HPX Smart Executors

Zahra Khatami
Center for Computation and Technology
Louisiana State University
The STE||AR Group, http://stellar-group.org
Baton Rouge, LA, USA
zkhata@lsu.edu

Lukas Troska
Center for Computation and Technology
Louisiana State University
The STE||AR Group, http://stellar-group.org
Baton Rouge, LA, USA
lukas.troska@gmail.com

Hartmut Kaiser
Center for Computation and Technology
Louisiana State University
The STE||AR Group, http://stellar-group.org
Baton Rouge, LA, USA
hkaiser@cct.lsu.edu

J. Ramanujam
Center for Computation and Technology
Louisiana State University
The STE||AR Group, http://stellar-group.org
Baton Rouge, LA, USA
ram@cct.lsu.edu

Adrian Serio
Center for Computation and Technology
Louisiana State University
The STE||AR Group, http://stellar-group.org
Baton Rouge, LA, USA
aserio@cct.lsu.edu

ABSTRACT

The performance of many parallel applications depends on loop-level parallelism. However, manually parallelizing all loops may result in degrading parallel performance, as some of them cannot scale desirably to a large number of threads. In addition, the overheads of manually tuning loop parameters might prevent an application from reaching its maximum parallel performance. We illustrate how machine learning techniques can be applied to address these challenges. In this research, we develop a framework that is able to automatically capture the static and dynamic information of a loop. Moreover, we advocate a novel method by introducing HPX smart executors for determining the execution policy, chunk size, and prefetching distance of an HPX loop to achieve higher possible performance by feeding static information captured during compilation and runtime-based dynamic information to our learning model. Our evaluated execution results show that using these smart executors can speed up the HPX execution process by around 12% – 35% for the Matrix Multiplication, Stream and 2D Stencil benchmarks compared to setting their HPX loop's execution policy/parameters manually or using HPX auto-parallelization techniques.

KEYWORDS

HPX, Logistic Regression Model, ClangTool.

ACM Reference format:
Zahra Khatami, Lukas Troska, Hartmut Kaiser, J. Ramanujam, and Adrian Serio. 2017. HPX Smart Executors. In *Proceedings of ESPM2'17: Third International Workshop on Extreme Scale Programming Models and Middleware, Denver, CO, USA, November 12–17, 2017 (ESPM2'17),* 8 pages.
https://doi.org/https://doi.org/10.1145/3152041.3152084

ESPM2'17, November 12–17, 2017, Denver, CO, USA

ACM ISBN 978-1-4503-5133-1/17/11…$15.00
https://doi.org/https://doi.org/10.1145/3152041.3152084

1 INTRODUCTION

Runtime information is often speculative. While runtime adaptive methods have been shown to be very effective – especially for highly dynamic scenarios – solely relying on them doesn't guarantee maximal parallel performance, since the performance of an application depends on both the values measured at runtime and the related transformations performed at compile time. Collecting the outcome of the static analysis performed by the compiler could significantly improve runtime decisions and therefore application performance [1–4].

There are many existing publications on automatically choosing optimization parameters based on static information extracted at compile time. For example in [5, 6] optimal scheduling for parallel loops is implemented dynamically at runtime by examining data dependencies captured at compile time. However, one of the challenges in these studies is the need to repeat their proposed methods for each new program, which in general is not desirable, as it requires extra execution time for each application for such parameters determination. Moreover, manually tuning parameters becomes ineffective and almost impossible when the parallel performance depends on too many parameters as defined by to the program. Hence, many researches have extensively studied machine learning algorithms which optimize such parameters automatically.

For example in [7], a neural network and decision tree are applied on the training data collected from different observations to predict the branch behavior in a new program. In [1] nearest neighbors and support vector machines are used for predicting unroll factors for different nested loops based on the extracted static features. In [8, 9], the logistic regression model is used to derive a learning model, which results in a significant speedup in compilation time of their studied benchmarks. Most of these optimization techniques require users to compile their application twice, first compilation for extracting static information and the second one for recompiling application based on those extracted data. None of these considers both static and dynamic information.

The goal of this paper is to optimize an HPX application's performance by predicting optimum parameters for its parallel algorithms by considering both static and dynamic information and to avoid unnecessary compilation. As all of the HPX parallel algorithms perform based on the dynamic analysis provided at runtime, this

technique is unable to achieve the maximum possible parallel efficiency in some cases:

- In [10, 11] different policies for executing HPX parallel algorithms are studied. However these policies should be manually selected for each algorithm within an application, which may not be an optimum approach, as a user should execute each parallel algorithm of his application with different execution policies to find the efficient one for that algorithm.
- Determining chunk size is another challenge in the existing version of the HPX algorithms. Chunk size is the amount of work performed by each task [12, 13] that is determined by an *auto_partitioner* exposed by the HPX algorithms or is passed by using *static/dynamic_chunk_size* as an execution policy's parameter [10]. However,
  (1) the experimental results in [4] and [3] showed that the overheads of determining chunk size by using the *auto_partitioner* negatively effected the application's scalability in some cases;
  (2) the policy written by the user will often not be able to determine the optimum chunk size either due to the limit of runtime information.
- In [14], we proposed the HPX prefetching method which aids prefetching that not only reduces the memory accesses latency, but also relaxes the global barrier. Although it results in better parallel performance for the HPX algorithms, however, a distance between each two prefetching operations should also be manually chosen by a user for each new program.

Automating these mentioned parameters selections by considering loops characteristics implemented in a learning model can optimize the HPX parallel applications performances. To the best of our knowledge, we present the first attempt to implement a learning model for predicting optimum loop parameters at runtime, wherein the learning model captures features both from static compile time information and from runtime introspection.

In this research, we introduce a new ClangTool *ForEachCallHandler* using LibTooling [15] as a custom compiler pass to be executed by the Clang compiler, which is intended to collect the static features at compile time. The logistic regression model is implemented in this paper as a learning model that considers these captured features for predicting efficient parameters for an HPX loop. For implementing this learning model on a loop, we propose new HPX smart executors that – when used on a parallel loop – instructs the compiler to apply this *ForEachCallHandler* tool on that loop. As a results, the loop's features will automatically be included in the prediction process implemented with that learning model. One of the advantages of this approach in utilizing HPX policies is that in practice it enables us to change the algorithms internal structure at runtime and therefore we do not have to compile the code again after the code transformation step.

This technique is able to use high-level programming abstractions and machine learning to relieve the programmer of difficult and tedious decisions that can significantly affect program behavior and performance. Our results show that combining machine learning, compiler optimizations and runtime adaptation helps us to maximally utilize available resources. This improves application performance by around 12% – 35% for the Matrix Multiplication, Stream and 2*D* Stencil benchmarks compared to setting their HPX loop's execution policy/parameters manually or using HPX auto-parallelization techniques.

The remainder of this paper is structured as follows: the machine learning algorithms that are used to study the learning models are discussed in section 2; the proposed model is discussed is more details in section 3, and section 4 provides the experimental results of this proposed technique. Conclusion and future works are explained in section 5.

2 LEARNING ALGORITHM

In this research we use the binary and multinomial logistic regression models [16] to select the optimum execution policy, chunk size, and prefetching distance for certain HPX loops based on both, static and dynamic information, with the goal of minimizing execution time. Logistic regression model has been used in several previous works [8, 17], and it is shown to be able to predict such parameters accurately. We will show later that the performance of these learning models has high accuracy for about 98% and 95% for the binary and multinomial logistic regression models respectively on the studied problems. Also, compared to the other learning models such as artificial neural networks (ANNs), the implemented logistic regression model has lower computational complexity. Moreover, since the chunk size values can be seen as a categorical variables, this makes the logistic regression models well-suited for our problem.

Here, the static information about the loop body (such as the number of operations, see Table 1) collected by the compiler and the dynamic information (such as the number of cores used to execute the loop) as provided by the runtime system is used to feed a logistic regression model enabling a runtime decision to obtain highest possible performance of the loop under consideration. The presented method relies on a compiler-based source-to-source transformation. The compiler transforms certain loops which were annotated by the user by providing special executors – discussed later in section 3.1 – into code controlling runtime behavior. This transformed code instructs the runtime system to apply a logistic regression model and to select either an appropriate code path (e.g. parallel or sequential loop execution) or certain parameters for the loop execution itself (e.g. chunk size or prefetching distance). We briefly discuss these learning models in the following sections.

2.1 Binary Logistic Regression Model

In order to select the optimum execution policy (sequential or parallel) for a loop, the binary logistic regression model is implemented to analyze the static information extracted from the loop by the compiler and the dynamic information as provided by the runtime. In this model, the weights parameters having k features $W^T = [\omega_1, \omega_2, ..., \omega_k]$ are determined by considering features values $x_r(i)$ of each experiment $X_i = [1, x_1(i), ..., x_k(i)]^T$ which minimize the log-likelihood of the Bernoulli distribution value as follow:

$$\mu_i = 1/(1 + e^{-W^T X_i}). \quad (1)$$

The values of ω are updated in each step t as follows:

$$\omega_{t+1} = (X^T S_t X)^{-1} X^T (S_t X \omega_t + y - \mu_t) \quad (2)$$

, where S is a diagonal matrix with $S(i,i) = \mu_i(1-\mu_i)$. The output is determined by considering the following decision rule:

$$y(x) = 1 \longleftrightarrow p(y=1|x) > 0.5 \tag{3}$$

2.2 Multinomial Logistic Regression Model

In order to predict the optimum values for the chunk size and the prefetching distance, the multinomial logistic regression model is implemented to analyze the static information extracted from the loop by the compiler and the dynamic information as provided by the runtime. If we have N experiments that are classified in C classes and each has K features, the posterior probabilities are computed by using a softmax transformation of the features variables linear functions for an experiment n with a class c as follow:

$$y_{nc} = y_c(X_n) = \frac{exp(W_c^T X_n)}{\sum_{i=1}^{C} exp(W_i^T X_n)} \tag{4}$$

The cross entropy error function is defined as follows:

$$E(\omega_1, \omega_2, ..., \omega_C) = -\sum_{n=1}^{N}\sum_{c=1}^{C} t_{nc} ln y_{nc} \tag{5}$$

, where T is a $N \times C$ matrix of target variables with t_{nc} elements. The gradient of E is computed as follows:

$$\nabla_{\omega_c} E(\omega_1, \omega_2, ..., \omega_C) = \sum_{n=1}^{N} (y_{nc} - t_{nc}) X_n \tag{6}$$

In this method, we use the Newton-Raphson method [18] to update the weights values in each step:

$$\omega_{new} = \omega_{old} - H^{-1} \nabla E(\omega) \tag{7}$$

where H is the Hessian matrix defined as follows:

$$\nabla_{\omega_i} \nabla_{\omega_j} E(\omega_1, \omega_2, ..., \omega_C) = \sum_{n=1}^{N} y_{ni}(I_{ij} - y_{nj}) X_n X_n^T \tag{8}$$

More details can be found in [19].

3 PROPOSED MODEL

In this section, we propose a new technique for applying the learning models discussed in section 2 to HPX loops. The goal of this technique is to combine machine learning methods, compiler transformations, and runtime introspection in order to maximize the use of available resources and to minimize execution time of the loops. Its design and implementation has several steps categorized as follow*:

(1) New HPX Smart Executors
(2) Features Extraction
(3) Design of Learning Model
(4) Learning Model Implementation

*This technique with its installation instructions are publicly available at https://github.com/STEllAR-GROUP/hpxML. Feel free to join our IRC channel $\#ste||ar$ if you need any help.

```
for_each(par_if, range.begin(), range.end(), lambda);

for_each(policy.with(adaptive_chunk_size()),range.begin(),range.
    end(),lambda);

for_each(make_prefetcher_policy(policy,
        prefetching_distance_factor,
        container_1,..., container_n),
        range.begin(),range.end(),lambda);
```

Figure 1: Loops using the proposed smart executors, which are recognized and instrumented by the compiler to allow HPX to consider the weights produced by the learning models when executing the loops.

static/dynamic	Information
dynamic	number of threads*
dynamic	number of iterations*
static	number of total operations per iteration*
static	number of float operations per iteration*
static	number of comparison operations per iteration*
static	deepest loop level*
static	number of integer variables
static	number of float variables
static	number of if statements
static	number of if statements within inner loops
static	number of function calls
static	number of function calls within inner loops

Table 1: Collected static and dynamic features. First 6 features marked with red* have been selected for our model using the decision tree classification technique [20, 21] to avoid overfitting the model.

3.1 New HPX Smart Executors

We introduce two new HPX execution policies and one new HPX execution policy parameter, which we refer to them as the *smart* executors in this paper, since they enable the weights gathered by the learning model to be applied on the loop. *par_if* and *make_prefetcher_policy* as the smart policies instrument executors to be able to consume the weights produced by a binary logistic regression model, which is used to select the execution policy corresponding to the optimal code path to execute (sequential or parallel), and a multinomial logistic regression model, which is used to determine an efficient prefetching distance. *adaptive_chunk_size* as the smart execution policy parameter uses a multinomial logistic regression model to determine an efficient chunk size. Fig.1 shows three loops defined with these smart execution policies and parameter that apply a *lambda* function over a *range*. We have created a new special compiler pass for clang which recognizes these annotated loops and transform them into equivalent code which instructs the runtime to apply the described regression models.

3.2 Features Extraction

Initially, we selected 10 static features to be collected at compile time and 2 dynamic features to be determined at runtime to be evaluated by our learning model. These features are listed in Table 1. Although it may not be the best possible set, it is very similar

```
class ForEachCallHandler:
     public RecursiveASTVisitor<ForEachCallHandler> { ...
  //Visit every call expression
  bool VisitCallExpr(const CallExpr *call) { ...
    //check if a call is an HPX algorithm
    if (func_string.find("hpx::parallel") !=string::npos) {

      //Capturing lambda function from a loop
      const CXXMethodDecl* lambda_callop =
            lambda_record->getLambdaCallOperator();
      Stmt* lambda_body = lambda_callop->getBody();

      //Capturing policy
      SourceRange policy(call->getArg(0)->getExprLoc(),
       call->getArg(1)->getExprLoc().getLocWithOffset(-2));

      //Extracting static/dynamic features
      analyze_statement(lambda_body);

      //Determining policy
      if (policy_string.find("par_if") != string::npos)
         policy_determination(call, SM);

      //Determining chunk size
      if(policy_string.find("adaptive_chunk_size")!=string::npos)
         chunk_size_determination(call, SM);

      //Determining prefetching distance
      if(policy_string.find("make_prefetcher_policy")!=string::
           npos)
         prefetching_distance_determination(call, SM);
         ...
}}}
```

Figure 2: The proposed ClangTool *ForEachCallHandler* to collecting static/dynamic information of each loop and implement a learning model based on the current smart executors.

to those considered in the other works [1, 2, 8], which in their results indicated that the set is sufficient to design a learning model for this type of problem. To avoid overfitting the model, we chose 6 critical features marked with red* color in Table 1 to include in the actual decision tree classification technique[20, 21], which reduces the initial features set in a tree building process based on information gain value. This value is used to decide which feature to be selected for splitting data at each step in a tree building process. More information about this technique can be found in [22, 23].

In order to collect static information at compile time, we introduce a new ClangTool named *ForEachCallHandler* in the Clang compiler as shown in Fig.2. This tool locates in the user source code instances of loops which use the proposed smart executors. Once identified, the loop body is then extracted from the *lambda* function by applying *getBody()* on a *lambda* operator *getLambdaCallOperator()*. The value of each of the listed static features is then recorded by passing *lambda* to *analyze_statement*. In order to capture dynamic features at runtime, the compiler inserts hooks (HPX API function calls) which are invoked by the runtime. In this instance the compiler will insert the call *hpx::get_os_thread_count()* and *std::distance(range.begin(), range.end())* which will return the number of OS threads as well as the number of iterations that the loop will run over, respectively.

3.3 Designing the Learning Model

To design an efficient learning model that could be able to cover various cases, we collected over 300 training data sets by analyzing Matrix multiplication application with different problem sizes that implements *par_if*, *adaptive_chunk_size* or *make_prefetcher_policy*

```
for_each(par_if,range.begin(),range.end(),lambda);
```

(a) Before compilation

```
if(seq_par({f0,..fn})) //extracted static information
  for_each(seq, range.begin(),range.end(),lambda);
else
  for_each(par, range.begin(),range.end(),lambda);
```

(b) After compilation

Figure 3: The proposed function *seq_par* for a implementing binary logistic regression model at runtime.

on its loops. The experimental results evaluated in Section 4 show that these training data* are enough to predict the HPX loop's parameters accurately for the studied applications: Matrix multiplication, Stream and 2*D* Stencil benchmarks. The regression models are designed based on these collected data, in which the values of ω from eq.2 and eq.7 are determined whenever the sum of square errors reaches its minimum value. Then they are stored in an output file named as *weights.dat* that will be used for predicting the optimal execution policy, chunk size, and prefetching distance at runtime. This learning step can be done offline, which also doesn't add any overhead at compile time nor does at runtime.

It should be noted that the multinomial logistic regression model must be initialized with the allowed boundaries for the chunk size and prefetching distance in order to choose an efficient value. In this study we selected 0.1%, 1%, 10%, or 50% of the iterations of a loop as chunk size candidates and 1, 5, 10, 100 and 500 cache lines as prefetching distance candidates. These candidates are validated with different tests and based on their results, they are selected. In order to derive the fidelity of the model, we train the algorithm using 80% of the test cases and use the remaining 20% as a trials to see how accurate the predictions are. Our results show that the binary logistic regression model is accurate in 98% of the trials and the multinomial logistic regression model is accurate in 95% of them.

3.4 Learning Model Implementation

3.4.1 Binary Logistic Regression Model (Execution Policy). We propose a new function *seq_par* that passes the extracted features for a loop that uses *par_if* as its execution policy. In this technique, a Clang compiler automatically adds extra lines within a user's code as shown in Fig.3 that allows the runtime system to decide whether execute a loop sequentially or in parallel based on the return value of *seq_par* from Eq.3. If the output is *false* the loop will execute sequentially and if the output is *true* the loop will execute in parallel.

3.4.2 Multinomial Logistic Regression Model (Chunk Size). We propose a new function *chunk_size_determination* that passes the extracted features for a loop that uses *adaptive_chunk_size* as its execution policy's parameter. In this technique, a Clang compiler changes a user's code automatically as shown in Fig.4 that makes runtime system to choose an optimum chunk size based on the

*The characteristics of the loops of these training data are available at https://github.com/STEllAR-GROUP/hpxML/blob/master/logisticRegressionModel/algorithms/inputs.

```
for_each(policy.with(adaptive_chunk_size()),
        range.begin(),range.end(),lambda);
```

(a) Before compilation

```
for_each(policy.with(chunk_size_determination({f0,...fn}))),
        range.begin(),range.end(),lambda);
```

(b) After compilation

Figure 4: The proposed function *chunk_size_determination* for implementing a multinomial logistic regression model at runtime.

```
for_each(make_prefetcher_policy(policy,
         prefetching_distance_factor,
         container_1,..., container_n),
         range.begin(),range.end(),lambda);
```

(a) Before compilation

```
for_each(make_prefetcher_policy(policy,
         prefetching_distance_determination({f0,...fn}),
         container_1,..., container_n),
         range.begin(),range.end(),lambda);
```

(b) After compilation

Figure 5: The proposed function *prefetching_distance_determination* for implementing a multinomial logistic regression model at runtime.

output of *chunk_size_determination* from Eq.4 that is based on the chunk size candidate's probability and it is computed using the values of the studied loop's features and the learning model's weights.

3.4.3 Multinomial Logistic Regression Model (Prefetching Distance). We propose a new function *prefetching_distance_determination* that passes the extracted features for a loop that uses *make_prefetcher_policy* as its execution policy. In this technique, a Clang compiler changes a user's code automatically as shown in Fig.5 that makes runtime system to choose an optimum prefetching distance based on the output of *prefetching_distance_determination*. This function also computes the outputs by implementing Eq.4 using the values of the studied loop's features and the learning model's weights.

As we can see, these proposed techniques consider both, the static and the dynamic information for determining an efficient execution policy, chunk size, and prefetching distance for a loop. In addition, this decision process is performed at runtime by computing outputs of *seq_par*, *chunk_size_determination* and *prefetching_distance_determination*, which avoids an extra compilation step. In other words, static information is collected during compilation and the decisions aiming at optimum parameters are made at runtime while taking into account additional runtime information. One of the other advantages of this method are that other parameters and executors attached to the current executors can be also reattached to the generated execution policy. Moreover, all of these smart executors can be used together by simply defining a loop policy to be "*make_prefetcher_policy(par_if, ...).with(adaptive_chunk_size())*". The experimental results of our proposed learning techniques discussed are presented in the next section.

4 EXPERIMENTAL RESULTS

In this section, we evaluate the performance of our proposed technique using Clang 4.0.0 and HPX *V*0.9.99 on the test machine with two Intel Xeon E5-2630 processors, each with 8 cores clocked at 2.4*GHZ* and 65*GB* of main memory. The main goal here is to illustrate that dynamic information obtained at runtime and static information obtained at compile time are both necessary to provide sufficient parallel performance and the proposed techniques are able to predict the optimum parameters for HPX loops based on these information*.

4.1 Artificial Test Cases

In this section, we evaluate the performance of the proposed techniques from Section 3 over 5 different artificial test cases shown in Table 2, in which each of them includes 4 loops with different characteristics. Each of these loops of each test cases is a Matrix multiplication computation with different problem sizes included in this table. The main purpose of these evaluations is to show the effectiveness of each proposed method on an HPX parallel performance.

4.1.1 par_if. Parallelizing all loops within an application may not result in a best possible parallelization, as some of the loops cannot scale desirably on more number of threads. For evaluating the effectiveness of the proposed *seq_par* function exposed by a smart executor *par_if* discussed in section 3, we study its implementation on the described 5 test cases. These test cases are selected to show that in case of having several loops within a parallel application, some of these loops should be executed in sequential to achieve a better parallel performance. Each of these test cases is executed three times by setting execution policies of the outer loops to be *seq*, *par*, or *par_if* in each time. The static and dynamic characteristics of each loop in each test are listed in Table 2. The execution policies determined by using *par_if* policy for each loop are also included in the column *Policy* of this Table.

Fig.6 shows the execution time for each test case and it illustrates that in most of them using *par_if* will outperform the basic policy *par*. The main reason of this improvement is that by considering the determined execution policy included in Table 2, as the execution policy *seq* is determined for some of the loops that cannot scale desirably on more number of threads, this technique results in outperforming manually parallelized code by around 15% – 20% for these test cases expect the first one. In this test case, however, the total execution time of the loops took slightly longer when invoked with *par_if*. This is due to the overhead generated during the invocation of the binary logistic regression model's cost function, manually setting their execution policy as *par* resulted in having a better performance.

4.1.2 adaptive_chunk_size. As discussed in Section 3, the proposed *chunk_size_determination* function exposed by a smart executor *adaptive_chunk_size* enables the runtime system to choose an efficient chunk size for a loop by considering static and dynamic features of that loop. As mentioned in section 3.3, this method selects between chunk sizes of 0.1%, 1%, 10%, or 50% of the iterations of a

*Applications evaluated in this Section are publicly available at https://github.com/STEllAR-GROUP/hpxML/tree/master/examples.

Test	Loop	Iterations	Total opr.	Float opr.	Comparison opr.	Loop level	Policy (Threads)	Chunk size%	Pref. dist.
1	l_1	10000	400100	200000	101010	2	par (8)	0.1	5
	l_2	20000	450026	250000	150503	2	par (8)	0.1	5
	l_3	20000	502040	250000	103051	2	par (8)	0.1	1
	l_4	500	550402	200000	150102	1	par (8)	10	5
2	l_1	150000	350106	101010	500	2	par (8)	0.1	10
	l_2	100	10050016	5000000	2505013	3	seq	10	1
	l_3	100	25000000	3010204	1500204	3	seq	10	1
	l_4	50000	4000450	200000	100150	1	par (8)	1	5
3	l_1	500	4504030	250000	150300	2	par (8)	1	10
	l_2	400	3502020	200000	100405	1	par (8)	1	10
	l_3	2000	250033	150000	103040	3	seq	10	5
	l_4	2500	350400	150000	100600	3	seq	10	5
4	l_1	20000	204002	100000	10320	2	par (8)	0.1	1
	l_2	30000	400000	150102	10000	2	par (8)	0.1	1
	l_3	300	550000	44000	20030	3	seq	10	5
	l_4	400	450000	50400	10602	3	seq	10	10
5	l_1	200	4502001	150000	101004	3	par (8)	1	1
	l_2	700	400020	300000	150006	3	par (8)	1	5
	l_3	300	302020	20000	14005	2	par (8)	1	5
	l_4	100	50400	20000	10110	2	seq	10	10

Table 2: Execution policy, chunk size and prefetching distance determined by the proposed techniques based on the static/dynamic information extracted from each loop and the weights provided by the learning models.

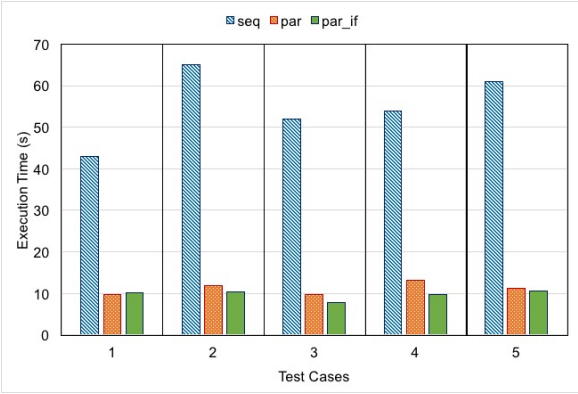

Figure 6: The execution time comparisons between setting execution policy of the loops to be *seq*, *par*, or *par_if*.

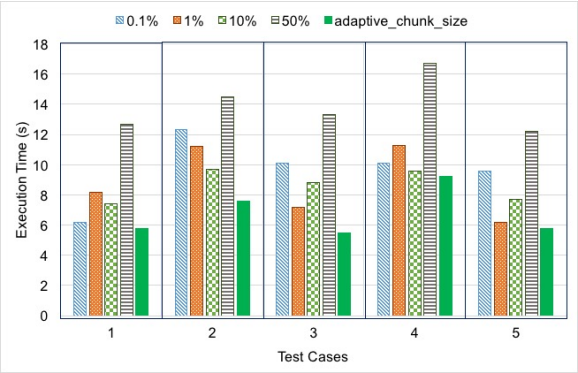

Figure 7: The execution time comparisons between setting chunk size of the loops to be 0.1%, 1%, 10%, or 50% of the iterations of a loop and the chunk size determined by using *adaptive_chunk_size*.

loop by comparing their probabilities in the multinomial logistic regression model's cost function.

Fig.7 shows the execution time for each test case in Table 2 by setting optimal chunk size of each loop. The chunk size determined by the algorithm for each loop are also included in the column *Chunk size%* of the Table 2. The overall performance of these cases show by an average of about 31%, 15%, 17% and 38% improvement over setting chunks to be 0.1%, 1%, 10%, or 50% of the iterations of a loop. The main reason of this improvement is that efficient chunk size helps in having even amount of work on each number of threads that results in reducing total overheads and latencies. These results also illustrate the importance of the chunk size's effect on an application's scalability and the capability of this method in improving parallel performance of an application by choosing efficient chunk size for each loop.

4.1.3 make_prefetcher_policy. As discussed in Section 3, the proposed *perfecting_distance_determination* function exposed by a smart executor *make_prefetcher_policy* allows the runtime system to choose an efficient prefetching distance for a loop by considering static and dynamic features of that loop. As it mentioned in Section 3.3, this method chooses between prefetching distances of 1, 5, 10, 100 and 500 cache lines by comparing their probabilities in the multinomial logistic regression model's cost function.

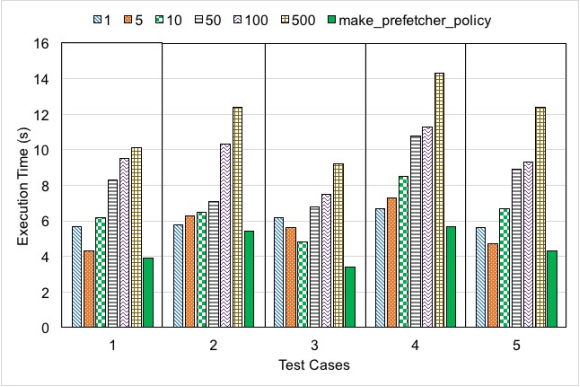

Figure 8: The execution time comparisons between setting the prefetching distance of the loops to be 1, 5, 10, 100, or 500 cache lines and the determined prefetching distance using *make_prefetcher_policy*.

Test	Iterations	Total opr.	Float opr.	Comp. opr.	level
Stream	50000000	8	8	0	0
Stencil	45	3502	2500	301	1

Table 3: Dynamic/static features for each benchmark.

```
for_each(policy, a_begin, a_end,[&](std::size_t i){
                c[i] = a[i]; // copy step
                b[i] = k * c[i]; // scale step
                c[i] = a[i] + b[i]; // adding step
                a[i] = b[i] + k * c[i]; // triad step
            });
```

Figure 9: Stream Benchmark.

Fig.8 shows the execution time for each prefetching size in each test case in Table 2. The prefetching distance determined by the algorithm for each loop are also included in the last column of the Table 2. The overall performance of these cases show by an average of about 25%, 19%, 14%, 33%, 24%, and 47% improvement over setting prefetching distances to be 1, 5, 10, 100, or 500 cache lines. The main reason of this improvement is that using efficient prefetching distance resulted in better cache usage that reduced the total overheads.

4.2 Real Benchmarks

In the previous section, we demonstrated the effectiveness of each proposed on an HPX parallel performance on 5 different test cases in which each of them includes 4 different loops for a matrix multiplication computation. In this section, we apply all of the proposed methods together on two different benchmarks: the Stream and 2*D* Stencil benchmarks. The previous training data is also used in the proposed techniques applied on these applications.

4.2.1 Stream Benchmark. This benchmark [24, 25] has been widely used for evaluating memory bandwidth of a system. In [10], the HPX executors performance were evaluated on this benchmark with 50 million data points over 10 iterations. The other characteristic information of this loop is included in Table 3. As shown in Fig.9,

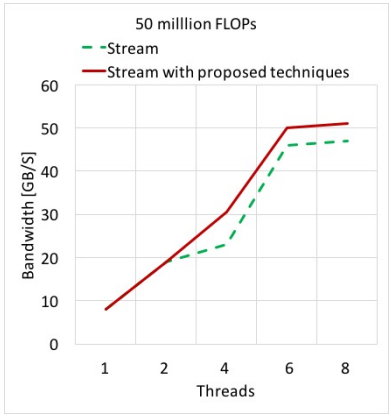

Figure 10: HPX Stream benchmark's strong scaling with/without using the proposed smart executors.

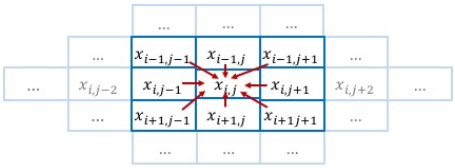

Figure 11: Heat Distribution Benchmark, 2*D* Stencil.

this application includes 4 operations over 3 equally sizes arrays (A, B and C) that are: copy ($C = A$), scale ($B = k \times C$), adding ($C = A + B$) and triad ($A = B + K \times C$). All three proposed smart executors are applied on this loop to make HPX to choose an execution policy, chunk size and prefetching distance efficiently. The speedup comparison results of the data transform measurements with/without using proposed techniques are illustrated in Fig.10. As we can see, using the proposed smart executors together on this benchmark improves HPX performance by an average of about 13% compared to using HPX auto-parallelization techniques without considering static/dynamic information and implementing machine learning technique.

4.2.2 Stencil Benchmark. The performance of different HPX scheduling policies on a 2*D* Stencil benchmark is studied in [11]. This application is a two dimensional heat distribution shown in Fig.11, in which the temperature of each point is computed based on the temperature of its neighbors. The characteristic information of this loop is included in Table 3. The speedup comparison results of HPX performance with/without using the proposed smart executors are illustrated in Fig.12. It shows HPX performance improvement by an average of about 22% by using the proposed techniques together on this loop compared to using HPX auto-parallelization techniques without considering static/dynamic information and implementing machine learning technique.

5 CONCLUSION AND FUTURE WORKS

The main goal of this paper is to illustrate a powerful new set of techniques that can be made available to application developers when compilers, runtime systems, and machine learning algorithms

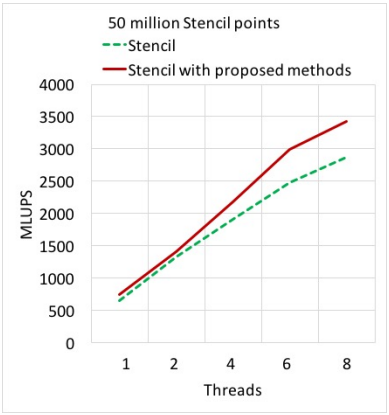

Figure 12: HPX 2D Stencil benchmark's strong scaling with/without using the proposed smart executors.

work in concert. These techniques developed here not only greatly improve performance, but users are able to reap their benefit with little cost to themselves. Simply by annotating their code with high level executors, users can see their application's performance increase in a portable way.

These results could have broad impact for applications and libraries as well as the maintainers and scientist that use them. The high level annotations increase the usability and therefore accessibility of runtime features that before would have taken a knowledgeable developer to implement. Due to the machine learning element, users will not have to worry about losing performance in different runtime environments that could manifest themselves. Finally, the inclusion of compiler information will allow these performance optimizations to be platform independent. These three features taken together present a notable solution to the challenges presented by an increasingly multi-core and heterogeneous world.

As powerful as these techniques may be, more work is needed to be done in order to fully realize the potential of this work. Notably, the breadth of performance characteristics needs to be more carefully studied to understand the core features that relate to performance. Additionally more research is needed to ensure that the characteristics measured here also are relevant for other architectures such as the new Knights Landing chipset. On a shorter time scale we intend to investigate extending the number of features for improving the resulting loop's parameters prediction.

In this paper, we have illustrated that the parallel performance of our test cases were improved by using a machine learning algorithm to determine either an appropriate code path (parallel or sequential loop execution) or certain parameters for the loop execution itself (chunk size or prefetching distance). The speedup results of these test cases and benchmarks showed by around 12% – 35% improvement compared to selecting execution policy, chunk size and prefetching distance of a loop without using static/dynamic information and machine learning technique. These results proved that combining machine learning techniques, compiler information, and runtime methods helps an application maximize the available resources*.

*This works was supported by NSF awards 1447831 and 1339782.

REFERENCES

[1] Mark Stephenson and Saman Amarasinghe. Predicting unroll factors using supervised classification. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 123–134. IEEE, 2005.

[2] Keith D Cooper, Devika Subramanian, and Linda Torczon. Adaptive optimizing compilers for the 21st century. *The Journal of Supercomputing*, 23(1):7–22, 2001.

[3] Zahra Khatami, Hartmut Kaiser, Patricia Grubel, Adrian Serio, and J Ramanujam. A massively parallel distributed n-body application implemented with hpx. In *Proceedings of the 7th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 57–64. IEEE Press, 2016.

[4] Zahra Khatami, Hartmut Kaiser, and J Ramanujam. Using hpx and op2 for improving parallel scaling performance of unstructured grid applications. In *Parallel Processing Workshops (ICPPW), 2016 45th International Conference on*, pages 190–199. IEEE, 2016.

[5] Lawrence Rauchwerger, Nancy M Amato, and David A Padua. A scalable method for run-time loop parallelization. *International Journal of Parallel Programming*, 23(6):537–576, 1995.

[6] Lawrence Rauchwerger, Nancy M Amato, and David A Padua. Run-time methods for parallelizing partially parallel loops. In *Proceedings of the 9th international conference on Supercomputing*, pages 137–146. ACM, 1995.

[7] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, James Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 19(1):188–222, 1997.

[8] Gennady Pekhimenko and Angela Demke Brown. Efficient program compilation through machine learning techniques. In *Software Automatic Tuning*, pages 335–351. Springer, 2011.

[9] John Cavazos and Michael FP O'boyle. Method-specific dynamic compilation using logistic regression. *ACM SIGPLAN Notices*, 41(10):229–240, 2006.

[10] Hartmut Kaiser, Thomas Heller, Daniel Bourgeois, and Dietmar Fey. Higher-level parallelization for local and distributed asynchronous task-based programming. In *Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware*, pages 29–37. ACM, 2015.

[11] Rekha Raj. Performance analysis with hpx. Master's thesis, Louisiana State University, 2014.

[12] Henry C Baker Jr and Carl Hewitt. The incremental garbage collection of processes. In *ACM Sigplan Notices*, volume 12, pages 55–59. ACM, 1977.

[13] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, and Dietmar Fey. 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*, page 6. ACM, 2014.

[14] Zahra Khatami, Hartmut Kaiser, and J Ramanujam. Redesigning op2 compiler to use hpx runtime asynchronous techniques. In *Parallel and Distributed Scientific and Engineering Computing (IPDPSW), 2017 18th IEEE International Workshop on*. IEEE, 2017.

[15] The Clang Team. Clang 5 documentation, LibTooling. https://clang.llvm.org/docs/LibTooling.html, 2017. [Online; accessed 1-Feb-2017].

[16] C Bishop. Pattern recognition and machine learning (information science and statistics), 1st edn. 2006. corr. 2nd printing edn. *Springer, New York*, 2007.

[17] Method-specific dynamic compilation using logistic regression.

[18] Tjalling J Ypma. Historical development of the newton–raphson method. *SIAM review*, 37(4):531–551, 1995.

[19] Ian Nabney. *NETLAB: algorithms for pattern recognition*. Springer Science & Business Media, 2002.

[20] Chotirat Ann Ratanamahatana and Dimitrios Gunopulos. Scaling up the naive bayesian classifier: Using decision trees for feature selection. 2002.

[21] Wei-Yin Loh. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23, 2011.

[22] Lior Rokach and Oded Maimon. *Data mining with decision trees: theory and applications*. World scientific, 2014.

[23] Wei-Yin Loh. Classification and regression trees. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 1(1):14–23, 2011.

[24] JD McCalpin. Stream: Sustainable memory bandwidth in high performance computers (2008), 1991-2007.

[25] JD McCalpin. Memory bandwidth and machine balance in current high performance computers. In *Committee on Computer Architecture (TCCA) Newsletter*, pages 19–25. IEEE, 1995.