Runtime Adaptive Task Inlining on Asynchronous Multitasking Runtime Systems

Bibek Wagle*†
Center for Computation & Technology
Louisiana State University, Baton Rouge, Louisiana, USA
bwagle3@lsu.edu

Mohammad Alaul Haque Monil*
Computer and Information Science
University of Oregon, Eugene, Oregon, USA
mmonil@cs.uoregon.edu

Kevin Huck
Oregon Advanced Computing Institute for Science and Society
University of Oregon, Eugene, Oregon, USA
khuck@cs.uoregon.edu

Allen D. Malony
Computer and information Science, Oregon Advanced Computing Institute for Science and Society
University of Oregon, Eugene, Oregon, USA
malony@cs.uoregon.edu

Adrian Serio†
Center for Computation & Technology
Louisiana State University, Baton Rouge, Louisiana, USA
aserio@cct.lsu.edu

Hartmut Kaiser†
Center for Computation & Technology
Louisiana State University, Baton Rouge, Louisiana, USA
hkaiser@cct.lsu.edu

ABSTRACT

As the era of high frequency, single core processors have come to a close, the new paradigm of many core processors has come to dominate. In response to these systems, asynchronous multitasking runtime systems have been developed as a promising solution to efficiently utilize these newly available hardware. Asynchronous multitasking runtime systems work by dividing a problem into a large number of fine grained tasks. However, as the number of tasks created increase, the overheads associated with task creation and management cannot be ignored. Task inlining, a method where the parent thread consumes a child thread, enables the runtime system to achieve the balance between parallelism and its overhead. As largely impacted by different processor architectures, the decision of task inlining is dynamic in nature. In this research, we present adaptive techniques for deciding, at runtime, whether a particular task should be inlined or not. We present two policies, a baseline policy that makes inlining decision based on a fixed threshold and an adaptive policy which decides the threshold dynamically at runtime. We also evaluate and justify the performance of these policies on different processor architectures. To the best of our knowledge, this is the first study of the impacts of adaptive policy at runtime for task inlining in an asynchronous multitasking runtime system on different processor architectures. From experimentation, we find that the baseline policy improves the execution time from 7.61% to 54.09%. Furthermore, the adaptive policy improves over the baseline policy by up to 74%.

CCS CONCEPTS

• **General and reference** → *Measurement*; *Performance.*

KEYWORDS

Task inlining, AMTs, Asynchronous Task Based Runtimes

ACM Reference Format:
Bibek Wagle, Mohammad Alaul Haque Monil, Kevin Huck, Allen D. Malony, Adrian Serio, and Hartmut Kaiser. 2019. Runtime Adaptive Task Inlining on Asynchronous Multitasking Runtime Systems . In *48th International Conference on Parallel Processing (ICPP 2019), August 5–8, 2019, Kyoto, Japan.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3337821.3337915

1 INTRODUCTION

As Dennard scaling [3] breaks down and Moore's Law [17] is nearing its end, there is a paradigm shift from single core high frequency processors to multi-core processors. Asynchronous multitasking runtime systems have emerged in the recent years in order to take advantage of the abundance of cores in today's systems. Asynchronous multitasking runtime systems are founded on the idea of decomposing the algorithm into fine grained units of work and executing them asynchronously. However, the benefits of fine grained tasking are often overshadowed by the overheads associated with the creation and management of these tasks.

In order for these runtime systems to effectively utilize the highly concurrent nature of today's architectures, effective management of overheads associated with asynchronous multitasking runtime systems is of utmost importance. Task inlining [16] is one of the techniques which can be utilized in order to reduce overheads of task creation and scheduling. In this technique, a parent task completes the work assigned for the child task in addition to its own. In doing so, the child task is never scheduled on a separate thread which circumvents the overheads associated with creating

*Both authors contributed equally to this research.
†The STE||AR Group, http://stellar-group.org

ICPP 2019, August 5–8, 2019, Kyoto, Japan

ACM ISBN 978-1-4503-6295-5/19/08…$15.00
https://doi.org/10.1145/3337821.3337915

and managing the child task. As a direct result, inlining naturally increases the granularity of the tasks.

However, the success of task inlining solely depends on the decision when to inline a task. If an aggressive task inlining mechanism is applied, an application may lose the available parallelism which directly contradicts the objectives of a task-based runtime. On the other hand, if task inlining is done rarely then the application will face unnecessary task creation overheads. Therefore, the decision of when to inline a task can carry severe performance implications. The proper granularity of a task, or in other words the amount of work each task should perform to amortize the cost of its overhead, depends on the processors that the application is running on. Defining a granularity which is appropriate for a particular processor architecture will not yield the best results on other machines. This problem is further complicated in heterogeneous processor environments where an efficient grainsize of each task will vary drastically depending on the architecture that the task is executed on. Other technologies, such as cloud computing, again add complexity as the architecture where the application will be executed, and the accelerators available on the node, may not be known until runtime. For these reasons, utilizing a compile-time constant for defining task granularity would not be a viable option. The granularity must be set at runtime so that an appropriate granularity can be tuned for the type of processor the application is running on.

In this research, we present an adaptive approach to task inlining on Phylanx [21], an array processing toolkit built on top of HPX [15], an asynchronous multitasking runtime system. A description of HPX and Phylanx is presented in section 2. At first, we design a baseline policy based on fixed granularity that decides whether a task will be inlined or not. We show that the baseline policy performs significantly better than simply spawning tasks at every opportunity. Through experimentation, we show why fixed granularity is not enough. We also design an adaptive policy which can select the granularity at runtime to provide better performance. The reason for using Phylanx for experimentation is two-fold. First, Phylanx, has the notion of *primitives*, which are independent operations that work on provided data. By default, each of these *primitives* is scheduled on new threads, thus providing our experiments with a well-defined set of tasks of varying lengths. Second, Phylanx has been designed to support machine learning applications which are often iterative by nature and are well suited for adaptivity. This feature will provide us with opportunities to take measurements and adapt our application accordingly.

The contributions of this paper are as follows:

- designing a baseline policy that makes task inlining decisions based on fixed thresholds.
- showing the impact of task granularity on different kinds of processor architecture and proving the need for an adaptive policy.
- implementing a dynamic policy that can decide task granularity suited for a particular architecture to provide better performance.

The rest of the paper is organized as follows: The next section of the paper provides the background information on the subject followed by a section on methodology of the research. Experimental results are discussed in section 4 followed by recent related work in section 5 and finally the conclusion in section 6.

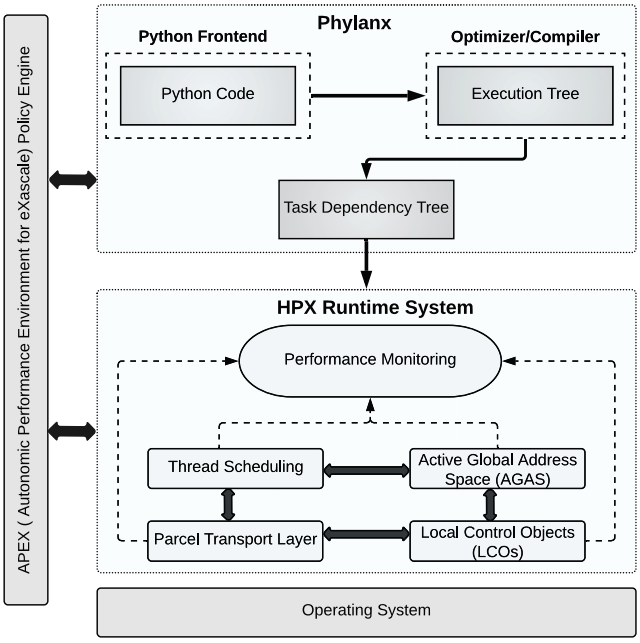

Figure 1: The architecture of HPX and Phylanx along with APEX. HPX consists of a *Threading Subsystem* responsible for scheduling *HPX threads* (lightweight tasks), a *Parcel Transport Layer* for handling message passing and remote method invocations, *Local Control Objects (LCOs)* for synchronization among tasks and an *Active Global Address Space (AGAS)* for addressing object across nodes. Phylanx, developed on top of HPX, transforms Python code into a task dependency tree which is executed by HPX. APEX provides performance monitoring facilities as well as a policy engine that enables runtime adaptive capabilities.

2 BACKGROUND

2.1 HPX

HPX is an asynchronous multi tasking runtime system with a C++ standards compliant API. The architecture of HPX along with its various subsystems is shown in Figure 1. Detailed information about HPX in [15]. In this section, we highlight the relevant information about HPX vital to the comprehension of the paper.

HPX exploits parallelism by executing lightweight tasks which are scheduled on top of the kernel threads. By default, HPX creates one kernel thread per core. The HPX scheduler schedules the lightweight tasks on top of these kernel threads. HPX is capable of executing a newly created task either as a new thread asynchronously or synchronously in the parent thread, which we will refer to as inlined execution. Asynchrony in HPX is managed via *futures* [1, 15]. A *future* is a placeholder for the result of some computation that is not yet ready. A task requesting the result of a *future* is suspended if the result is unavailable. When the *future* becomes ready, wherein the results of the computation is available, the suspended tasks are resumed.

Another important feature of HPX is the *dataflow* [4, 5] utility. HPX makes use of *dataflow* objects for managing data dependencies. A *dataflow* waits until a provided set of *futures* have become ready before executing a predefined callable which relies on the results referenced by the *futures*. In this work, we are able to use the *dataflow* objects as an injection point for our threading policies.

Finally, HPX provides a system wide support for gathering performance information, known as the performance counter framework. Users can employ this feature to extract information about the state of the application and runtime. If the pre-defined performance counters do not provide the user with needed functionality, one can easily create a new counter which will report the requested information. This tool is useful for instrumentation and debugging purposes. In addition, HPX and the performance counter framework integrate with APEX, described in Section 2.3, which provides additional measurements and runtime adaptive capabilities.

2.2 PHYLANX

Phylanx is a task based, asynchronous array computing toolkit designed to support machine learning applications. User code, written in Python, is transformed into a tree of Phylanx *primitives* known as an execution tree. A *primitive* is an object which can take input, such as the result of a previously executed *primitive*, and exposes a method named *eval* which performs an operation on the object's inputs. Instead of returning the value computed by the *primitive*, however, the *eval* function will return a *future* to the computed value. An execution tree is a collection of these objects which describe the dependencies between all the operations in an application. In this formulation, the nodes of the tree are the *primitives* while the edges of the tree represent dependencies between them. The architecture of Phylanx is shown in Figure 1.

During execution, Phylanx starts to evaluate the execution tree by calling the *eval* function on the root node. Each dependency of this *primitive* calls the *eval* function on each of its dependencies. This operation traverses the tree until a *leaf node*, or a node with no dependencies, is reached. It is important to note that as the execution tree is being traversed the actual execution of the tasks have not yet begun. Rather a task graph of futures is being created where each *future* represents a dependency on a previous operation. Once the leaf nodes have been reached, the task graph then begins to execute, as the execution of a leaf *primitive* does not depend on the results of another calculation. The task graph is then summarily executed as the results of dependencies are met, eventually returning the result of the entire tree. As the evaluation of a child node is completed, the result of its execution is passed to the parent node. The result of the entire tree is ready after the root node has finished execution.

Because *eval* uses HPX *dataflow* in order to launch a *primitive's* operations, we have a runtime injection point where we can decide whether to execute a *primitive's* children asynchronously in a new task or synchronously by inlining the execution. We have added Phylanx specific performance counters that report the amount of time spent executing each subtree of the execution tree, as well as a counter which reports the number of times a node was executed. Using these tools, we can take measurements of executing *primitives* and apply this information to future scheduling decisions.

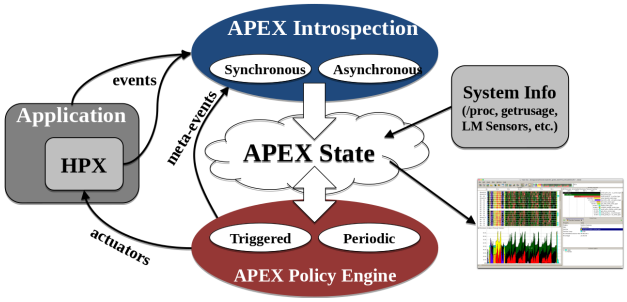

Figure 2: Interaction of APEX with the HPX runtime

2.2.1 Alternating Least Squares (ALS) Benchmark. In order to analyze the effects of overheads of scheduling and executing tasks on Phylanx, We used a reference implementations of Alternating Least Squares [11]. Figure 3 is a while loop taken of the reference implementation of the Alternating Least Squares in Phylanx. In Figure 4, this loop is visualized using the Phylanx visualization tool [22] where every node is a particular instance of a *primitive*. The full code for the benchmark can be found in the Phylanx Github repository [9].

2.3 APEX

APEX [12] (Autonomic Performance Environment for Exascale) is a performance measurement library for distributed, asynchronous mutlitasking runtime systems such as HPX. It provides lightweight measurement (task < 1ms) and high concurrency. To support performance measurement in systems that employ user level threading, APEX uses a dependency chain rather than the call stack to produce traces. APEX supports both synchronous and asynchronous introspection. As depicted in Figure 2, APEX collects data through inspectors. The synchronous module of APEX uses an event API and event listeners. Whenever an event occurs, APEX, using this aforementioned API, makes a decision to start, stop, yield or resume timers for correct measurements. The asynchronous module, however, does not rely on events, rather it executes desired functionality periodically.

The policy engine of APEX provides a lightweight API to engineer policies that can improve the performance of the application, execute a desired functionality on the runtime or select important runtime and application parameters. There are two ways to register a policy: 1. Triggered and 2. Periodic. A triggered policy can be initiated by a specific event within the HPX runtime. Several of these events are available by default to the user. Additionally, it is also possible to provide a user defined event, known as a custom trigger. The second class of policies, the periodic policy, operates without any event. Rather, this policy uses a defined timer which is specified during the policy's registration. All policies are stored in a policy queue and executed as instructed. The policy engine is integrated with Active Harmony [19], an online tuning library. Defined policies can use this library to converge on a set of optimum parameters by observing the wall time of the application or by looking at the introspection data gathered by APEX.

In this research, we use the policy engine to design and develop the inlined execution policy by observing execution time to find

out optimal threshold that decides whether a task will be executed synchronously or asynchronously.

3 METHODOLOGY

In order to adaptively decide which tasks we want to inline, we outline two techniques. The first is semi-automated wherein user input regarding inlining threshold is required before execution whereas the second one is fully automated wherein all decisions are handled automatically by the runtime system.

3.1 Baseline Policy

As described earlier in section 2.2 , each *primitive* in Phylanx has a method called *eval* which evaluates the work defined by that *primitive*. HPX can decide whether to execute the *eval* method asynchronously as a new task or synchronously by inlining the work in the parent task.

The baseline policy for task inlining is shown in Algorithm 1. Given an iterative application, the execution time for each *primitive* instance is evaluated *count_threshold* times in order to obtain the average execution time of the *primitive* instance. If during the lifetime of the application, *count_threshold* measurements are not obtained for a *primitive*, no decision will be made regarding the inlining of the task. If the previous *primitive* was executed asynchronously the next execution will be executed asynchronously. Conversely, the execution will be synchronous if the previous execution was synchronous. On the other hand, if measurements are obtained and the average execution time is below the *lower_threshold*, any future tasks created for that *primitive* instance will be executed synchronously and if the average execution time is above the *upper_threshold*, any future tasks created for that *primitive* instance will be executed asynchronously. In case where the average execution time of the *primitive* instance lies between the thresholds, the task will be executed with its previous mode of execution until more measurements for the execution time is gathered.

A drawback of the baseline policy lies in the fact that the optimal threshold values varies with different architectures and the number of threads used. We will discuss this in more detail in section 4.2. This warrants the use of runtime adaptive policies for inlined execution. The default values for thresholds were initially chosen by benchmarking overheads associated with HPX *futures*. The default values for *count_threshold* is set to 5, *lower_threshold* is set to 350 μs and the default for *upper_threshold* is set to 500 μs.

3.2 Adaptive Policy

3.2.1 Motivation behind the adaptive policy. Adaptive policies are proven techniques used to tune parameters that depend on architecture, the amount of parallelism and the communication pattern that the application exhibits. Runtime adaptivity can be applied to runtime systems (like HPX) or on the framework (Phylanx) itself. The motivation for using adaptive policies emerges from the limitations exhibited by the baseline policy. These are described in the previous section and are supported by experimental data in the section 4.2. A fixed threshold might work for a given architecture and known application characteristics but for a truly heterogeneous system, where different nodes are participating in a distributed environment, setting a generic threshold for all the nodes is not practical. One way to solve this problem would be to determine the threshold at compile time. However, in many workflows applications are compiled on a login node and then executed on different machines. Therefore, it is sensible to determine the threshold for task inlining at runtime instead.

Algorithm 1 Baseline Policy

$exec\_count \Leftarrow 0$
$exec\_time \Leftarrow 0$
$count\_threshold \Leftarrow 5$
$lower\_threshold \Leftarrow 350000$
$upper\_threshold \Leftarrow 500000$
for <For Every Primitive Instance in Parallel> **do**
  if $exec\_count \geq count\_threshold$ **then**
    $average\_exec\_time \Leftarrow \frac{exec\_time}{exec\_count}$
  end if
  if $average\_exec\_time \geq upper\_threshold$ **then**
    $inline\_task = false$
  else if $average\_exec\_time \leq lower\_thredhold$ **then**
    $inline\_task = true$
  else
    $inline\_task = undecided$
  end if
end for

```
while(i < num_items,
    block(
        store(conf_i, slice_column(conf, i)),
        store(c_i, diag(conf_i)),
         store(p_i, __ne(conf_i, 0.0, true)),
         store(A, dot(dot(transpose(X),c_i), X) + XtX),
         store(b, dot(dot(transpose(X),(c_i + I_u)), transpose(p_i))),
         store(slice(Y, list(i, i + 1, 1),nil), dot(inverse(A), b)),
         store(i, i + 1)
    )
),
```

Figure 3: Partial code for the while loop in the Alternating Least Squares algorithm

3.2.2 What to tune and which metrics indicate better performance. In the baseline policy, the decision regarding task inlining is made based on a fixed threshold. However, if we observe closely, the two thresholds (the upper and the lower) are providing a gap which acts as a hysteresis so that the decision does not fluctuate when there is a small change in the execution time. So, tuning one threshold with a fixed hysteresis is logical and helps to reduce the search space and overhead of the policy. The threshold can be tuned based on observed average execution time for the *primitive* instances for a defined window.

3.2.3 Granularity of the adaptive Policy. An important question arises about the granularity of the adaptive policy. Should there be one threshold for each type of *primitive* for the entire application or is a more granular control of the threshold desired? To answer this question we look at the structure of the Phylanx framework. As described in the previous sections, Phylanx translates Python-like user code into an execution tree made up of Phylanx *primitives*. Each instance of each type of *primitive* exhibits its own behavior as

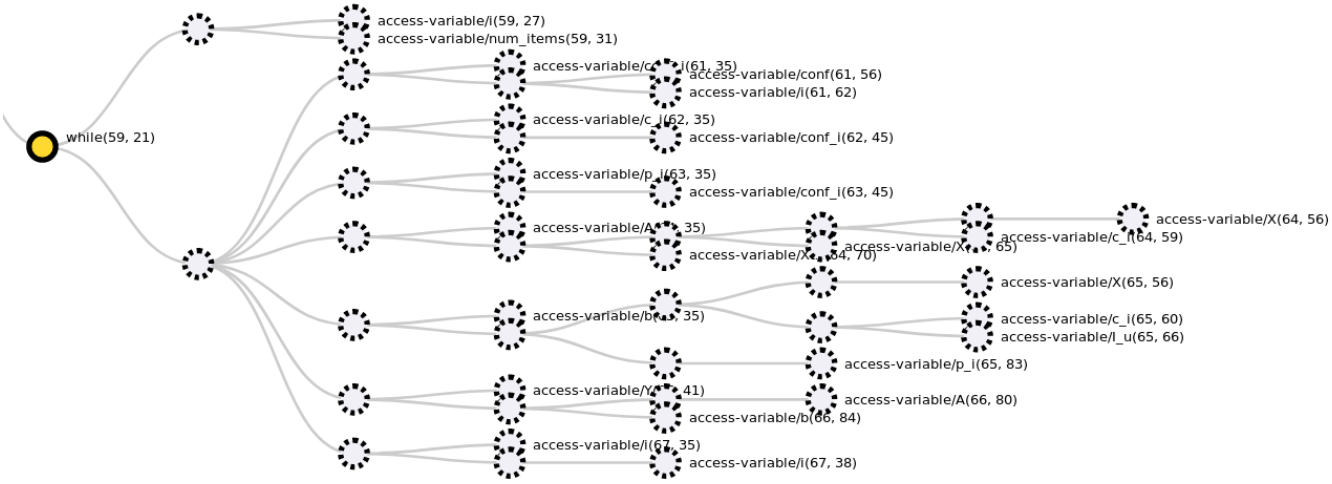

Figure 4: Visualization of the while loop in the Alternating Least Squares algorithm

the inputs to each *primitive* may be different. For this reason, tuning a class of *primitives* to a common threshold does not make much sense and adapting the threshold for each instance of a *primitive* is expedient.

As mentioned in the description of the Alternating Least Squares benchmark in section 2.2.1, the Phylanx framework will create a different number of HPX tasks depending on the inputs to the algorithm. We define the threshold for each of the instances and tune the threshold based on its average execution time for an observed period to decide on a threshold that improves performance (reduce execution time) for that instance only. There are couple of challenges that come with this approach. The first challenge is to create an APEX policy instance for each primitive we need to tune and the second challenge is to manage the overhead that would be incurred from creating these policies. The first challenge is addressed by APEX itself as it can handle more than thousand policies to tune parameters. For the second challenge, an overhead study is given on the experimental result section.

3.2.4 How the adaptive Policy works. The pseudo code of adaptive APEX policy is shown in Algorithm 2. In the beginning, all *primitive* instances are set to use a default threshold and hysteresis value. It is important to note that a policy is not required for every *primitive* instance. A policy should only be created when the instance will be executed many times, such as when the *primitive* is within a for loop. Only then will the policy have the opportunity to converge. In order to launch a policy, it has to be registered to APEX policy engine. We defined a custom policy which is triggered when *eval* or *exec_count* is called more than a *count_threshold_1* (at least 5 times). During registration, several properties of the policy are defined including the search space, the search strategy and the tuning parameter. The search space is the number of values which are needed to be tested in order to find an optimum value. The search strategy refers to the algorithms used to determine the optimum values. APEX uses search strategies provided by Active Harmony to find these values in a given search space. Active Harmony provides strategies such as EXHAUSTIVE which tests every possible combination and determines the most efficient one and PARALLEL_RANK_ORDER which attempts to find a local minima in the search space which is suboptimal. Each of these strategies work best when combined with an appropriately sized search space.

Finally, the tuning parameter is the metric that determines the effectiveness of different parameter settings. In our case we use the average execution time of the *primitive* to tune our policy. Due to the overheads associated with measuring tuning parameters and executing policies it is important to be able to cease triggering the policy. We do this by monitoring convergence of our policy parameter. Once we have determined that a policy is converged, the policy is de-registered and it will no longer be triggered.

When a *primitive* instance reaches the defined number of executions, the custom policy is called and APEX launches the policy. APEX grabs the metric and a threshold from the search space and sends them to Active Harmony. Active Harmony stores the metric and the threshold and uses the defined search strategy to propose the next threshold to APEX. APEX sets this threshold in HPX and observes the impact of the decision at the next policy invocation. After the policy is successfully invoked the counters are reset to start a fresh observation window. Every time the policy is executed the search for optimal threshold progresses. Based on the search strategy, when the impact of all (or a subset of) possible thresholds are tried Active Harmony sends a signal to APEX about convergence and the policy is deregistered after setting the optimal threshold in HPX. From this point, this threshold is used to make the task inlining decision for the rest of the application's execution.

After the policy is invoked or checked for convergence, the average execution time is calculated. Similar to the baseline policy, if the average execution time is bigger than the threshold plus hysteresis then a new task is created. If the average execution is smaller, the work will be attached as a continuation to the current task. The section below provides preliminary results from our investigation of measuring task scheduling and execution overheads.

Algorithm 2 Adaptive APEX Policy

```
for <For Every Primitive Instances in Parallel> do
    Threshold ⇐ 425000
    Hysteresis ⇐ 75000
    if exec_count ≥ count_threshold_1 then
        RegisterAPEXPolicy
    end if
    if exec_count ≥ count_threshold_2 && PolicyNotConverged
then
        LaunchAPEXPolicy
        SendCounterValuesToActiveHarmony
        RecieveNewThresholdFromAPEX
        ConfigureThresholdToHPX
        ResetCounter
    end if
    if exec_count ≥ count_threshold_1 then
        avg_exec_time ⇐ exec_time / exec_count
        if avg_exec_time ≥ Threshold + Hysteresis then
            inline_task = false
        else if avg_exec_time ≤ Threshold − Hysteresis then
            inline_task = true
        else
            inline_task = undecided
        end if
    end if
end for
```

Table 1: Specifications of Nodes used

Node	Marvin	Bahram	Trillian
Microarchitecture	Sandy Bridge	Haswell	Bulldozer
Processor Number	E5-2450	E5-2660v3	6272
Number of CPUs	2	2	4
Cores per CPU	8	10	16
Total Cores	16	20	64
Frequency	2.1GHz	2.60GHz	2.1GHZ
Memory	48 GB	128GB	128GB

4 EXPERIMENTAL RESULTS

4.1 Experimental Testbed

For the preliminary experiments, we used the Marvin, Bahram and Trillian nodes of the ROSTAM [8] cluster located at LSU running the 64 bit Centos GNU/Linux kernel version 3.10.0. The specifications for the nodes are listed in Table 1

4.2 Inlining Threshold

The results of executing the Alternating Least Squares benchmark for an input of size 400 x 400 elements and 10 iterations on Sandy Bridge, Haswell and Bulldozer nodes are shown in Figure 5, Figure 6 and Figure 7 respectively. We observed that for different values of the lower and upper thresholds there was a marked difference in execution time of the application on all three architectures. Since a task is executed asynchronously only if the execution time of a task is greater than the upper threshold, smaller threshold values results in more short lived tasks being executed asynchronously.

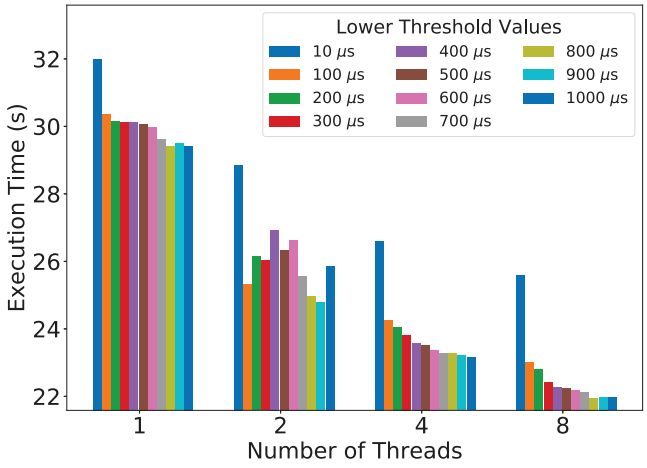

Figure 5: Variation in execution time of the Alternating Least Squares benchmark for different threshold values on Intel Xeon (Sandy Bridge) node with upper threshold being set at 100 μs above lower threshold. The size of the input for the benchmark was 400 x 400 elements and the application was run for 10 iterations.

In our experiments, upper threshold was set at 100 μs above lower threshold. Our experiments showed that the optimal value for the threshold varies with the number of threads used. On the Sandy Bridge nodes, the optimal value for the lower threshold was 800 μs for 1 thread and 8 thread case, whereas it was 900 μs and 1000 μs for 2 threads and 4 threads respectively. Similar trends were noted for the Haswell nodes wherein the optimal threshold varied depending upon the number of threads used.

We noted that for higher thread counts, the overall runtime of the application does not vary much by changing the threshold after a certain point. As an example, setting the threshold at 800 μs for 8 thread case on the Sandy Bridge nodes resulted in an improvement of the overall runtime by less than 5% compared to the a suboptimal threshold of 300 μs. Similar results were noted for the Bulldozer nodes as well as the Haswell architecture. Moreover, it was observed that on the Bulldozer node, the improvements starts diminishing after a much higher threshold value of 900 μs. This trend clearly demonstrates that the inlining threshold varies between architectures and is a variable that we can adapt at runtime in order to improve application performance.

4.3 Baseline Policy

The improvement in execution time obtained by executing the Alternating Least Squares benchmark for a various matrix sizes on the Sandy Bridge and Bulldozer nodes is shown in Figure 8 and Figure 9. Results from Haswell nodes are not shown as they were similar to Sandy Bridge results. We executed the Alternating Least Squares benchmark with four different problem sizes on varying number of threads under two conditions. In the first case, the full problem was run with completely asynchronous execution. This was achieved

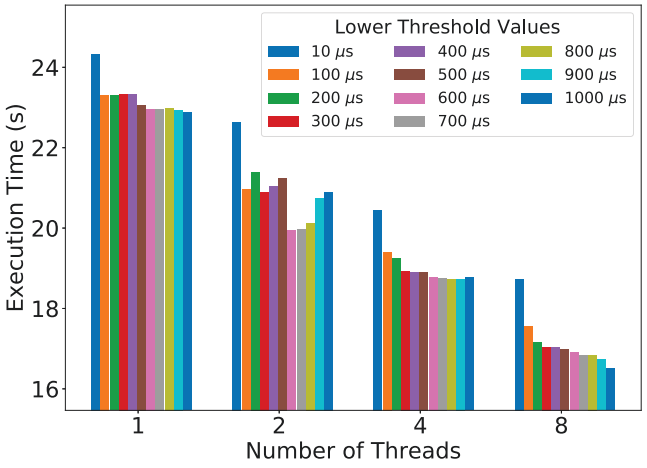

Figure 6: Variation in execution time of the Alternating Least Squares benchmark for different threshold values on Intel Xeon (Haswell) node with upper threshold being set at 100 μs above lower threshold. The size of the input for the benchmark was 400 x 400 elements and the application was run for 10 iterations.

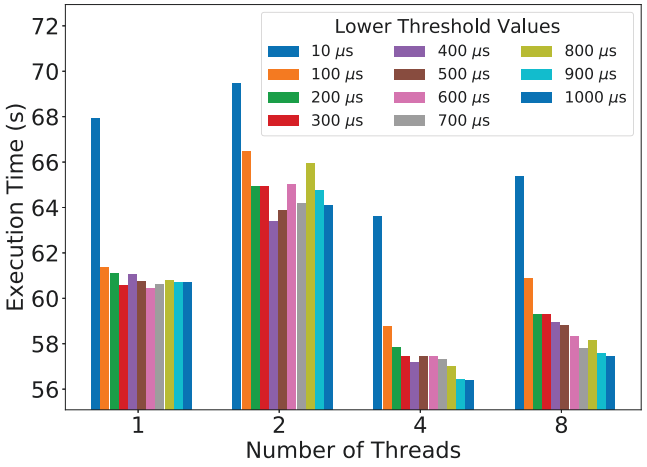

Figure 7: Variation in execution time of the Alternating Least Squares benchmark for different threshold values on AMD Opteron (Bulldozer) node with upper threshold being set at 100 μs above lower threshold. The size of the input for the benchmark was 400 x 400 elements and the application was run for 10 iterations.

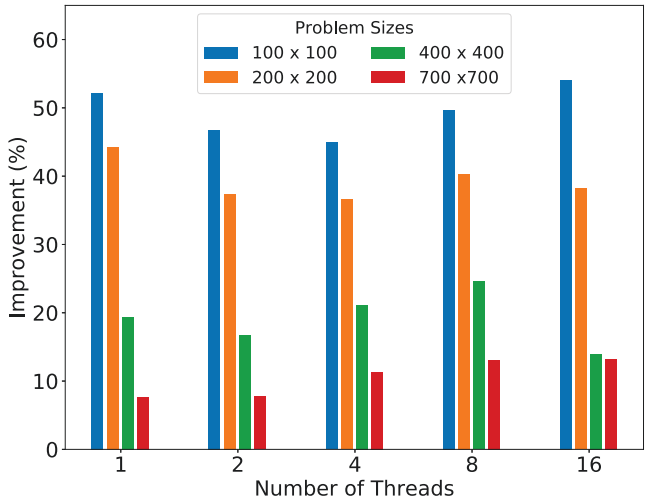

Figure 8: Percentage improvement in execution time obtained from using the Baseline Policy on Intel Xeon (Sandy Bridge) node with respect to fully asynchronous execution for various problem sizes for the Alternating Least Squares Benchmark.

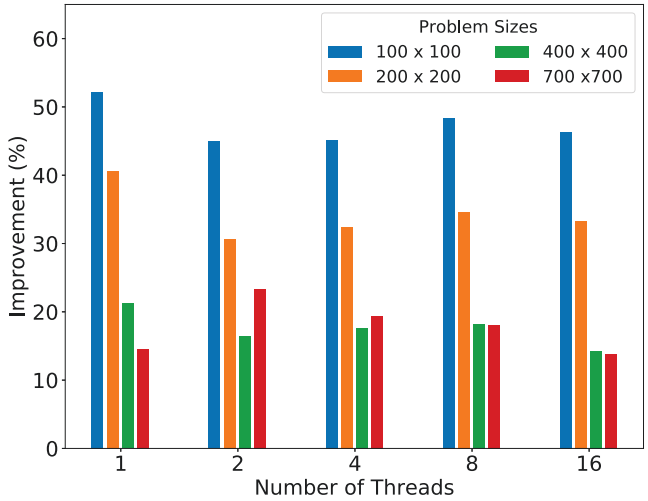

Figure 9: Percentage improvement in execution time obtained from using the Baseline Policy on AMD Opteron (Bulldozer) node with respect to fully asynchronous execution for various problem sizes for the Alternating Least Squares Benchmark.

by setting the *upper_threshold* and *lower_threshold* to zero. In the second case, no values were selected for the *upper_threshold* and *lower_threshold* parameters. This resulted in the program being run with the baseline policy set with the default values for the thresholds.

We observed that on both architectures, the baseline policy had the most benefit on smaller problem sizes. Smaller problem sizes decrease the amount of work contained in each task. Thus the overheads associated with creating and managing these tasks overshadows the gains of concurrency provided by asynchronous execution. Nevertheless, as the problem size is increased, the benefits of task inlining gradually decreases, as expected, because larger problem sizes result in larger tasks that have runtimes significantly longer than the overheads required to create them, thereby amortizing their cost.

4.4 The Adaptive APEX Policy

From the previous sections, it is evident that the baseline policy provides better results when compared to a default asynchronous execution. In this section, we compare the APEX policy with the baseline policy on different kinds of processors.

4.4.1 AMD processors. In order to observe the impact of the APEX policy, we ran the Alternating Least Squares benchmark on an AMD processor (Bulldozer) with different number of threads and varying number of iterations. Each experiment was run 5 times and then the results were averaged to ensure that the result is invariant with respect to other interferences the node might have. At first, we ran the application using the baseline policy and then, we ran the same experiments with the APEX policy turned on. The result is depicted in Figure 10.

We observed a considerable amount of improvement while using the APEX policy for each experiment. The APEX policy improved upon the baseline policy up to 74%. It is clear that for various numbers of threads the APEX policy provides a consistent behavior whereas the baseline policy was unable to keep the execution time from increasing with the number of threads. The experiment demonstrates that the APEX policy is successfully able to set the threshold to a value which reduces the tasking overhead by eliminating unnecessary task creation and scheduling. Additionally, our experiments do not exhibit any scaling. This tells us that there is not enough parallelism in the application itself and suggests that we should run the policy with larger problem sizes. Nevertheless, it is clear that the APEX policy is not impacted by the available parallelism of the application and it enables Phylanx users to run applications without worrying about the number of threads they are going to use.

In order to create enough parallelism in the application, we use the Phylanx ALS algorithm with a bigger data set. This application takes the data set from a CSV file instead of using a default dataset. For these experiments, we keep varying the data size (in row and columns) and number of threads. The result is portrayed in Figure 11. We observe that the baseline policy does scale as we use more threads. This supports our previous conjecture that our previous experiments would benefit from more parallelism. However, even these larger data sets provide the algorithm with a limited amount of parallelism. After several threads have been added to the execution, the execution time increases. For example, if we select the data set with 500 rows and 5000 columns, the benchmark will take 1119 seconds to execute using the baseline policy, as we increase the number of threads to 8 and 16 we see the execution time decrease to 831 seconds and to 805 seconds respectively. However, when the number of threads goes higher than 16, we can see the execution time begins to increase.

Now, we observe the benefit of APEX policy from the same Figure 11. For 18 cases that we depicted in this figure, we can find improvement in 16 cases while two cases show that the baseline policy performs better with a margin of ~1%. However, for the 16 cases where APEX policy shows improvement varies in a range of 0.4% - 16%. The average improvement for all the improving case is 5% (standard deviation is 4 %). So we can expect 1% - 9% improvement for most cases. Even though large improvements in the execution time of the application was not seen as opposed to the previous case with small data sizes, APEX policy still provides a considerable improvement margin for an application with a large data set and enough parallelism.

4.4.2 Intel Xeon processors. Section 4.2 demonstrated the importance of task inlining on various architectures. In this section, we conduct similar experiments as the previous one but on an Intel Xeon processors. We ran our experiments on the Marvin nodes of the ROSTAM [8] cluster. The results are depicted in Figure 12. A similar trend as Figure 10 is found in this experiment. However, there is not much improvement visible from the APEX policy. We observe almost identical execution time with the baseline and APEX policy. Out of 20 experiments, we found improvement in 7 cases and performance degradation for the remaining cases. The average improvement for the 7 cases is 1.8% while for the 13 cases where the baseline policy performed better the average stands on 2%. On average there is no improvement found from using the APEX policy.

The APEX policy determines the optimal threshold for each primitive instance and the baseline policy defines the static threshold. If the baseline policy defined threshold is already optimal or close to the optimal, then we will not find a visible difference between the baseline and APEX policy. As described earlier, the baseline policy sets the upper and the lower threshold to 350 and 500 μs respectively. Looking back at the result presented in Figure 5, we find that as long as the lower threshold is more than 300 μs almost all the threshold configurations provide similar results. Furthermore, the lower threshold of the baseline policy is 350 μs the baseline policy is providing a close to the optimal result. For this reason, we are not observing any visible improvement from the APEX policy. Moreover, APEX policy contributes a small amount of overhead which in turn negates whatever small improvement would have been observed.

4.4.3 Overhead of the APEX policy. Theoretically, the overhead of the APEX policy is constant for an algorithm and does not change with the data size. Every algorithm has a fixed number of primitive instances that call an APEX policy a fixed number of times. In order to measure the overhead, we have compared 100 runs of the Alternating Least Squares benchmark each using different data sizes. We found that for this algorithm the APEX policy introduces a total of 5 seconds, on average, to the execution time. For a larger data set, where the Alternating Least Squares benchmark takes 30 minutes to execute, 5 seconds is a negligible amount of overhead.

5 RELATED WORK

Lazy task creation strategy was proposed in [16] where task creation was avoided until processing resources were free. With regards to OpenMP tasks, Duran [7] proposed cut-off technique in order to improve performance. The cutoff was based on either the max number of tasks in the system or max task recursion level. ATC (Adaptive Task Cutoff) was proposed in [6], where the cutoff decision was based on the profiling data obtained from the application at runtime and assumes that all tasks at a given level will have same similar behavior. Adaptive Task granularity(ATG) was proposed in [2] for irregular task parallel programs. ATG switches between help first and serialization policy depending upon the number of tasks created in the system. However effects of processor architectures were

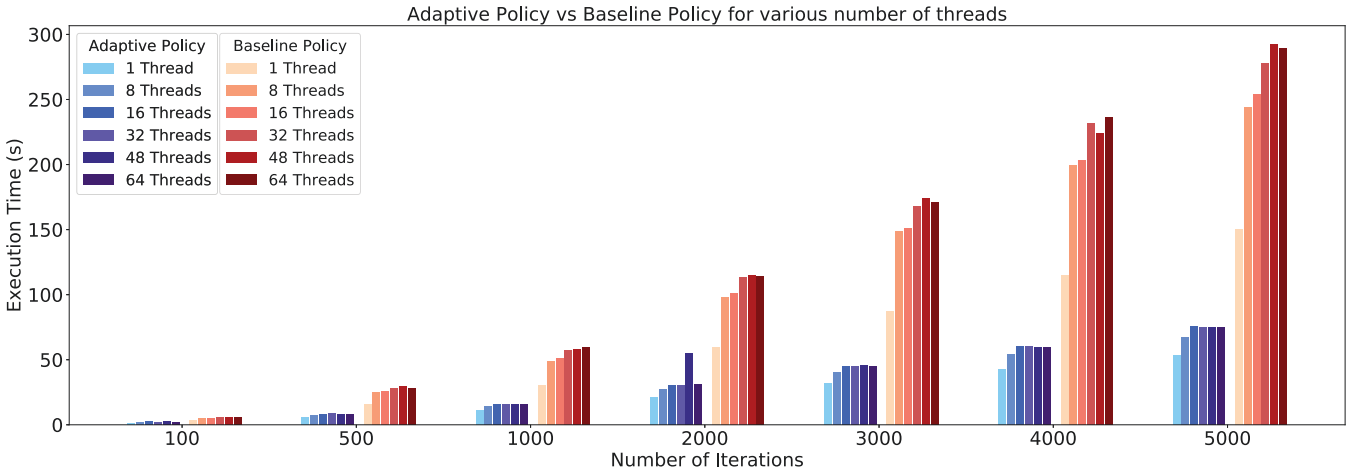

Figure 10: Comparison of the Adaptive APEX and Baseline policies on AMD Opteron (Bulldozer) node for the Alternating Least Squares benchmark for an input size of 10 x 5 elements running for various iterations and number of threads.

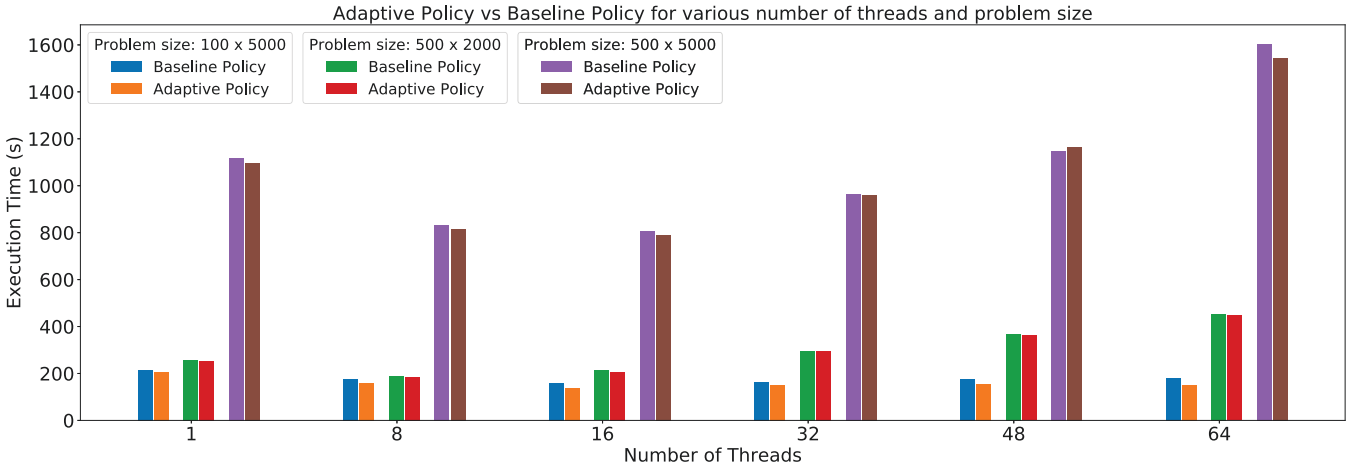

Figure 11: Comparison of the Adaptive APEX and Baseline policies on AMD Opteron (Bulldozer) node for the Alternating Least Squares benchmark for an input size of 100 x 5000, 500 x 2000 and 500 x 5000 elements and various number of threads.

not considered. With regards to compiler based approaches, a multiversioning approach was proposed in [20], where a combination of compiler and runtime approach was used. Here, multiple versions of tasks with varying granularity was generated at compile time and one was then selected at runtime by tracking task demand. A compiler based static cutoff along with two optimizations namely code-bloat-free inlining and loopification was proposed in [14]. An auto-tuning framework for divide and conquer task parallel programs was proposed in [13] which was implemented as an optimization pass in LLVM. In the context of asynchronous multitasking runtime systems, Sun [18] developed the ParSSSE (Parallel State Space Search Engine) Framework for Charm++ and looked at adaptive grain size control in the context of parallel state search methods. Grubel [10], used performance counters in HPX for dynamically tuning grain size of 1d-stencil application. Our proposed method is application agnostic and no change in application source code is required. Furthermore, our proposed method relies on the actual execution time of the tasks in order to make decisions regarding task inlining for future executions of the tasks.

6 CONCLUSION AND FUTURE WORK

In this paper, we presented runtime adaptive techniques for task inlining using HPX and APEX. We applied our methods on the Alternating Least Squares benchmarks in Phylanx. We are able to show that our method of task inlining does not add any substantial overhead in the test applications. Furthermore, we were able to show that our method provided an improvement over completely asynchronous execution of iterative machine learning applications via baseline policy and an advanced policy using APEX. One of the limitations of the method outlined in this paper is the fact that we could not conduct experiments and analyze the impact of the policy in a heterogeneous environment. We are currently working

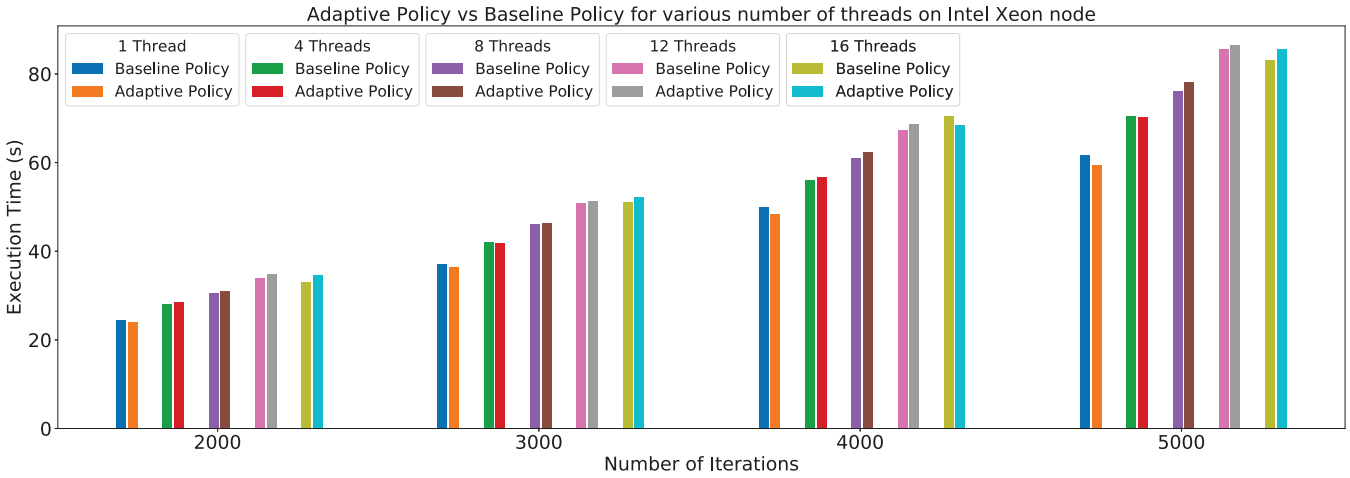

Figure 12: Comparison of the Adaptive APEX and Baseline policies on Intel Xeon (Sandy Bridge) node for the Alternating Least Squares benchmark for an input size of 10 x 5 elements running for 2000, 3000, 4000 and 5000 iterations.

on the GPU version of Phylanx and in the future, we plan to study the impact of dynamic policies in heterogeneous environments.

ACKNOWLEDGMENTS

This work was funded by the NSF Phylanx project award #1737785 and the Department of Defense (DoD) through DTIC Contract FA8075-14-D-0002/0007.

REFERENCES

[1] Henry C. Baker, Jr. and Carl Hewitt. 1977. The Incremental Garbage Collection of Processes. In *Proceedings of the 1977 Symposium on Artificial Intelligence and Programming Languages*. ACM, New York, NY, USA, 55–59. https://doi.org/10.1145/800228.806932

[2] Jianmin Bi, Xiaofei Liao, Yu Zhang, Chencheng Ye, Hai Jin, and Laurence T. Yang. 2014. An Adaptive Task Granularity Based Scheduling for Task-centric Parallelism. In *Proceedings of the 2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS) (HPCC '14)*. IEEE Computer Society, Washington, DC, USA, 165–172. https://doi.org/10.1109/HPCC.2014.32

[3] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. 1974. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits* 9, 5 (Oct 1974), 256–268. https://doi.org/10.1109/JSSC.1974.1050511

[4] J. B. Dennis. 1974. First Version of a Data Flow Procedure Language. In *Programming Symposium, Proceedings Colloque Sur La Programmation*. Springer-Verlag, Berlin, Heidelberg, 362–376. http://dl.acm.org/citation.cfm?id=647323.721501

[5] Jack B. Dennis and David P. Misunas. 1975. A Preliminary Architecture for a Basic Data-flow Processor. In *Proceedings of the 2Nd Annual Symposium on Computer Architecture (ISCA '75)*. ACM, New York, NY, USA, 126–132. https://doi.org/10.1145/642089.642111

[6] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. 2008. An Adaptive Cut-off for Task Parallelism. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC '08)*. IEEE Press, Piscataway, NJ, USA, Article 36, 11 pages. http://dl.acm.org/citation.cfm?id=1413370.1413407

[7] Alejandro Duran, Julita Corbalán, and Eduard Ayguadé. 2008. Evaluation of OpenMP Task Scheduling Strategies. In *OpenMP in a New Era of Parallelism*, Rudolf Eigenmann and Bronis R. de Supinski (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 100–110.

[8] Ste||ar Group. 2017. Running HPX on ROSTAM. https://github.com/STEllAR-GROUP/hpx/wiki/Running-HPX-on-Rostam. (2017).

[9] Ste||ar Group. 2018. ALS algorithm code in PHYSL. https://github.com/STEllAR-GROUP/phylanx/blob/master/examples/algorithms/als/als.physl. (2018).

[10] Patricia A Grubel. 2016. *DYNAMIC ADAPTATION IN HPX - A TASK-BASED PARALLEL RUNTIME SYSTEM*. Ph.D. Dissertation.

[11] Yifan Hu, Yehuda Koren, and Chris Volinsky. 2008. Collaborative filtering for implicit feedback datasets. In *Data Mining, 2008. ICDM'08. Eighth IEEE International Conference on*. Ieee, 263–272.

[12] Kevin A Huck, Allan Porterfield, Nick Chaimov, Hartmut Kaiser, Allen D Malony, Thomas Sterling, and Rob Fowler. 2015. An autonomic performance environment for exascale. *Supercomputing frontiers and innovations* 2, 3 (2015), 49–66.

[13] S. Iwasaki and K. Taura. 2016. Autotuning of a Cut-Off for Task Parallel Programs. In *2016 IEEE 10th International Symposium on Embedded Multicore/Many-core Systems-on-Chip (MCSoC)*. IEEE Computer Society, Los Alamitos, CA, USA, 353–360. https://doi.org/10.1109/MCSoC.2016.51

[14] S. Iwasaki and K. Taura. 2016. A static cut-off for task parallel programs. In *2016 International Conference on Parallel Architecture and Compilation Techniques (PACT)*. 139–150. https://doi.org/10.1145/2967938.2967968

[15] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 2014. HPX: A Task Based Programming Model in a Global Address Space. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models (PGAS '14)*. ACM, New York, NY, USA, Article 6, 11 pages. https://doi.org/10.1145/2676870.2676883

[16] Eric Mohr, David A. Kranz, and Robert H. Halstead, Jr. 1990. Lazy Task Creation: A Technique for Increasing the Granularity of Parallel Programs. In *Proceedings of the 1990 ACM Conference on LISP and Functional Programming (LFP '90)*. ACM, New York, NY, USA, 185–197. https://doi.org/10.1145/91556.91631

[17] Gordon E Moore et al. 1965. Cramming more components onto integrated circuits. (1965).

[18] Yanhua Sun, Gengbin Zheng, Pritish Jetley, and Laxmikant V. Kalé. 2011. Parssse: an Adaptive Parallel State Space Search Engine. *Parallel Processing Letters* 21, 3 (2011), 319–338. https://doi.org/10.1142/S0129626411000242

[19] Cristian Ţăpuş, I-Hsin Chung, Jeffrey K Hollingsworth, et al. 2002. Active harmony: Towards automated performance tuning. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*. IEEE Computer Society Press, 1–11.

[20] Peter Thoman, Herbert Jordan, and Thomas Fahringer. 2013. Adaptive Granularity Control in Task Parallel Programs Using Multiversioning. In *Euro-Par 2013 Parallel Processing*, Felix Wolf, Bernd Mohr, and Dieter an Mey (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 164–177.

[21] R. Tohid, B. Wagle, S. Shirzad, P. Diehl, A. Serio, A. Kheirkhahan, P. Amini, K. Williams, K. Isaacs, K. Huck, S. Brandt, and H. Kaiser. 2018. Asynchronous Execution of Python Code on Task-Based Runtime Systems. In *2018 IEEE/ACM 4th International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. 37–45. https://doi.org/10.1109/ESPM2.2018.00009

[22] Katy Williams, Alex Bigelow, and Kate Isaacs. 2019. Visualizing a Moving Target: A Design Study on Task Parallel Programs in the Presence of Evolving Data and Concerns. *arXiv e-prints*, Article arXiv:1905.13135 (May 2019), arXiv:1905.13135 pages. arXiv:cs.HC/1905.13135