Applying Logistic Regression Model on HPX Parallel Loops

Zahra Khatami

Lukas Troska

Hartmut Kaiser

J. Ramanujam

Louisiana State University
The STE||AR Group, http://stellar-group.org

15th Charm++ Workshop

Outline

Motivation

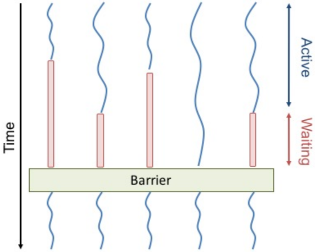
- Loop-level parallelism.
  1. Some of the loops cannot scale desirably to a large number of threads.
  2. Overheads of manually tuning loop parameters.
- Considering both dynamic runtime and static compile time information to achieve maximal parallel performance.

HPX[1]

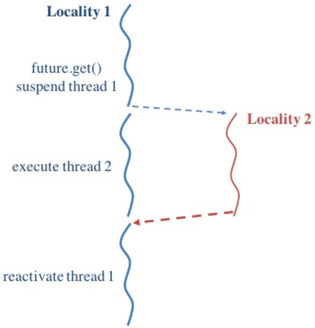
- ✓ Parallel C++ runtime system.
- ✓ Enabling fine-grained task parallelism: Resulting in a better load balancing.
- ✓ Providing efficient scalable parallelism.
- ✓ Reducing SLOW factors:
  1. **S**tarvation,
  2. **L**atencies,
  3. **O**verhead,
  4. **W**aiting.

[1] Kaiser, Hartmut, et al. "Hpx: A task based programming model in a global address space." Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models. ACM, 2014.

HPX


Time
Active
Waiting
Barrier

(a)


Locality 1
future.get()
suspend thread 1
Locality 2
execute thread 2
reactivate thread 1

(b)

HPX Current Challenges

Policy	Description	Implemented by
seq	sequential execution	Parallelism TS, HPX
par	parallel execution	Parallelism TS, HPX
par_vec	parallel and vectorized execution	Parallelism TS
seq(task)	sequential and asynchronous execution	HPX
par(task)	parallel and asynchronous execution	HPX

execution_policy: specifying execution restrictions of the work items:

- sequential execution policy: run sequentially.
- parallel execution policy: run in parallel.

Problem: Manually selecting execution policies for executing HPX parallel algorithms[1].

[1] H. Kaiser, T. Heller, D. Bourgeois, and D. Fey. "Higher-level parallelization for local and distributed asynchronous taskbased programming." In Proceedings of the First International Workshop on Extreme Scale Programming Models and Middleware, pages 29–37. ACM, 2015..

HPX Current Challenges

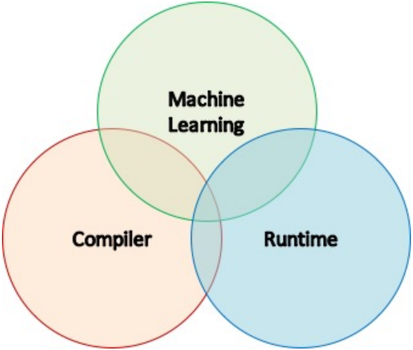
- chunk_sizes: Overheads of determining chunk size[1]:
  1. *auto_partitioner*: exposed by the HPX algorithms.
  2. *static/dynamic chunk*: execution policy's parameter.

[1] Z. Khatami, H. 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.

Solution

✓Automating parameters selections by considering loops characteristics implemented in a learning model.

Our Goal



✓Combining machine learning technique, compiler and runtime methods for utilizing maximum resource availability.

Proposed Method [1]

1. Designing Learning Model
2. Special Execution Policy
3. Feature Extraction: Collecting static and dynamic features
4. Learning Model Implementation

[1] Z. Khatami, L. Troska, H. Kaiser, and J. Ramanujam, "Applying Machine Learning Techniques on HPX Parallel Algorithms," in proceeding IPDPS PhD Forum, 2017.

Designing Learning Model

✓Logistic regression models [1]

- execution_policy: Binary logistic regression model.
- chunk_sizes: Multinomial logistic regression model.

[1] https://github.com/STEllARGROUP/hpxML/LearningAlgorithm

Binary Logistic Regression Model

- Output = Sequential or parallel

Updating weights: $W^T = [\omega_0, \omega_1, \omega_2, ...]$

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

Experiments: $X(i) = [1, x_1(i), x_2(i), ...]^T$

$S(i, i) = \mu(i)(1 - \mu(i))$

Bernoulli distribution value: $\mu(i) = 1/(1 + e^{-W^T x(i)})$

Decision rule: $y(x) = 1 \longleftrightarrow p(y = 1|x) > 0.5$

Multinomial Logistic Regression Model

- Output = Efficient chunk size → 0.001, 0.01, 0.1, and 0.5 of the loop's iteration.

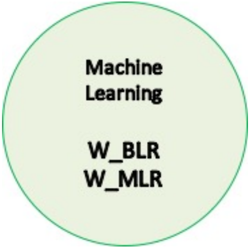
Updating weights: $\omega_{new} = \omega_{old} - H^{-1}\nabla E(\omega)$

Cross entropy error function:
$E(\omega_1, \omega_2, ..., \omega_C) = -\sum_{n=1}^{N}\sum_{c=1}^{C} t_{nc} ln y_{nc}$

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

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

✓Machine Learning



Special Execution Policy & Parameter

✓ Applying it on a loop makes implementing learning model on that loop.

- execution_policy → *par_if* (execution policy).
- chunk_sizes → *adaptive_chunk_size()* (execution policy's parameter).

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

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

Feature Extraction & Selection

✓ Introducing new ClangTool named *ForEachCallHandler*.

```
virtual void run(const MatchFinder::MatchResult &Result){
  ...
  if (policy_string.find("par_if") != string::npos ||
       policy_string.find("adaptive_chunk_size")!=string::
      npos){
       extract_features(lambda_body);
       ...
  }
}
```

Feature Extraction[1]

Type	Information
dynamic	number of threads
dynamic	number of iterations
static	number of total operations
static	number of float operations
static	number of comparison operations
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

[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.

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

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

Feature Selection

Type	Information
dynamic	number of threads*
dynamic	number of iterations*
static	number of total operations*
static	number of float operations*
static	number of comparison operations*
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

* Features selected with implementing decision tree classification technique[1].

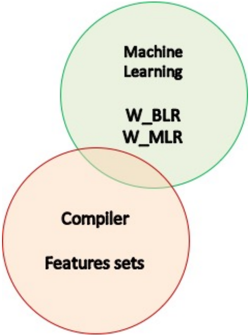
[1] Loh, Wei-Yin. "Classification and regression trees." Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 1.1 (2011): 14-23.

Learning Model Implementation

✓ *seq_par* & *chunk_size_determination*: making runtime choosing loop's parameters by considering static and dynamic features in costs_fnc cost function.

```
bool seq_par(F &&features)
{
  return costs_fnc(features, retrieving_BLR_weights());
}

dynamic_chunk_size chunk_size_determination(F &&features)
{
  return costs_fnc(features, retrieving_MLR_weights());
}
```

✓Machine Learning & Compiler



Learning Model Implementation

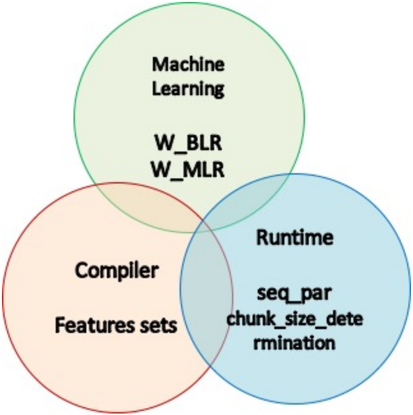
Before compilation:

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

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

After compilation:

```
if(seq_par(EXTRACTED_STATIC_DYNAMIC_FEATURES))
  for_each(seq, range.begin(),range.end(),lambda);
else
  for_each(par, range.begin(),range.end(),lambda);


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

✓Machine Learning & Compiler & Runtime



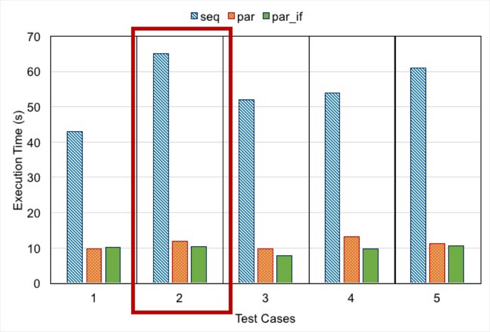
Experimental Results

Item	Detail
CPU	Intel Xeon E5-2630
Compiler	Clang 4.0.0
Cores	8
Frequency	2.4GHZ
OS	32 bit Linux Mint 17.2
HPX	0.9.99
Main Memory	65GB

Experimental Results

Test	Loop	Itr.	Total opr.	Float opr.	Cmpr. opr.	level	Policy	% Chunk size
1	l_1	10000	400100	200000	101010	2	par (8)	0.001
	l_2	20000	450026	250000	150503	2	par (8)	0.001
	l_3	20000	502040	250000	103051	2	par (8)	0.001
	l_4	500	550402	200000	150102	1	par (8)	0.1
2	l_1	150000	350106	101010	500	2	par (8)	0.001
	l_2	100	10050016	5000000	2505013	3	seq	0.1
	l_3	100	25000000	3010204	1500204	3	seq	0.1
	l_4	50000	4000450	200000	100150	1	par (8)	0.01
3	l_1	500	4504030	250000	150300	2	par (8)	0.01
	l_2	400	3502020	200000	100405	1	par (8)	0.01
	l_3	2000	250033	150000	103040	3	seq	0.1
	l_4	2500	350400	150000	100600	3	seq	0.1
4	l_1	20000	204002	100000	10320	2	par (8)	0.001
	l_2	30000	400000	150102	10000	2	par (8)	0.001
	l_3	300	550000	44000	20030	3	seq	0.1
	l_4	400	450000	50400	10602	3	seq	0.1
5	l_1	200	4502001	150000	101004	3	par (8)	0.01
	l_2	700	400020	300000	150006	3	par (8)	0.01
	l_3	300	302020	20000	14005	2	par (8)	0.01
	l_4	100	50400	20000	10110	2	seq	0.1

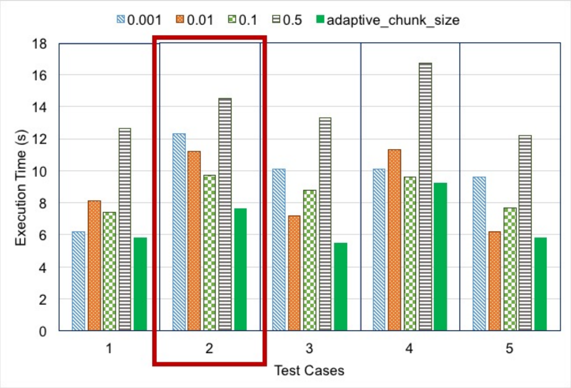
Experimental Results



- As all 4 execution policy determined for the first test is par, the overhead of the costs_fnc resulted in degrading performance.

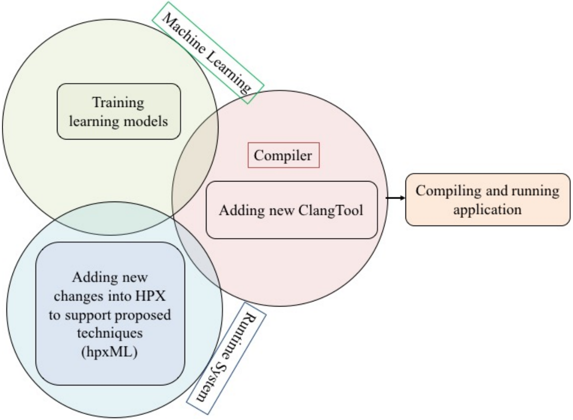
✓ 15% – 20% improvement.

Experimental Results



✓ 45%, 32%, 37% and 58% improvement over setting chunks to be 0.001, 0.01, 0.1, or 0.5 iterations.

Conclusion



- ✓ https://github.com/STEllAR-GROUP/hpxML
- ✓ Join our IRC channel #*ste*||*ar* if you need any help ☺ .

Thanks for your attention!
Questions?