$_T$ = AVG$(WorkRate)$
10 end
11 return $T_{DL}$ = $\arg\max_T$ AWR
```

Note that, despite the several cases, the computation reduces to simply maximizing AWR, which is itself a simple calculation. Figure 4 shows the optimal T_{DL} for various U from 0.1 to 1 in tenth increments. We note the variety of ranges and shapes of the curves: these are being investigated further.

E. The benefit of EWM

In this section we examine the benefit of synchronous EWM with optimal deadlines under the current workloads. To do so we find the AWRs for selected Us (0.5 and 0.9) and compare with the AWRs of the baseline (average of T_{aw}/T_{iter} for all processes and all iterations with no deadline or EW).

TABLE III: Average work rates AWR for baseline and EWM for two extra work utility coefficients U

Application	AWR		
	baseline	$U = 0.5$	$U = 0.9$
miniVite	99.7%	99.7%	99.7%
miniFE	99.8%	99.8%	99.9%
HPCCG	83%	90%	97.71%
CoMD	99.9%	99.9%	99.9%
LULESH	94%	96.59%	99.9%

F. Discussion

With the current preliminary results we have found that, for two applications, EWM gives sufficient benefit to make this an approach worth examining further. Part of the promise is that the slack time we observed is much less than indicated by the results in [2]. In that study, however, there were larger problem sizes, which led to higher imbalance (1024*512*192 versus 60*60*60), and also a higher number of processes. Moreover, in [2] OpenMP parallelism was used in addition to MPI. So, for example, for miniFE there were 128 nodes with 24 MPI processes per node and 48 OpenMP threads for each MPI process. All of these are more realistic configurations for HPC.

A further question is how often T_{DL} should be recomputed. The calculation is simple, which indicates frequent computation; however, the limited range of optimal T_{DL} (in Figure 4) indicates that this may not be necessary. This must be investigated further.

V. Related Work

We first emphasize that, despite use of deadlines and prediction, that EWM is very different from most real time methods. For example, the prediction is *a posteriori* and the deadlines purely for optimization. Over EWM is more resembles autotuning than RTOS.

A different approach addresses idle time by slowing down leader processes to reduce power and energy consumption. For instance, [8] presents a new power management mechanism based on a timeout algorithm which makes it possible to achieve performance-neutral power saving in MPI applications without requiring application source code modifications. The work [9] presents a system to exploit slack time spent by nodes at synchronization points by reducing the energy gear on those nodes.

Application-aware optimizations are also possible as demonstrated by, e.g, Zhao et al. [24], who optimize performance with online decision. This determines the best time to impose the synchronization barriers for the training phase of deep learning applications. The authors in [25] present a BSP-based aggressive synchronization model for iterative machine learning jobs. More specifically, in their model, the authors terminate the iteration job corresponding to the slowest task as soon as the fastest tasks have completed their jobs. The remaining data of the terminated job of the slowest worker is prioritized in the next iteration.

VI. Conclusion

In this paper we present a new method, called extra work method (EWM), to exploit the time wasted by processes idling

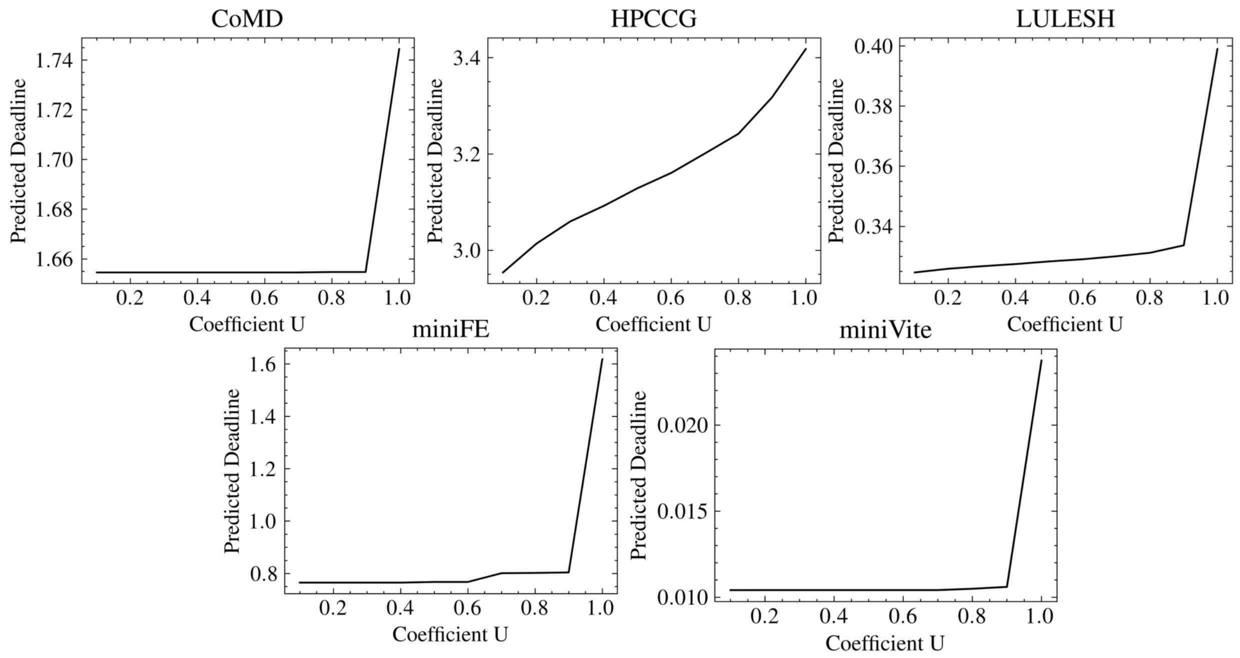

Fig. 4: Deadline Prediction with different Coefficient U

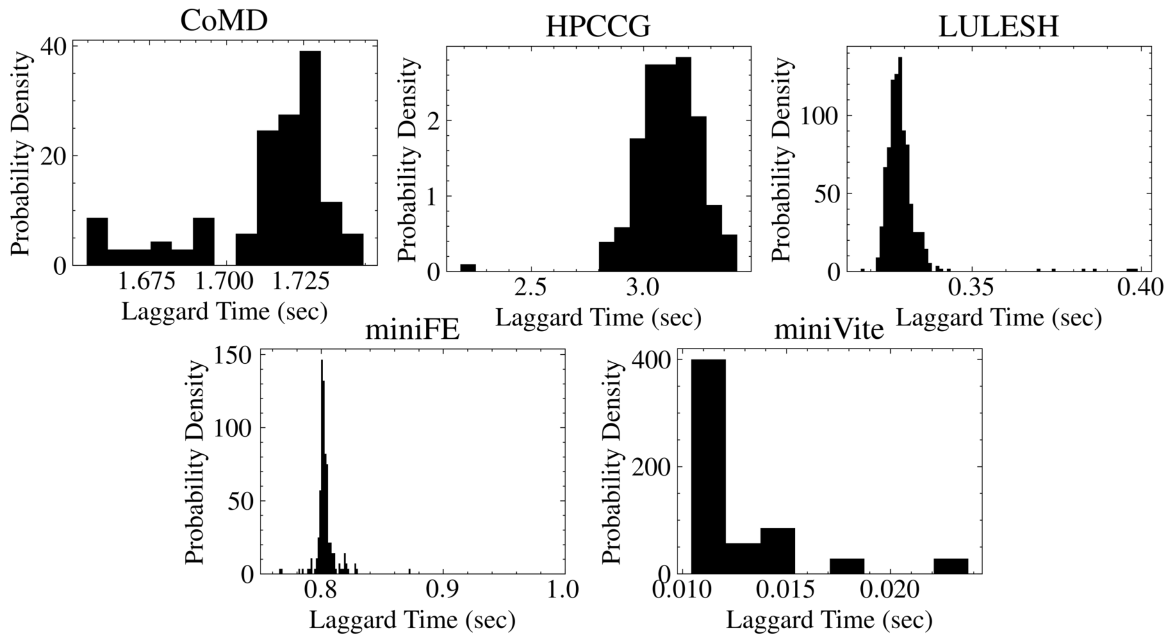

Fig. 5: Probability Density Function (PDF) of T_{lag} for different applications

while running bulk synchronous parallel (BSP) applications. This idling is a result of many factors including workload imbalance and various forms of system noise. EWM is based on, first, using the idle time for extra work, and, second, by optimizing the method by predicting a deadline for supersteps of BSP. In this preliminary study we run proxy applications on a production supercomputer and find that there is benefit for all five of the applications with three applications benefiting by roughly 10% or more. Since these proxy applications are drastically simplified versions of their production versions, and since we were only (so far) able to run small scale samples, we believe there is a good possibility that EWM will have real-world applicability.

There is much work in progress, beginning with running larger problem sizes, then using openMP parallelism in addition to MPI, also using more nodes, finally using other applications, including miniAMR and production codes, e.g., OpenMM instead of CoMD.

ACKNOWLEDGEMENTS

This work was supported, in part, by the NSF through award CCF-1919130; and by the NIH through award R44GM128533.

References

[1] D. Culler, J. Singh, and A. Gupta, *Parallel Computer Architecture: A Hardware/Software Approach*. San Francisco, CA: Morgan-Kaufmann, 1999.

[2] 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.

[3] P. Haghi, A. Guo, Q. Xiong, R. Patel, C. Yang, T. Geng, J. Broaddus, R. Marshall, A. Skjellum, and M. Herbordt, "FPGAs in the Network and Novel Communicator Support Accelerate MPI Collectives," in *IEEE High Performance Extreme Computing Conference*, 2020.

[4] 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.

[5] 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.

[6] 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.

[7] 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. New York, NY, USA: Association for Computing Machinery, 2014, p. 77–82. [Online]. Available: https://doi.org/10.1145/2642769.2642786

[8] D. Cesarini, A. Bartolini, A. Borghesi, C. Cavazzoni, M. Luisier, and L. Benini, "Countdown slack: A run-time library to reduce energy footprint in large-scale mpi applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 11, pp. 2696–2709, 2020.

[9] N. Kappiah, V. Freeh, and D. Lowenthal, "Just in time dynamic voltage scaling: Exploiting inter-node slack to save energy in mpi programs," in *SC '05: Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005, pp. 33–33.

[10] 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 *Practice and Experience in Advanced Research Computing on Sustainability, Success and Impact*, 2017.

[11] "ECP Proxy Applications," https://proxyapps.exascaleproject.org/ecp-proxy-apps-suite/.

[12] T. Cheatham, A. Fahmy, D. Stefanescu, and L. Valiant, *Bulk Synchronous Parallel Computing — A Paradigm for Transportable Software*. Boston, MA: Springer US, 1996, pp. 61–76. [Online]. Available: https://doi.org/10.1007/978-1-4615-4123-3_4

[13] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '07. New York, NY, USA: Association for Computing Machinery, 2007, p. 235–244. [Online]. Available: https://doi.org/10.1145/1250734.1250761

[14] 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.

[15] S. Ghosh, M. Halappanavar, A. Tumeo, A. Kalyanaraman, H. Lu, D. Chavarrià-Miranda, A. Khan, and A. Gebremedhin, "Distributed louvain algorithm for graph community detection," in *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2018, pp. 885–895.

[16] "ECP Proxy Applications, miniFE Catalog," https://github.com/Mantevo/miniFE.

[17] "HPCCG Benchmark," https://github.com/Mantevo/HPCCG.

[18] "CoMD proxy application," http://www.exmatex.org/comd.html.

[19] "Livermore unstructured lagrangian explicit shock hydrodynamics," https://asc.llnl.gov/codes/proxy-apps/lulesh.

[20] T. Jones, G. Ostrouchov, G. A. Koenig, O. H. Mondragon, and P. G. Bridges, "An evaluation of the state of time synchronization on leadership class supercomputers," *Concurrency and Computation: Practice and Experience*, vol. 30, no. 4, p. e4341, 2018, e4341 cpe.4341. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.4341

[21] C. B. Stunkel, R. L. Graham, G. Shainer, M. Kagan, S. S. Sharkawi, B. Rosenburg, and G. A. Chochia, "The high-speed networks of the summit and sierra supercomputers," *IBM Journal of Research and Development*, vol. 64, no. 3/4, pp. 3:1–3:10, 2020.

[22] Y. Geng, S. Liu, Z. Yin, A. Naik, B. Prabhakar, M. Rosenblum, and A. Vahdat, "Exploiting a natural network effect for scalable, fine-grained clock synchronization," in *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. Renton, WA: USENIX Association, Apr. 2018, pp. 81–94. [Online]. Available: https://www.usenix.org/conference/nsdi18/presentation/geng

[23] A. Guo, "Mapping applications onto fpga-centric clusters," Master's thesis, Department of Electrical and Computer Engineering, Boston University, 2020, proQuest Number: 27963166. [Online]. Available: https://open.bu.edu/handle/2144/40943

[24] X. Zhao, M. Papagelis, A. An, B. X. Chen, J. Liu, and Y. Hu, "Elastic bulk synchronous parallel model for distributed deep learning," in *2019 IEEE International Conference on Data Mining (ICDM)*, 2019, pp. 1504–1509.

[25] S. Wang, W. Chen, A. Pi, and X. Zhou, "Aggressive synchronization with partial processing for iterative ml jobs on clusters," in *Proceedings of the 19th International Middleware Conference*, ser. Middleware '18. New York, NY, USA: Association for Computing Machinery, 2018, p. 253–265. [Online]. Available: https://doi.org/10.1145/3274808.3274828