Improving the Parallel Performance of an NBody Application Using Adaptive Techniques in HPX

Zahra Khatami[1,2], Hartmut Kaiser[1,2], and J. Ramanujam[1]
[1]Center for Computation and Technology, Louisiana State University, Louisiana, USA
[2]The STE||AR Group, http://stellar-group.org
zahra.k.khatami@oracle.com, hkaiser@cct.lsu.edu, jxr@ece.lsu.edu

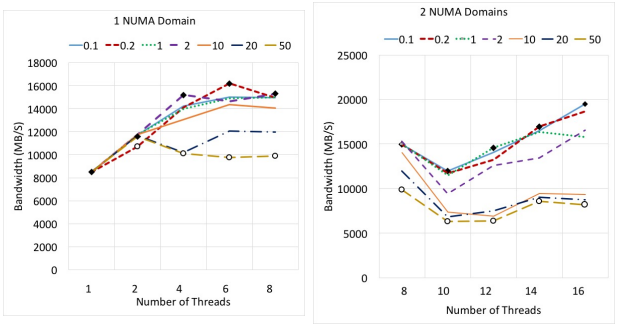
***Abstract*—The overheads of manually tuning a loop's parameters, such as chunk size, might prevent an application from reaching its maximum parallel performance. In this paper, we address this challenge by implementing a multinomial logistic regression model on an HPX loop. We present a framework that captures both the static and dynamic information of the runtime environment and feeds this information to a learning model which assigns an efficient loop chunk size automatically. Our evaluated execution results show that the proposed technique improves the performance of an NBody application by an average about 33%, 17%, and 19% compared to using existing HPX auto-parallelization tools when run with problem sizes of 10^5, 10^6, and 10^7 particles respectively.**

I. Introduction

While HPX runtime methods have been shown to be very effective in achieving improved scalability compared to the existing parallel techniques, (especially for highly dynamic applications such as NBody) they do not 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. We believe that collecting the outcome of the static analysis performed by the compiler could significantly improve HPX runtime decisions and therefore the application's performance. For example, determining chunk size is still one of the biggest challenges in the existing version of the HPX algorithms. This parameter can only be determined efficiently when the decision considers both static and dynamic information about the runtime environment. However, currently it is determined by either an *auto_partitioner* exposed by the HPX algorithms at runtime or by assigning a static/dynamic chunk size manually through an execution policy's parameter.

We have proposed a new technique that automates the chunk size selection by implementing a learning model which considers the loop's environment characteristics. In this technique, a new ClangTool *ForEachCallHandler*, implemented using LibTooling, is introduced as a custom compiler pass. This feature collects the static features of the system at compile time and stores them for future use. A multinomial logistic regression model then considers these captured static features in addition to dynamic features captured at runtime to predict an efficient chunk size for an HPX loop. In order to instruct the compiler to apply this method on a particular parallel loop, we propose a new execution policy parameter. This 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.

We show that this technique improves an NBody application's parallel performance by an average about 33%, 17%, and 19% for problem sizes of 10^5, 10^6, and 10^7 particles respectively compared to using existing auto-tuning parameters.



(a) (b)

Figure 1: NBody parallel performance with 10^6 particles using different chunk sizes.

To the best of our knowledge, we present the first attempt to implement a learning model for predicting an optimum chunk size at runtime, wherein the learning model captures features both from static compile time information and from runtime introspection.

II. Motivation

In HPX algorithms, the chunk sizes of an HPX loop is determined by either an *auto-partitioner* exposed by the HPX algorithm or a user defined policy. To demonstrate the importance of this effect on an application's scalability, we show the performance of an NBody application with 10^6 particles when assigning different chunk sizes in Fig.1. The chunk sizes include: 0.1%, 0.2%, 1%, 2%, 10%, 20%, or 50% of the particles. For each series on the graph, we marked the highest value with a "◆" and the lowest value with a "○". As illustrated, the most performant setting was to use 2 NUMA Domains and 16 threads with chunk size = 0.1% of the particles, which is approximately a 53% percentage difference over the worst case with chunk size = 50% of the particles.

III. Proposed Model

The implementation of our proposed model has several stages categorized as follow:

A. Special Execution Parameter

We introduce a new HPX execution policy parameter that enables the weights gathered by the learning model to be applied on the loop: *adaptive_chunk_size*. This parameter uses a multinomial logistic regression model to determine an efficient chunk size. Fig.2 shows a loop defined with this new execution parameter that applies a *computing_forces_on_each_particle* lambda function on each particle in each *time* step. We have introduced a new special compiler pass for clang that recognizes these annotated loops and transform them into equivalent code which instructs the runtime to apply the regression model.

```
for_each(policy.with(adaptive_chunk_size()),
         time.begin(),time.end(),
         computing_forces_on_each_particle);
```

Figure 2: A loop using the new execution parameter.

Type	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 I: Collected static and dynamic features.

B. Feature Extraction

Initially, we selected 12 static/dynamic features shown in Table I to be collected and evaluated by our learning model. To avoid overfitting the model, we chose 6 critical features shown in red∗ in Table I by using decision tree classification technique. To collect static information at compile time, we introduce a new ClangTool, *ForEachCallHandler*, in the Clang compiler shown in Fig.3. This tool locates in the user source code instances of loops which use *adaptive_chunk_size* as an execution policy parameter. Once identified by this ClangTool, the value of each of the listed static features is then analyzed by passing *lambda* to *analyze_statement*. For dynamic features, the compiler inserts HPX API function calls that are invoked at runtime. So 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 at runtime, respectively.

C. Learning Model Implementation

In order to pass the extracted features of a loop to the learning model cost function, we created a new function *chunk_size_determination*. Fig.4 shows how a Clang compiler automatically changes a user's code to enable a runtime system to choose an optimum chunk size based on the output of *chunk_size_determination*. This decision process is performed at runtime which avoids an extra compilation step.

```
bool VisitCallExpr(const CallExpr *call){
  if (func_string.find("hpx::parallel::") !=string::npos)
    if(policy_string.find("adaptive_chunk_size")!=string::
        npos)
      analyze_statement(lambda_body);
      chunk_size_determination(call, SM);
```

Figure 3: The proposed ClangTool *ForEachCallHandler*.

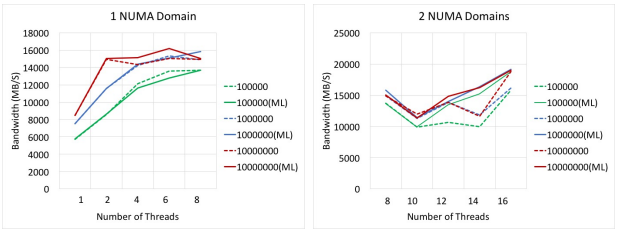
```
for_each(policy.with(adaptive_chunk_size()),
         time.begin(),time.end(),
         computing_forces_on_each_particle);
```

(a) Before compilation

```
for_each(policy.with(
         chunk_size_determination({f0,...fn}, weights))),
         time.begin(),time.end(),
         computing_forces_on_each_particle);
```

(b) After compilation; $\{f_0, ...f_n\}$ is the list of the extracted static/dynamic features and *weights* are the learning model weights values.

Figure 4: The proposed function *chunk_size_determination*.



(a) (b)

Figure 5: NBody parallel performance when assigning chunk sizes with auto_partitioner versus using the proposed technique.

IV. Experimental Results

In this section, we evaluate the performance of our proposed technique using Clang 4.0.0 and HPX $V0.9.99$ on a test machine with two Intel Xeon E5-2630 processors, each with 8 cores clocked at $2.4GHZ$ and $65GB$ of main memory. Fig.5a and 5b show the bandwidth rates for an NBody application with different problem sizes on 1 NUMA and 2 NUMA domains respectively. These results compare assigning chunk sizes automatically by using the HPX facility *auto_partitioner* and using the proposed technique discussed in Section III by annotating the loop with *adaptive_chunk_size*. In this case, the application gained up to 33%, 17%, and 19% improvement with application sizes of 10^5, 10^6, and 10^7 particles respectively. 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.

V. Conclusion

- ✓ The technique proposed in this research enables HPX parallel loops to consider both static and dynamic information when choosing their chunk sizes.
- ✓ This technique is implemented as a new execution policy, which enables the HPX to assign an efficient chunk size at runtime and avoids the need to compile the code a second time after the code transformation.
- ✓ Users can increase their application's performance simply by annotating their code with a high level execution policy parameters.