Abstract—The performance of many parallel applications depend on the loop-level parallelism. However, manually parallelizing all loops may result in degrading parallelization performance, as some of the loops cannot scale desirably on more number of threads. In addition, the overheads of manually setting chunk sizes might avoid an application to reach 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 for determining execution policy and chunk size of a loop within an application by considering those captured information implemented within our learning model. Our evaluated execution results show that the proposed technique can speed up the execution process up to 45%.

I. INTRODUCTION

Runtime information is often speculative, solely relying on it doesn’t guarantee maximizing parallelization performance, since the parallelization performance of an application depends on both the values measured at runtime and the related transformations performed at compile time. Collecting outcome of the static analysis performed by the compiler could significantly improve the runtime performance. These captured information should be analyzed to optimize the application’s parameters for achieving maximum parallelization. However, manually tuning parameters becomes ineffective and almost impossible when too many features are given to the program. Hence, many researches have extensively studied machine learning algorithms to optimize such parameters automatically.

For example 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 [2], clustering algorithm is implemented for examining different benchmarks for their similarities to reduce the time needed for evaluating other similar benchmarks and estimating their performances. In [3], neural network and decision tree are applied on the training data collected from different observations to predict the branch behavior in a new program.

Most of these existing 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. Also, none of them considers both static and dynamic information. The goal of this research is to optimize an HPX performance by predicting optimum execution policy and efficient chunk size for its parallel algorithms by considering both static and dynamic information and to develop a technique to avoid unnecessary compilation. To the best of our knowledge, we present a first attempt in implementing learning model for the loop parameters prediction at runtime, in which designing these runtime techniques and capturing learning models features are automatically performed at compile time.

II. LEARNING ALGORITHM

A. Binary Logistic Regression Model

For predicting optimum execution policy (sequential or parallel), we implement a binary logistic regression model [4] for analyzing extracted information from a loop. The weights parameters $W^T = [\omega_0, \omega_1, \omega_2, \ldots]$ are determined by considering features values $x_r(i)$ of each experiment $X(i) = [1, x_1(i), x_2(i), \ldots]^T$ for minimizing log-likelihood of the Bernoulli distribution value $\mu(i) = 1/(1 + e^{-W^T x(i)})$. The values of $\omega$ are updated as follow:

$$\omega_{k+1} = (X^T S_k X)^{-1} X^T (S_k X \omega_k + y - \mu_k) \quad (1)$$

In equation (1), $S$ is a diagonal matrix with $S(i, i) = \mu(i)(1 - \mu(i))$. The output is determined by considering decision rule as follow:

$$y(x) = 1 \iff p(y = 1|x) > 0.5 \quad (2)$$

B. Multinomial Logistic Regression Model

For predicting optimum chunk size, we implement a multinomial logistic regression model [4] for analyzing extracted information from a loop. The posterior probabilities are computed by using softmax transformation of the feature variables linear functions as follow:

$$y_{nk} = y_k(\phi_n) = \frac{\exp(W_n^T \phi(X_n))}{\sum_j \exp(W_j^T \phi(X_n))} \quad (3)$$

The cross entropy error function is defined as follow:

$$E(\omega_1, \omega_2, \ldots, \omega_k) = -\sum_n \sum_k t_{nk}\ln y_{nk} \quad (4)$$

The gradient of $E$ is computed as follow:

$$\nabla \omega_j E(\omega_1, \omega_2, \ldots, \omega_k) = \sum_n (t_{nj} - y_{nj}) \phi(x_n) \quad (5)$$

We use the Newton-Raphson for updating the weights values:

$$\omega_{new} = \omega_{old} - H^{-1} \nabla E(\omega) \quad (6)$$

, where $H$ is the Hessian matrix defined as follow:

$$\nabla \omega_k \nabla \omega_j E(\omega_1, \omega_2, \ldots, \omega_k) = \sum_n y_{nk} (I_{nj} - y_{nj}) \phi(x_n) \phi^T(x_n) \quad (7)$$
class ForEachCallHandler::public MatchFinder::MatchCallback{
  virtual void run(const MatchFinder::MatchResult &Result) {
    ...
    const SourceManager *SM = Result.Manager;
    // Capturing lambda function from a loop
    const CXXMethodDecl *lambda_callop = lambda_record->getLambdaCallOperator();
    StringRef lambda_body = lambda_callop->getBody();
    // Capturing policy
    SourceRange policy = call->getArg(0)->getExprLoc() ->
    call->getArg(1)->getExprLoc(), getLocWithOffset(-2));
    std::string policy_string = Lexer::getSourcetext(CharSourceRange::getSourceRange(policy));
    // Determining chunk size if a current policy's parameter is adaptive_chunk_size
    if (policy_string.find("adaptive_chunk_size") != string::npos) {
      // Extracting static information from lambda function
      analyze_statement(lambda_body);
      chunk_size_determination(call, SM);
    } else {
      // Determining chunk size at runtime.
      chunk_size_determination(call, SM);
    }
  }
};

void run(const MatchFinder::MatchResult &Result) {
  ...
  const SourceManager *SM = Result.Manager;
  // Capturing lambda function from a loop
  const CXXMethodDecl *lambda_callop = lambda_record->getLambdaCallOperator();
  StringRef lambda_body = lambda_callop->getBody();
  // Capturing policy
  SourceRange policy = call->getArg(0)->getExprLoc() ->
  call->getArg(1)->getExprLoc(), getLocWithOffset(-2));
  std::string policy_string = Lexer::getSourcetext(CharSourceRange::getSourceRange(policy));
  // Determining chunk size if a current policy's parameter is adaptive_chunk_size
  if (policy_string.find("adaptive_chunk_size") != string::npos) {
    // Extracting static information from lambda function
    analyze_statement(lambda_body);
    chunk_size_determination(call, SM);
  }
};

Figure 1: Before compilation.

<table>
<thead>
<tr>
<th>static/dynamic</th>
<th>Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>dynamic</td>
<td>number of threads+</td>
</tr>
<tr>
<td>dynamic</td>
<td>number of iterations+</td>
</tr>
<tr>
<td>static</td>
<td>number of total operations+</td>
</tr>
<tr>
<td>static</td>
<td>number of float variables+</td>
</tr>
<tr>
<td>static</td>
<td>number of comparison operations+</td>
</tr>
<tr>
<td>static</td>
<td>deepest loop level+</td>
</tr>
<tr>
<td>static</td>
<td>number of integer variables</td>
</tr>
<tr>
<td>static</td>
<td>number of float variables</td>
</tr>
<tr>
<td>static</td>
<td>number of if statements</td>
</tr>
<tr>
<td>static</td>
<td>number of function calls within inner loops</td>
</tr>
<tr>
<td>static</td>
<td>number of function calls</td>
</tr>
</tbody>
</table>

Table 1: Collected static and dynamic features.

B. Feature Extraction

We collect 10 static features at compile time and 2 dynamic features at runtime to determine a learning model that are listed in Table 1. Although it may not be the best possible set, but it is very similar to those considered in the other works [1], [5], in which their results proved that set is sufficient to design a learning model. The first two features are measured dynamically at runtime and the rest of features are collected at compile time. For this purpose, we introduce a new class named ForEachCallHandler in the Clang compiler as shown in fig.2 that is intended to collect static information at compile time for the loops that use par_if as their execution policy or adaptive_chunk_size as their execution policy parameter. Each feature has a member in that class and they are calculated for each detected loop. These features are extracted from lambda function of the loop by applying getBody() on a lambda operator getLambdaCallOperator(). Then, the value of each of them are recorded by passing lambda to analyze_statement. Dynamic features are also measured by implementing std::get_os_thread_count() and std::distance(range.begin(), range.end()).

For avoiding overfitting problem, we choose 5 critical features marked with red+ color in Table 1 by implementing Principal Component Analysis Algorithm [4].

C. Learning Model Implementation

1) Implementing binary logistic regression model for determining efficient execution policy: A new function seq_par is proposed to pass the extracted features for the loops that use par_if as their execution policy. In this technique, the compiler adds extra lines within a user’s code automatically as shown in fig.3a that makes runtime to decide whether execute a loop sequentially or parallel based on the output of seq_par from eq.2, in which the output 0 results in executing loop sequentially and the output 1 results in executing loop in parallel. The input of this function includes the extracted static information that is initialized during compilation. Number of threads and number of iterations are also measured and included in that features

for_each(par_if, range1.begin(), range1.end(), lambda1);
for_each(policy, with(adaptive_chunk_size), range2.begin(), range2.end(), lambda2);

Figure 3: The proposed seq_par.

set at runtime. Fig.3b shows the policy determination approach implemented within seq_par for computing cost function by considering features and weights.

2) Implementing multinomial logistic regression model for determining efficient chunk size: A new function chunk_size_determination is proposed to pass 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.4a that makes runtime to choose an optimum chunk size by considering the output of chunk_size_determination from eq.3, that is based on the chunk size candidate’s probability. In addition to the extracted compile time static information, number of threads and number of iterations are also automatically measured and included in this function at runtime. Fig.4b shows the chunk size determination approach implemented within chunk_size_determination for computing cost function by considering features and weights values.

for_each(policy, with(chunk_size_determination((f0, f1, ..., fn)), range2.begin(), range2.end(), lambda2));

Figure 4: The proposed chunk_size_determination.
Table II: Execution policy and chunk size determined by seq_par and chunk_size_determination implementation. The values of the fields marked with * are divided by $10^5$ because of the limited space.

<table>
<thead>
<tr>
<th>test</th>
<th>loop</th>
<th>*l.</th>
<th>*l.o.</th>
<th>*l.t.o.</th>
<th>*l.c.</th>
<th>*l.c.o.</th>
<th>lvl</th>
<th>policy</th>
<th>chunk size</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$l_1$</td>
<td>10</td>
<td>400</td>
<td>200</td>
<td>101</td>
<td>2</td>
<td>par</td>
<td>0.001</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_2$</td>
<td>20</td>
<td>450</td>
<td>250</td>
<td>150</td>
<td>2</td>
<td>par</td>
<td>0.000</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_3$</td>
<td>20</td>
<td>502</td>
<td>250</td>
<td>103</td>
<td>2</td>
<td>par</td>
<td>0.000</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_4$</td>
<td>0.5</td>
<td>550</td>
<td>200</td>
<td>150</td>
<td>1</td>
<td>par</td>
<td>0.1</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>$l_1$</td>
<td>150</td>
<td>350</td>
<td>101</td>
<td>0.5</td>
<td>2</td>
<td>par</td>
<td>0.001</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_2$</td>
<td>0.1</td>
<td>10500</td>
<td>5000</td>
<td>2505</td>
<td>3</td>
<td>seq</td>
<td>0.1</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_3$</td>
<td>0.1</td>
<td>2500</td>
<td>3010</td>
<td>1500</td>
<td>3</td>
<td>seq</td>
<td>0.1</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_4$</td>
<td>50</td>
<td>4000</td>
<td>200</td>
<td>100</td>
<td>1</td>
<td>par</td>
<td>0.001</td>
<td>1.000</td>
</tr>
<tr>
<td>3</td>
<td>$l_1$</td>
<td>0.5</td>
<td>4504</td>
<td>250</td>
<td>150</td>
<td>2</td>
<td>par</td>
<td>0.001</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_2$</td>
<td>0.4</td>
<td>3502</td>
<td>200</td>
<td>100</td>
<td>2</td>
<td>par</td>
<td>0.001</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_3$</td>
<td>2</td>
<td>250</td>
<td>150</td>
<td>103</td>
<td>3</td>
<td>seq</td>
<td>0.1</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_4$</td>
<td>2.5</td>
<td>350</td>
<td>150</td>
<td>100</td>
<td>3</td>
<td>seq</td>
<td>0.1</td>
<td>1.000</td>
</tr>
<tr>
<td>4</td>
<td>$l_1$</td>
<td>20</td>
<td>204</td>
<td>100</td>
<td>10</td>
<td>2</td>
<td>par</td>
<td>0.001</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_2$</td>
<td>30</td>
<td>400</td>
<td>150</td>
<td>10</td>
<td>2</td>
<td>par</td>
<td>0.001</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_3$</td>
<td>0.3</td>
<td>550</td>
<td>44</td>
<td>20</td>
<td>3</td>
<td>seq</td>
<td>0.1</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_4$</td>
<td>0.4</td>
<td>450</td>
<td>50</td>
<td>10</td>
<td>3</td>
<td>seq</td>
<td>0.1</td>
<td>1.000</td>
</tr>
<tr>
<td>5</td>
<td>$l_1$</td>
<td>0.2</td>
<td>4502</td>
<td>150</td>
<td>101</td>
<td>3</td>
<td>par</td>
<td>0.001</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_2$</td>
<td>0.7</td>
<td>400</td>
<td>300</td>
<td>150</td>
<td>3</td>
<td>par</td>
<td>0.001</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_3$</td>
<td>0.3</td>
<td>302</td>
<td>20</td>
<td>14</td>
<td>2</td>
<td>par</td>
<td>0.001</td>
<td>1.000</td>
</tr>
<tr>
<td></td>
<td>$l_4$</td>
<td>0.1</td>
<td>50</td>
<td>20</td>
<td>10</td>
<td>2</td>
<td>seq</td>
<td>1</td>
<td>1.000</td>
</tr>
</tbody>
</table>

IV. EXPERIMENTAL RESULTS

In this section, we evaluate the performance of the proposed techniques over different test cases with different characteristics shown in Table II, using Clang 4.0.0 and HPX 0.9.99 and on the test machine with two Intel Xeon E5-2630 processors, each with 8 cores clocked at 2.4GHZ and 65GB.

1) seq_par: This function is able to make runtime to decide whether execute a loop sequentially or in parallel by considering static and dynamic features of that loop. Fig.5a shows the execution time for tests with 4 loops per each in Table II by choosing seq or par as an execution policy of all of its loops and implementing this proposed technique for choosing execution policy of those loops. Their determined final execution policies are included in Table II. Fig.5a illustrates that as the execution policy of all of the four loops of the first test case is determined as par by implementing this technique, due to the overhead of the policy_costs_func cost function, manually setting their execution policy as par resulted in having a better performance. However for the rest of the test cases, it illustrates that execution policy seq is determined for some of the loops that cannot scale desirably on more number of threads, which results in outperforming manually parallelized code by around 15% ~ 20%.

2) chunk_size_determination: This function is able to make runtime to choose an efficient chunk size for a loop by considering static and dynamic features of that loop. It should be noted that the multinomial logistic regression model requires to know the chunk size candidates for choosing efficient one among them, which are chosen to be 0.001, 0.01, 0.1, and 0.5 of the number of iteration of a loop in this research. Fig.5b shows the execution time for tests with 4 loops per each in Table II by setting chunk size of all of its loops to be one of the candidates and determining efficient one using this proposed technique. Their determined chunk size are included in the last column of the Table II. The overall performance of these cases show up to 45%, 32%, 37% and 58% improvement over setting chunks to be 0.001, 0.01, 0.1, or 0.5 iterations.

V. CONCLUSION AND FUTURE WORKS

In this paper, we developed new techniques that are able to implement the binary and multinomial logistic regression model to determine an optimum execution policy and chunk size for an HPX loop. These techniques are able to consider both static and dynamic features of a loop and to implement a learning technique at runtime to make an optimum decision for its execution without requiring extra compilation. We illustrated that the parallel performance of our test cases were improved by around 15% ~ 45% using our proposed technique. These results proved that combining machine learning technique, compiler and runtime methods helps in utilizing maximum resource availability for optimizing HPX parallel performance.

REFERENCES


