Abstract

The major challenge : the difficulty of improving application scalability with conventional techniques.

One of the solutions : prefetching data before its actual access is executed.

The generic prefetching scheme proposed in HPX, which results in:

✓ improving the parallel performance by leveraging the abstraction capabilities,

✓utilizing asynchronous task-based execution flow,

✓exploiting execution policies for the fine-grained control.

HPX Data Prefetching Iterator

Zahra Khatami, Hartmut Kaiser, and J. Ramanujam

Center for Computation and Technology, Louisiana State University, The STE||AR Group, http://stellar-group.org

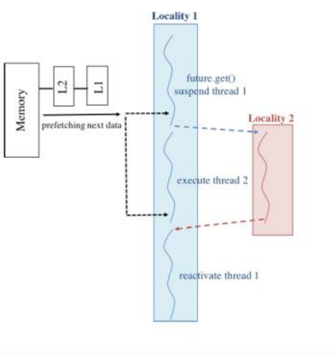

Figure 1: The principle of the operation of *future* in HPX. Thread 1 is suspended only if the results from locality 2 are not readily available. Thread 1 access the *future* value by performing a *future:get()*. If the results are available Thread 1 continues to complete execution. Data of the next chunk is prefetched for each thread as well.

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

Table 1: The execution policies defined by the Parallelism TS and implemented in HPX.

HPX

➢parallel C++ runtime system

➢enables fine-grained task parallelism

➢fully generic higher-level API

➢extensible framework for parallelizing application

Introduction

Data prefetching methods:

•Hardware prefetching method: predicting the future cache misses by using the past access pattern with considering the data stream.

•Software prefetching method: prefetching data before the execution of its actual access by using the prefetch directives into the code.

•Thread based prefetching method: executing code in the prefetcher thread context and bringing the data of the next cache line into the shared cache before the main thread accesses it:

- ✓ Precomputing the load addresses accurately.
- ✓ Following more complex pattern compared to the other methods.

However, scaling can be degrade with

Thread based prefetching: Cache misses, Global barriers

and Resource competition.

The cache prefetcher used in HPX aids prefetching that

✓reduces the memory accesses latency, and

✓inhibits the global barrier.

✓*for_each* helps creating sufficient parallelism by determining the number of the iterations to run on each HPX thread.

✓HPX threads makes the invocation of the loop asynchronous, while the data of all containers within the loop of the next step is prefetched in each iteration.

✓HPX is able to prefetch data in sequential or in parallel with applying an execution policy.

✓HPX prefetcher works with any data type of the containers and even if each container has different data type.

Prefetching Iterator Implemented in HPX

•***for_each*** is one of the HPX parallel algorithms used to evaluate the proposed prefetching method.

•Data of the next iteration step is prefetched in the cache memory with the prefetching iterator called in each iteration within the ***for_each*** .

•HPX combines prefetching method with the asynchronous task execution by providing a new *future* instance representing the result of the function execution (Figure 1).

•The program execution is divided into several chunks within ***for_each*** (Figure 2) and its iterator is developed to prefetch the data of the next chunk size in either sequential or in parallel.

•The prefetching iterator is initialized in ***make_prefetcher_context*** and it executes with ***ctx.begin().*** ***ctx*** is the struct that references to all container in the

•The distance between each two prefetching operations is computed based on the value of ***prefetch_distance_factor***, which is the factor of the length of the cache line.

Experimental Results

In an N-Body problem, there are N particles moving under the influence of the gravitational attraction.

Prefetching iterator increases bandwidth vs. standard random access iterator by 30% on average using two NUMA domains with 8 threads each (figure 3).

The results of the performance of the prefetching iterator with different ***prefetch_distance_factor*** are shown in figure 4 and 5 for 1 and 2 NUMA domains respectively:

•For the large distance, data prefetching cannot improve the parallel performance.

•Very small prefetcher distances make more data to be prefetched,
which become more expensive and dominate the gains from prefetching.

Results

```
auto ctx = hpx::parallel::make_prefetcher_context(
  loop_range.begin(), loop_range.end(),
  prefetch_distance_factor,
  container_1, container_2, ..., container_n);

hpx::parallel::for_each(policy,
        ctx.begin(), ctx.end(),
        [&](std::size_t i)
        {
           container_1[i] = ...;
           container_2[i] = ...;
            .
            .
            .
           container_n[i] = ...;
         });
```

Figure 2: The prefetching method used in for_each

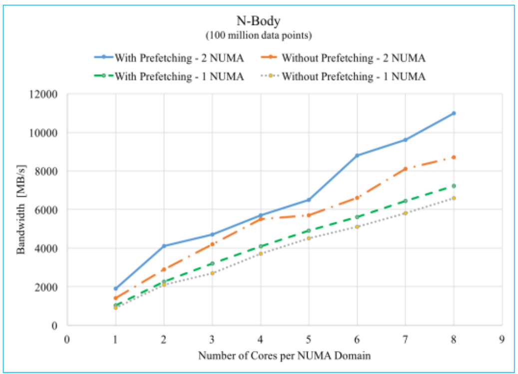

Figure 3: The data transfer rate of for_each with the standard random access iterator versus prefetching iterator

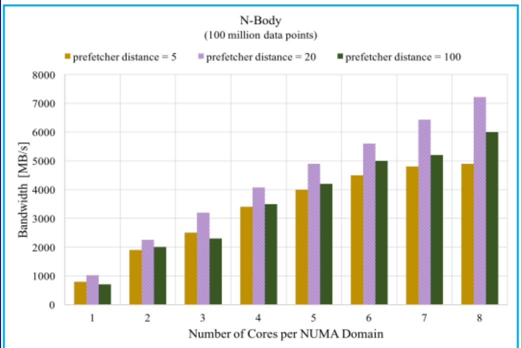

Figure 4: 1 NUMA Domain-The data transfer rate

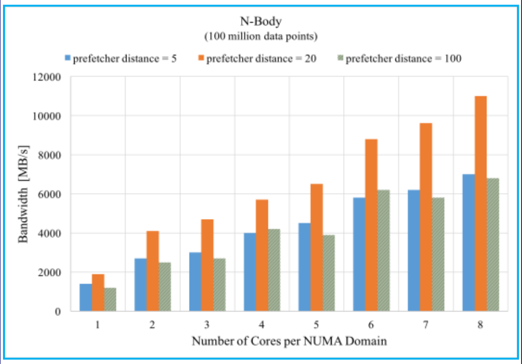

Figure 5: 2 NUMA Domain-The data transfer rate

We would like to thank Antoine Tran Tan and Adrian Serio from Louisiana State University for the invaluable and helpful suggestions to improve the quality of the research. This works was supported by the NSF award 1447831.