High-level Status Summary

Technology	**Description (Institution)**	**Status**
Resilience	Containment Domains in SWARM (ETI)	In Progress
Application migration	MPI Interoperability (ETI)	In Progress
Parallelizing Compiler	Runtime support for block sparse data structures (Reservoir)	In Progress
Group Locality	Fine Grain Multithreading via Compiler Support (PNNL)	In Progress
Parallel Language	Evaluation of SPMD mode with NAS Parallel Benchmarks (UIUC)	In Progress
Parallel Language & Parallelizing Compiler	Integration of PIL with R-Stream Compiler to generate SWARM code (UIUC+Reservoir)	In Progress

Table of content:

Summaries of Quarterly Work (Q11)

ETI Work

During this reporting period (Q11: 03/01/2015 - 05/31/2015), ETI has been working on the following tasks, according to the SOW.

Task 8.1: Research integration of containment domain execution and recovery with codelet scheduling (in progress)

Task 8.2: Research integration of CD persistence infrastructure API with SWARM (in progress)

Task 10.1: Study MPI interoperability (in progress)

Resilience (containment domains)

Software and hardware errors are expected to be a much larger issue on exascale systems than current hardware. For this reason, resilience must be a major component of the design of an exascale system. By using containment domains, we propose a resilience scheme that works with the type of codelet-based runtimes expected to be utilized on exascale systems. We implemented a prototype of our containment domain framework in SWARM, and adapted a Cholesky decomposition program written in SWARM to use this framework.

In this quarter, we have analyzed the results and produced a detailed report on resilience. We will demonstrate the feasibility of this approach by showing the low overhead and high adaptability of our framework. There is a detailed report included in the "Topic detail" section below.

Results of our resilience research is summarized in the following ETI Technical Report.

- Sam Kaplan, Sergio Pino and Guang R. Gao, “Landing Containment Domain on SWARM: Toward a Robust Resiliency Solution on A Dynamic Adaptive Runtime Machine”. ETI Technical Report Dynax-Swarm-2015-01, in preparation.

MPI Interoperability

Exascale software will be unable to rely on minimally invasive system interfaces to provide an execution environment. Instead, a task-parallel software runtime layer is necessary to mediate between an application and the underlying hardware and software. Industry and academia have years of effort developing MPI codes. For this reason, a progressive transition to the new exascale execution models will require the interoperability with legacy MPI codes. Ideally this interoperability should not degrade the current performance of legacy codes, but it may hinder optimal performance and programmer intervention may be required to remove bottlenecks. In this quarter, we explored two methods to provide this interoperability named MPI+SWARM and Codelet MPI. First, MPI+SWARM takes an MPI program and add SWARM calls. Second, Codelet MPI creates an MPI compatibility layer in SWARM which is used by applications. We show the feasibility of these approaches by presenting some simple applications. There is detailed report included in the "Topic detail" section below.

These results are summarized in the following ETI Technical Report.

Sergio Pino and Guang R. Gao, “Legacy MPI Codes and its interoperability with fine grain task-parallel runtime systems for Exascale”. ETI Technical Report Dynax-Swarm-2015-02, in preparation.

Future work

For Q12, we expect to:

- Complete tasks 8.1 and 8.2
- Complete the study of MPI interoperability task 10.1 by implementing a benchmark such as matrix-matrix multiply or cholesky.
- We have two technical report in preparation for the previous tasks. The tentative named of the reports are “Landing Containment Domain on SWARM: Toward a Robust Resiliency Solution on A Dynamic Adaptive Runtime Machine” and “Legacy MPI Codes and its interoperability with fine grain task-parallel runtime systems for Exascale”.

Reservoir Work

This quarter, Reservoir has contributed to the following SOW tasks:

Task 2.4: Research compiler code generation for data placement and movement

Task 3.4: Optimize unstructured computations

Task 5.7: SCALE and R-Stream code generation

Reservoir worked on automatic parallelization to block-structured computations, a class of unstructured codes. The approach relies on lifting sparsity information from the element level to outer loop levels, and using logical DMAs as a layer to abstract away distributed sparse data structures. This is described in more detail in the *Parallelization of block-structured codes* section below.

We also supported UIUC in creating a 2-level parallelization of HTA/PIL programs by generating programs that can be parallelized by R-Stream to SWARM. We discuss the issues encountered and their fix in section *R-Stream support for PIL parallelization.* We remind the reader that the technical plan was improved at the beginning of the period of performance, going from a “R-Stream to SCALE to SWARM” code generation path to a “R-Stream to SWARM” path, the latter offering more control over the generated code (and also potentially less overhead).

Future work

For Q12, we expect to complete our support for block structured codes. We will also keep supporting UIUC with Task 5.7, although we are not expecting much more work to be done on our end.

UIUC Work

In this quarter, we focused on the following two tasks:

Task 5.3: Evaluation of the PIL implementation and API (in progress)

Task 5.7: SCALE and R-Stream code generation (in progress)

Performance Evaluation

In the last quarter, we finished the implementation of shared-memory SCALE backend for SPMD PIL and the HTA library on top of SPMD PIL. This quarter, we performed extensive analysis in order to understand the execution behavior and performance overhead. We analyzed the results of both the fork-join execution and the SPMD execution of NAS Parallel Benchmarks (NPB). We found that the fork-join mode is preferable due to the bulk-synchronous nature of the algorithms used in NPB. In order to get a better understanding of applications which are not bulk-synchronous, we implemented a tiled dense Cholesky factorization in HTA and analyzed the performance. The details are presented in Topic Details section below.

SCALE and R-Stream code generation

We continue to work on the integration of PIL with the R-Stream compiler. The integration is mostly complete with a single issue preventing the completion. The issue is the interface between the PIL side and R-Stream side of the code. Since SWARM codelets are non preemptable, when PIL makes the call to the R-Stream compiled function, the PIL codelet making the call has to wait for the function to complete. However the R-Stream compiled function is a collection of codelets. While PIL waits for the codelets to complete, they are taking up SWARM worker threads. If the number of PIL codelets exceeds the number of SWARM threads, the program will deadlock. We are working on a solution to restrain the parallelism used on the PIL side to free up some SWARM worker threads for R-Stream generated code to utilize.

Future work

In the last quarter, the UIUC team will continue the performance optimization of the HTA and PIL implementation and we also plan to evaluate programmability using objective metrics.

PNNL Work

During this reporting period (Q11: 03/01/2015 - 05/31/2015), PNNL has been working on the following tasks, according to the SOW.

> Task 9.2: Investigate initial and static data placement at each memory level(in progress)

Abstract

Group Locality is a concept in which threads collaborate at a very fine grain level. By being aware of access pattern of neighboring threads, it improves locality, performance and reduces memory access latency. So far we have shown that our Group Locality framework allows multilevel parallelism with our jagged tiling technique for stencils with complicated dependencies. It also uses micro data flow execution within tiles to allow fine-grain execution.

Introduction

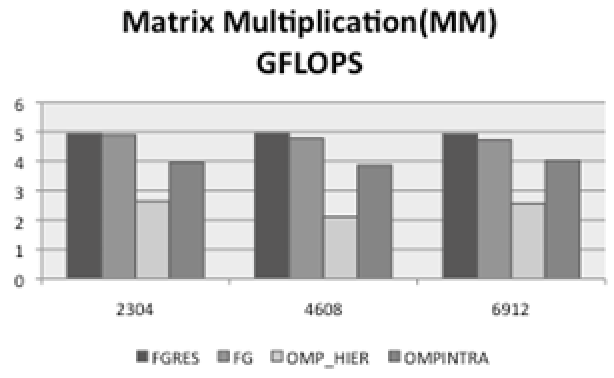
Another important aspect of Group Locality that we have explored in this quarter is data movement and restructuring to improve memory access behavior amongst a group of threads. Such strategy is applied based on the access pattern and the reuse. Moving data comes with an overhead of extra memory space and the memory operation. However, when multiple threads share the same space minimizing interference and improving locality, the cost gets amortized and provide an overall gain in performance.

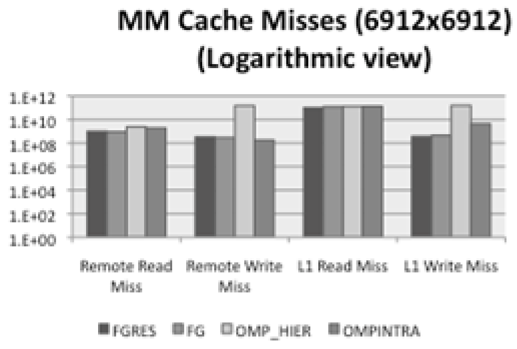
We use multiple level transformations to bring data closer in memory. Using transformation matrix *'T',* we move the data from original space '*A*', such that restructuring space *R*= *TA.*

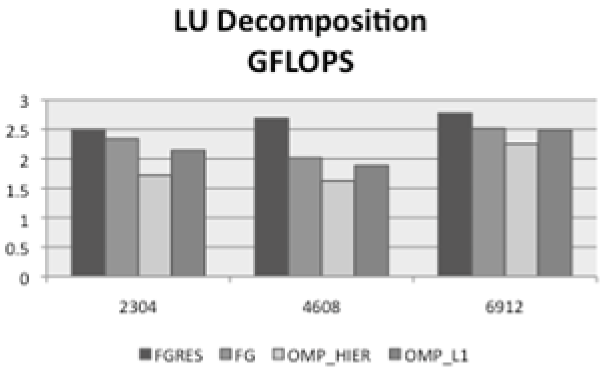
Results

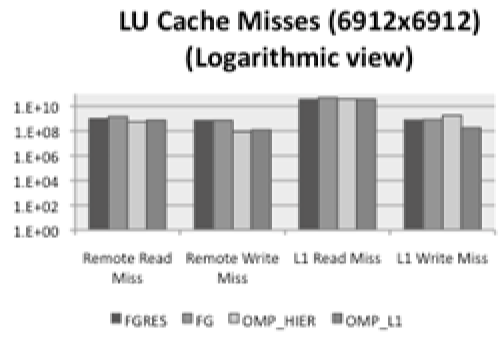
We ran our experiments on the TileGX architecture with 36 cores, partitioned into 4 cores per group, that form a square on the TileGX grid. We used Matrix Multiplication (MM) and LU Decomposition (LU) for our experiments since both allow significant amount of data reuse.

The restructuring is done to make the access contiguous and bring all elements within the outer tile together in the restructured space. This approach improves the residence of the data in both local caches and the grid (TileGx has a shared L2 cache in the grid). The charts below show the performance and the cache misses obtained using performance counters for MM and LU respectively. The x-axis on the left chart shows three different matrix sizes whereas the right chart depicts various cache miss events. 'FGRES' is our original, parallelism exploiting framework 'FG' enhanced with restructuring optimizations. We compare our results with two OMP strategies that exploit either hierarchical- 'OMP_HIER' or intra-parallelism 'OMPINTRA'. Our results show that when threads collaborate, we gain in performance and reduce cache misses. In the case of MM, our techniques show better performances when compared to the matrix multiply code running with OpenMP. However, restructuring provides a small improvement over our fine-grained implementation. In the case of LU, the restructuring improvements are more predominant due to the higher reuse opportunities that the LU code offers. Currently, we are further characterizing and analyzing these kernels.









Issues

Previously, our experiments consisted of grouping Hyperthreads in Intel Xeon Phi that work together as a unit. With TileGX, one of the major issues we encountered while forming a group with physical cores was the synchronization overhead. To mitigate such issues, we experimented with many techniques and currently settled with the mix of both fine-grain synchronization and a coarse grain barrier.

Future Work
We believe that communicating across cores and keeping all of them up to date with the data in the restructuring space results in lot of coherency traffic. We plan to analyze such traffic and reduce it to improve performance further.

Topic Detail:

ETI:

Resilience (containment domains): Tasks 8.1 and 8.2

Abstract

Software and hardware errors are expected to be a much larger issue on exascale systems than current hardware. For this reason, resilience must be a major component of the design of an exascale system. By using containment domains, we propose a resilience scheme that works with the type of codelet-based runtimes expected to be utilized on exascale systems. We implemented a prototype of our containment domain framework in SWARM, and adapted a Cholesky decomposition program written in SWARM to use this framework. We will demonstrate the feasibility of this approach by showing the low overhead and high adaptability of our framework.

Introduction

Exascale systems are expected to exhibit a much higher rate of faults than current systems, for a few reasons. Given identical hardware, failure rate will increase at least linearly with number of nodes in a system. In addition, exascale hardware will include more intricate pieces, including smaller transistors, which will be less reliable due to manufacturing tolerances and cosmic rays. Software will also have increased complexity, which again results in more errors. [1] The combination of the above factors indicates that resilience will be incredibly important for exascale systems, to a higher degree than it has for any preceding generation of hardware.

On current systems, most resilience methods take the form of checkpointing. Common types of checkpointing exhibit flaws that limit their scalability to exascale, due to the larger amount of state needing to be saved, and the lower mean time between failures. For this reason, it is desirable to have a resilience scheme that requires no coordination and can scale to any workload size. To this end, we leveraging ideas from containment domain research performed by Mattan Erez and his team at University of Texas at Austin [2]. Similar to codelet model used in SWARM, containment domains exhibit a distributed, fine-grained, hierarchical nature. For this reason, we expect the impact of containment domains to be well realized when mapping onto a codelet model. We hope to show the feasibility of this approach by implementing containment domains in SWARM, using a continuation-based API.

SWARM (SWift Adaptive Runtime Machine) is a codelet-based runtime created at ETI. We have previously adapted applications to use fine-grained, distributed, low-overhead SWARM codelets, and have demonstrated positive results in both performance and scalability. Because of its efficiency, maturity, and programmability, as well as our own familiarity with it, SWARM was chosen as the underlying runtime for our resilience research.

By implementing a prototype containment domain framework in SWARM, we show the feasibility of utilizing containment domains in a codelet-based runtime. Specifically, we created a continuation-based API to allow containment domains to conform to the requirements of the codelet model: fine-grained, non-blocking, and largely self-contained. We adapted a Cholesky decomposition program written in SWARM to use this API, showing that necessary functionality is implemented and performs correctly. We also benchmarked this program to show that our implementation of containment domains has a very low overhead.

Background

Containment domains were first proposed by Mattan Erez at University of Texas at Austin. Described as a "scalable and efficient resilience scheme" [3], containment domains are a possible solution to the increased error rate expected to be seen on exascale systems. By allowing data preservation and recovery in an application-specific manner, containment domains allow for uncoordinated, low-overhead resilience.

At a high-level, a containment domain contains four components: data preservation, to save any necessary input data; a body function which performs algorithmic work; a detection function to identify hardware and software errors; and a recovery method, to restore preserved data and re-execute the body function. The detection function is a user-defined function that will be run after the body. It may check for hardware faults by reading error counters, or for software errors by examining output data (e.g. using a checksum function). Since containment domains can be nested, the recovery function may also escalate the error to its parent. Since no coordination is needed, any number of containment domains may be in existence, with multiple preserves and recoveries taking place simultaneously.

An initial prototype implementation of containment domains was developed by Cray [4]. In addition, a more fully-featured containment domain runtime is in currently in development by Mattan Erez and his team. However, none of these implementations support a continuation-based model. If exascale hardware is going to use a codelet-based runtime, it is necessary to adapt these ideas to support such a model. For this reason, it is important that we demonstrate use of a codelet-based runtime, in this case SWARM.

Containment Domains in SWARM

We developed our containment domain API as an additional feature of the SWARM runtime. This allows us to leverage existing runtime features and internal structures in order to support the hierarchical nature of containment domains. The main features needed include data preservation, user-defined fault detection functions, and re-execution of failed body functions. This feature set was realized by implementing the following functions:

`swarm_ContainmentDomain_create(parent)`: Create a new containment domain as a child of `parent`.

`swarm_ContainmentDomain_begin(THIS, body, body_ctxt, check, check_ctxt, done, done_ctxt)`: Begin execution of the current containment domain (`THIS`) by scheduling the `body` codelet with `body_ctxt` as its context parameter. The `begin` codelet is scheduled with its `NEXT` parameter set to a callback within the SWARM runtime, which will schedule the `check` codelet with `check_ctxt` as its context. In the `check` codelet, `NEXT` is again set to a SWARM-internal codelet, which takes a boolean success value as the `INPUT` parameter. If this value is `TRUE`, the `done` codelet is scheduled, with `done_ctxt` as its context. Codelet chaining from `begin` and `check` is supported, provided the user preserves the original `NEXT` parameter and schedules it from every end point in the chain.

`swarm_ContainmentDomain_preserve(THIS, data, length, id)`: In the specified containment domain (`THIS`), do a memory copy of `length` bytes from `data` into a temporary location inside the CD. We support multiple preservations per CD (e.g. to allow preservation of tiles within a larger array, such that the individual tiles are non-contiguous in memory), by adding a user-selected `id` field. For each containment domain in SWARM, a boolean value is set based on its execution status. On first execution, data is preserved normally. On subsequent executions, data is copied in reverse (i.e. from the internal preservation into the `data` pointer).

`swarm_ContainmentDomain_finish(THIS)`: Close current containment domain (`THIS`), discard all of its preserved data, and make its parent active.

Example

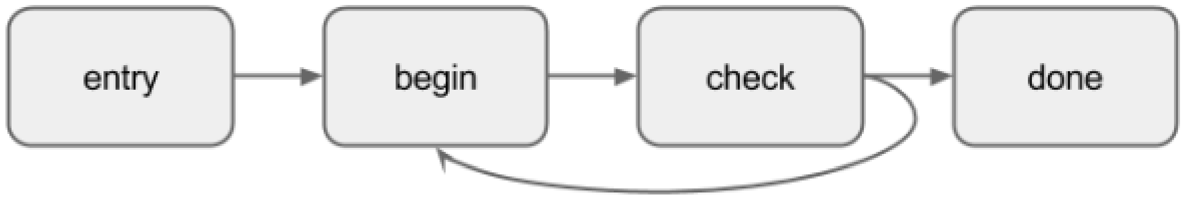


Figure 1: Simple CD application

Figure 1 shows a graph of a very simple program using containment domains. This example shows a small program that multiplies two integers, and uses a single containment domain. The `entry` codelet initializes the containment domain, and enters it. The `begin` codelet multiplies its two inputs, and stores the result in `C`. The `check` codelet performs the same multiplication, and compares with the original result in `C`. If the results are not the same, an error has occurred and must be corrected. The `begin` codelet is re-executed, and the inputs are recovered from their saved locations. This continues until the `begin` and `check` codelets achieve the same result, in which case the `done` codelet is called, and the runtime is terminated.

Results

The following observations were made as a result of our research.

Observation 1: Feasibility

It is feasible to adapt a codelet-based application to use containment domains. This can be seen from the implementation of our framework in SWARM, and our working Cholesky application using said framework.

Observation 2: Efficiency
Our implementation has a very low overhead. As shown in Figures 2 and 3, the relative overhead of a containment domain is dependent on the amount of work performed inside it. If the workload is large enough, the overhead from adding containment domains is negligible.

Observation 3: Resilience
As shown in Figure 4, it is possible to simulate errors, and have the containment domain framework re-execute codelets as necessary to achieve the correct result.

To show that our prototype implementation has sufficient functionality and efficiency, we instrumented a Cholesky decomposition program in SWARM to use containment domains. This was based on a Cholesky implementation included as an example in the SWARM distribution.

The Cholesky program has only three main codelets, one for each linear algebra routine run on a tile (POTRF, TRSM, and GEMM/SYRK), and each of these is called a number of times for each tile. For our purposes, each of these is considered a containment domain. In our model, we only considered faults similar to arithmetic errors; that is, incorrectly calculated results. For this reason, we did not need to preserve input data unless it would be overwritten by an operation (e.g. the input/output tile for a POTRF operation). In order to simulate arithmetic errors, rather than relying on error counters from actual faulty hardware, a random number generator was used. If a random number was below a configurable threshold, a fault was deemed to have occurred.

For all of the following experiments, the program was run on a dual-processor Intel Xeon system, using 12 threads. Although the machine has 24 hyperthreads available, benchmarks showed that limiting the runtime to 12 threads gave the best performance for this program. The workload sizes were confirmed to not exhaust the physical memory of the machine. Since the execution time of the program naturally varies between runs (due to factors like scheduling differences), all of the included times are the average of 5 runs.

In Figure 2, we show the execution time for our Cholesky program with different tile sizes, with a constant matrix size. The overhead from our implementation of containment domains can be seen below. Data preservation accounts for the bulk of this overhead. Besides preservation, overhead comes from the additional codelets needed (check and done functions), and random number generation needed to determine success. These sources of overhead are separated from preservation overhead in the graph.

Less total data is preserved with fewer tiles (hence fewer iterations), since each iteration must preserve data from the rest of the matrix. For this reason, the time spent preserving data decreases sharply with a larger tile size. However, increasing the tile size too much results in less parallelism overall.

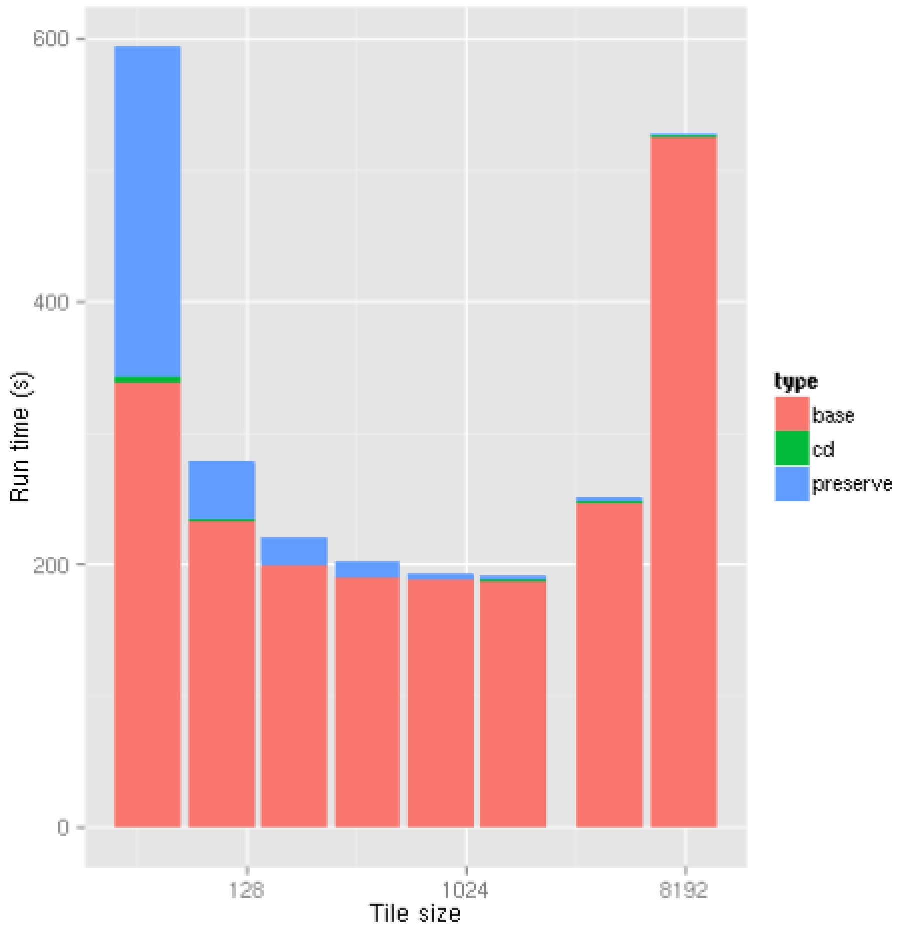


Figure 2: Cholesky execution time

Figure 3 shows the percentage overhead from containment domains (excluding preservation) for each tile size. Since the expected additional work is constant for each containment domain (scheduling and running the check and done codelets), regardless of tile size, the percentage overhead is larger for smaller tile sizes, for which less algorithmic work is performed in a containment domain. Except for the smallest tile sizes we tested, the variation is largely due to random scheduling perturbations (including one case where the program ran slightly faster with containment domains!). This shows that assuming enough work is done per containment domain, the overhead from our framework is negligible

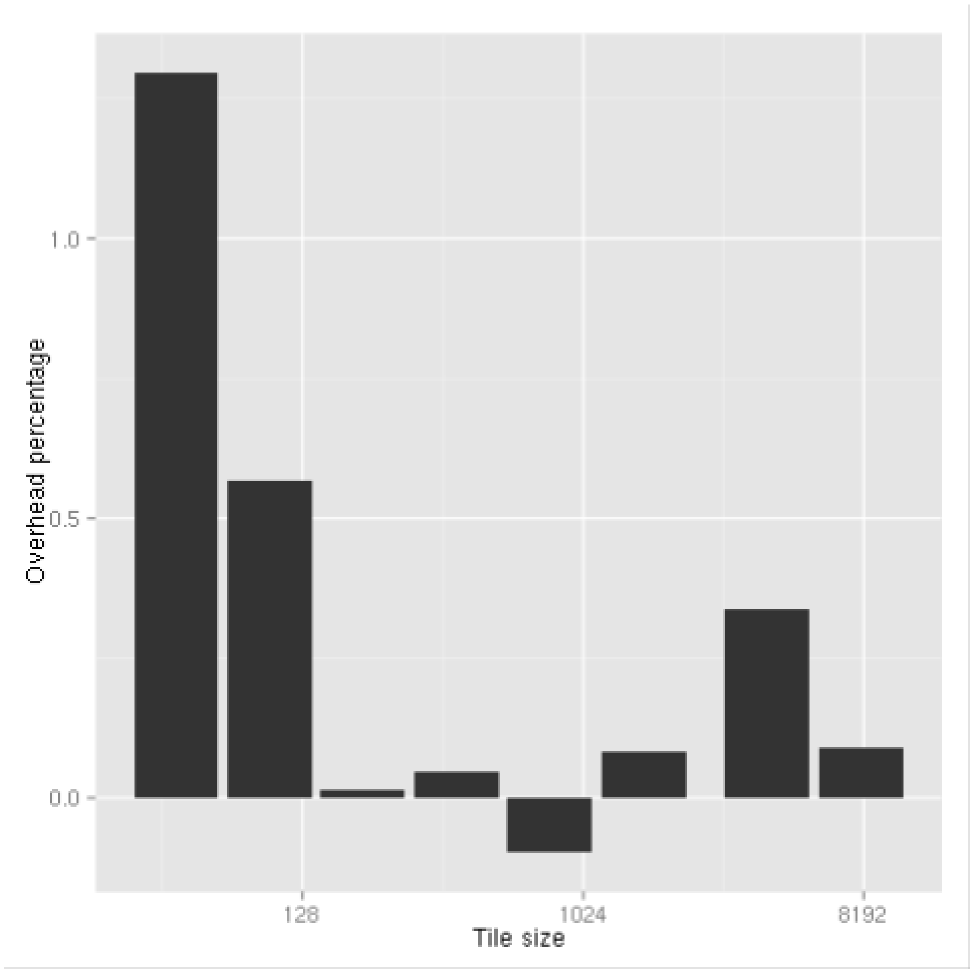


Figure 3: Percentage overhead vs. tilesize

We also examined the running time of our Cholesky program with different simulated error rates. Each of these experiments was performed with a 40000x40000 matrix (same as above), with tile size 200x200. As shown in Figure 4, the running time conforms reasonably well to the idealized case (i.e. an error rate of 0.5 results in a 2x execution time). The resulting time is slightly lower than expected because a small portion of code (to check readiness of codelet dependencies) is outside of containment domains and therefore not re-executed with the main body functions.

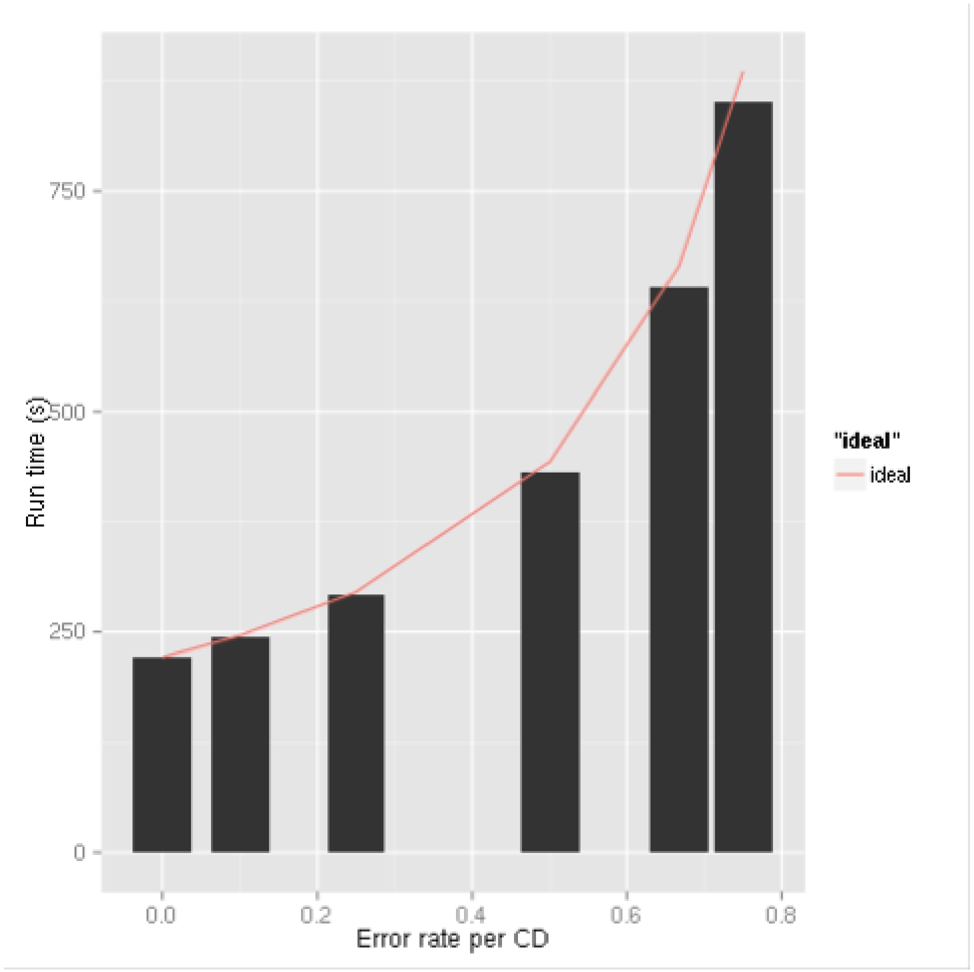


Figure 4: Execution time with simulated errors

Conclusions and Future work

In conclusion, we have demonstrated that containment domains can be adapted to the codelet model. Our Cholesky application shows that containment domains can be used in a decentralized, continuation-based manner, to provide a fine-grained, low-overhead framework for resilience. Although the best method for adapting individual applications is still an open problem, we have shown that this is an approach worth pursuing.

Our prototype requires the user to obey a few limitations, due to its limited feature set. Firstly, we only support single-node SWARM applications. We also only allow references to the inner-most CD on preservation. To avoid data duplication, a fully fleshed out implementation would allow preservation in an outer CD, which could be referred to by multiple child CDs. An obvious source of improvement would be to add support for these missing features.

Due to the Cholesky program's very decentralized call graph, it was not feasible to add nested containment domains in any useful manner. If we had another example, it would likely provide additional insight. We attempted to implement an SCF program, but due to the increased complexity of the application, we ran into issues completing this in time.

Sources

[1] Cappello, F., Geist, A., Gropp, W., Kale, S., Kramer, B., & Snir, M. (2014). Toward Exascale Resilience: 2014 update. *Supercomputing Frontiers And Innovations, 1*(1), 5-28. doi:http://dx.doi.org/10.14529/jsfi140101
[2] http://lph.ece.utexas.edu/public/CDs/ContainmentDomains
[3] Jinsuk Chung, Ikhwan Lee, Michael Sullivan, Jee Ho Ryoo, Dong Wan Kim, Doe Hyun Yoon, Larry Kaplan, and Mattan Erez. **Containment Domains: A Scalable, Efficient, and Flexible Resilience Scheme for Exascale Systems.** *In the Proceedings of SC'12*. November, 2012.
[4] http://craycontainment.sourceforge.net/index.html

SWARM and MPI Interoperability: Task 10.1

Abstract

Exascale software will be unable to rely on minimally invasive system interfaces to provide an execution environment. Instead, a task-parallel software runtime layer is necessary to mediate between an application and the underlying hardware and software. Industry and academia have years of effort developing MPI codes. For this reason, a progressive transition to the new exascale execution models will require the interoperability with legacy MPI codes. Ideally this interoperability should not degrade the current performance of legacy codes, but it may hinder optimal performance and programmer intervention may be required to remove bottlenecks. In this work, we focus on the codelet-based execution model called SWARM, and explore two methods to provide this interoperability named MPI+SWARM and Codelet MPI. First, MPI+SWARM takes an MPI program and add SWARM calls. Second, Codelet MPI creates an MPI compatibility layer in SWARM which is used by applications. We show the feasibility of these approaches by presenting some simple applications. These examples are attached as a .zip file.

Introduction

The DynAX project needs to interoperate with legacy MPI codes. Because the codes are being modified and recompiled to fit into a new exascale paradigm, we assume that the codes

can be recompiled through the XStack software. We also note that interoperability with MPI should not degrade the current performance of legacy codes, but it may hinder optimal performance and programmer intervention may be required to remove bottlenecks.
Legacy code can be parallelized between MPI calls rather straightforwardly. The main MPI thread will be suspended while the parallel code is executed, then the main thread is resumed in a manner similar to how OpenMP and MPI interoperate today. However, this method is limited because only the main MPI thread may make MPI calls.

In this work, we focus on the codelet-based execution model called SWARM, and explore two methods to provide this interoperability named MPI+SWARM and Codelet MPI. First, MPI+SWARM takes an MPI program and add SWARM calls. Second, Codelet MPI creates an MPI compatibility layer in SWARM which is used by applications. We show the feasibility of these approaches by presenting some simple applications.

MPI+SWARM
The first approach for the MPI interoperability is called MPI+SWARM. Here, a developer takes a base MPI program and add SWARM calls in a similar way as the hybrid model MPI+OpenMP. In this approach, SWARM doesn't perform MPI calls to communication routines (point to point communication routines, such as MPI_Send) or Collective Communication Routines (such as synchronization, data movement, or collective computation). The general code structure for this approach is presented below. This can be considered as the first step in order to test the interoperability between the two runtime systems, and doesn't target real applications to use it.

```
#include <eti/swarm_convenience.h>
#include <mpi.h>
// Declare N codelets
CODELET_DECL(c0);
...
CODELET_DECL(cN-1);
int main(int argc, char *argv[]) {
        …
        // Initialize MPI.
        MPI_Init(...); // parallel code begins
        // some MPI calls
        …
        // using SWARM to exploit parallelism in each MPI process
        swarm_posix_enterRuntime(NULL, &CODELET(...), …, ...);
        …
        // more MPI calls
        ...
        // Terminate MPI environment
```

```
            MPI_Finalize();
            ...
}
// Codelet implementations
...
```
Fig 1: General structure of a MPI+SWARM application

It is important to notice that we assume a basic interaction as described below:

1. Perform MPI calls to communication routines (point to point communication routines, such as MPI_Send) or Collective Communication routines (such as synchronization, data movement, or collective computation).
2. Enter the swarm runtime system passing the necessary data to the first codelet.
3. Performing the intra-node parallel work with SWARM.
4. Exiting the SWARM runtime.
5. Perform MPI calls to communication routines or Collective Communication routines.
6. Go to 1 is needed, if not exit MPI environment.

Codelet MPI

The second approach for the MPI interoperability is called Codelet MPI. Codelet MPI creates an MPI compatibility layer in SWARM which is used by applications. We addressed this in two ways. First, at the user level code, we created codelets that perform blocking MPI send or recv, each codelet schedules its continuation once the blocking call returns control to the codelet. Second, It is basically a library that provides two general purpose codelets, one to perform non-blocking MPI_Send and the other to perform non-blocking MPI_Recv, each codelet schedules its continuation when the test for completion of the non-blocking MPI call is true. For this first version, we assume that the application has data dependencies between codelets, so a codelet that performs an MPI_recv operation will need to make sure the data has been received before scheduling its continuation codelet, an example of this behavior is presented in fig 4 and fig 5.

Creating a Codelet MPI that uses MPI blocking calls

By creating codelets that wrapped MPI blocking calls we created some simple applications that demonstrate the possibility of performing MPI calls inside codelets, and it motivated us to pursue the approach described in the next section. This straightforward approach uses a similar basic interaction or code structure as the one defined in the section MPI+SWARM. However, in this approach the step 3 assumes that SWARM codelets not just perform the intra-node parallel but also can perform MPI calls.

An example of a codelet performing a blocking MPI_Send is shown below, it is part of a modified version that uses SWARM and MPI of the mpi_ping.c example presented in [1]. The drawback of this implementation comes from the restricted functionality that a codelet must follow. In SWARM, codelets must not block, because it ties up the runtime thread indefinitely and could stalls out program execution [2]. Thus, unless you can afford that cost, it is better to find a way to define a codelet that interacts with MPI in a non-blocking approach, as presented in the next section.

```
CODELET_IMPL_BEGIN_NOCANCEL(rank0_Send)
  info* data = (info*) (swarm_natP_t) THIS;

  int dest = 1, source = 1, rc;
  rc = MPI_Send(&data->outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);

  // continuation
  swarm_schedule(&CODELET(rank0_Recv), THIS, NULL, NULL, NULL);

CODELET_IMPL_END;
```

Fig 2: Excerpt of code for a codelet using MPI blocking calls.

Creating a Codelet MPI that uses MPI non-blocking calls

We implemented this functionality in a library called dynax_mpix.h. This library provides two general purpose codelets, one for perform asynchronous MPI_Isend and the other to perform asynchronous MPI_Irecv. We assume that the application has data dependencies between codelets, so a codelet that performs an MPI non-blocking call will need to make sure the data has been received before scheduling its continuation codelet. These codelets uses the underlying MPI non-blocking calls and check (without tie up a runtime thread indefinitely) whether or not the operation was successful. If it was successful, then the codelet calls the continuation codelet defined by the user. If the non-blocking operation hasn't complete then the codelet yields its control in order to give room for another codelet to execute in the current runtime thread.

The API exposes three components and it is presented in figure 3. First, there is a typedef struct called mpix_str, which is used to pass the mpi required information as the THIS parameter in the dynax_mpix codelets.

Second, the codelet to perform non-blocking MPI send is called mpix_send. It takes as its THIS argument an instance of mpix_str. Third, the codelet to perform non-blocking MPI recv is called mpix_recv. As mpix_send, it takes as its THIS argument an instance of mpix_str. As presented in the code excerpt, the codelet perform an MPI_Isend/MPI_Irecv call and test continuously if the non-blocking operation has finished. If it has not finished then execute another codelet by calling swarm_yield(). With this simple approach SWARM is able to perform MPI calls without tie up a SWARM runtime thread indefinitely.

```
//  Information: buffer, count, type, dest, tag, comm used to call the underlying MPI call
typedef struct {
   int rank;
   void* buf;
   int count;
   MPI_Datatype type;
   int dest_src;
   int tag;
   MPI_Comm comm;
} mpix_str;

// Codelet to manage non-blocking mpi sends
CODELET_DECL(mpix_send);

// Codelet to manage non-blocking mpi recvs
CODELET_DECL(mpix_recv);

// Uses MPI_Test to tests for the completion of a send or receive
bool mpix_mpiTest(MPI_Request* req);
...
CODELET_IMPL_BEGIN_NOCANCEL(mpix_send)

   mpix_str data = *(mpix_str*) (swarm_natP_t) THIS;
   MPI_Request req;

   // non-blocking sending the data
   MPI_Isend(data.buf, data.count, data.type, data.dest_src, data.tag, data.comm, &req);

   // while "no finishing with non-blocking send" then execute another codelet
   while(!mpix_mpiTest(&req))
           swarm_yield();

   // schedules the continuation
   swarm_schedule(NEXT, NEXT_THIS, INPUT, NULL, NULL);

CODELET_IMPL_END;

CODELET_IMPL_BEGIN_NOCANCEL(mpix_recv)

   mpix_str data = *(mpix_str*) (swarm_natP_t) THIS;
   MPI_Request req;
   // non-blocking receiving the data
   MPI_Irecv(data.buf, data.count, data.type, data.dest_src, data.tag, data.comm, &req);
   // while "no finishing with non-blocking recv" then execute another codelet
   while(!mpix_mpiTest(&req))
            swarm_yield();
   // schedules the continuation
   swarm_schedule(NEXT, NEXT_THIS, INPUT, NULL, NULL);

CODELET_IMPL_END;
...
```

Fig 3: Excerpt from the dynax_mpix.h library. This library encapsulates and offers codelets for MPI and SWARM interoperability.

Results

In figure 4 and figure 5, we show the tracing for an synthetic application that the dynax_mpix library. This application schedules send and recv operations using the codelets defined before and also schedules a dummy codelet that consumes cpu without real work done. Figure 4, shows that the the 8 workers mainly execute the dummy codelet (blue label).

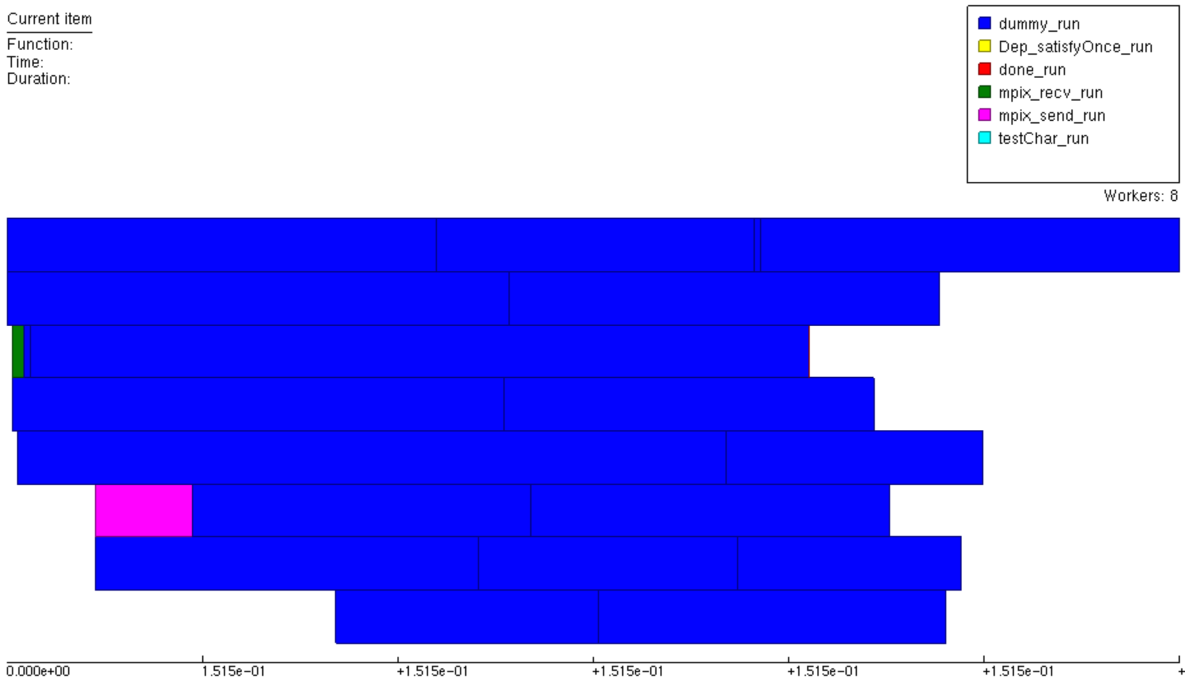


Fig 4: Tracing of the program with all the codelets selected for visualization.

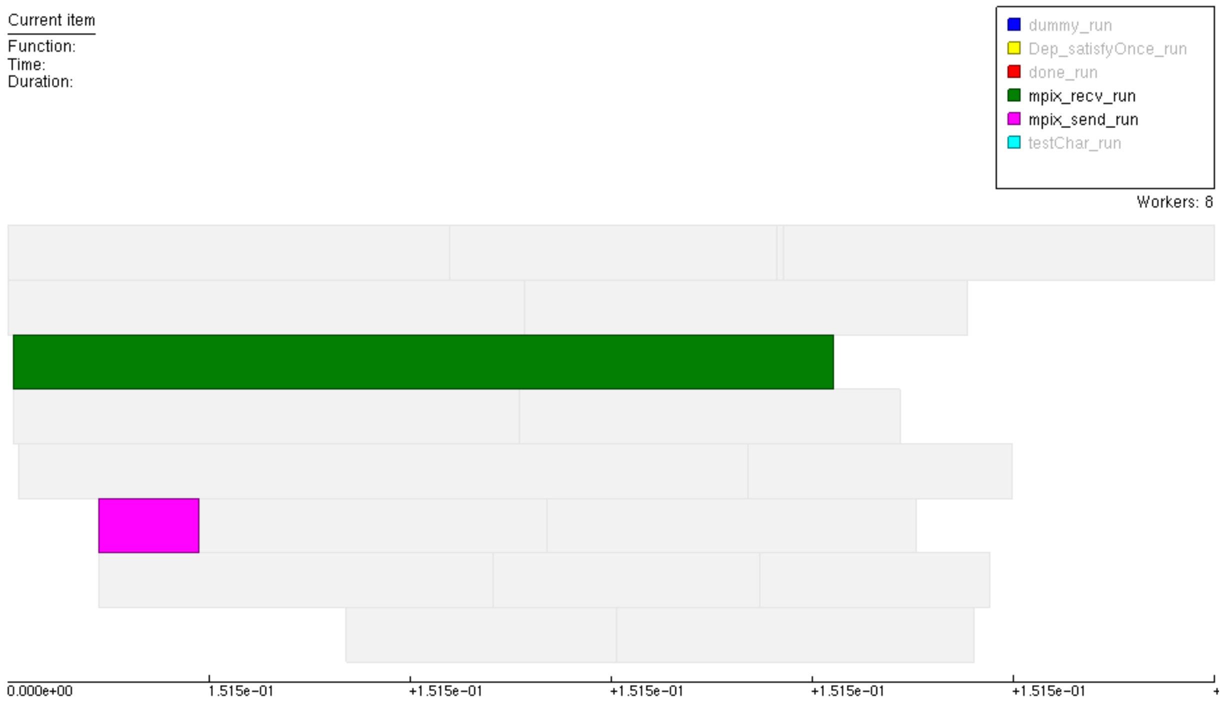


Fig 5: Tracing of the program with just mpix_recv and mpix_send codelets selected for visualization.

In figure 5, we can see the same tracing of the program with just mpix_recv and mpix_send codelets selected for visualization. In here, we can see that mpix_send ran in background the check for completion and allows the runtime thread to schedule other codelets.

Conclusions and Future work

In conclusion, we have demonstrated that SWARM and MPI runtimes can interoperate in a simple application. The examples presented in this work and attached as a .zip file, are simple enough to evaluate the feasibility of the approach and serve as basis to the creation of a more complex benchmarks in the following quarter. Our sample applications shown that MPI calls can be used in a decentralized, continuation-based manner, to provide a fine-grained, low-overhead framework for MPI interoperability with SWARM.

For the dynax_mpix library more work needs to be done to handle errors in the underlying MPI send or recv. For instance, if the message is never sent or it is lost, the current implementation will keep checking indefinitely for the non-blocking call to finish.

References

[1] Message Passing Interface (MPI). High Performance Computing Training. Lawrence Livermore National Laboratory. Retrieved May 4, 2015, from https://computing.llnl.gov/tutorials/mpi

[2] Working with codelets. Programmer's Guide to the SWARM API - SWARM documentation. ET International Inc. Retrieved May 18, 2015, from http://www.etinternational.com/downloads/swarm_docs/swarm/programmers-guide/index.htm

Reservoir Labs

Parallelization to block-structured codes

Our strategy for parallelizing block-structured codes is *not* to take sparse, unstructured-block programs as input and try to parallelize them. Because compilers are intrinsically limited to analyzing static code, such an approach typically forces the compiler to make conservative assumptions on the data. As a result, an excessive amount of data have to be considered aliased, usually impeding any useful level parallelization and optimization.

Instead, we approach the problem at a higher level, where the user provides an unoptimized program and specifies its sparsity properties. The main advantages of this approach is that it greatly reduces the programming effort for the user, and it also simplifies parallelization and locality analysis and optimization for the compiler.

The implementation of this capability can be decomposed in four parts:
- Enabling the user to formulate sparse computations and sparse data in her input program
- Representing sparsity in the parallelization tool (R-Stream), and using the representation to drive specific optimizations
- Generation of sparse (here: block-structured) code
- Runtime support that implements the block-structured abstractions targeted by the generated code. This quarter, we have worked on the implementation of the runtime support for block-structured data structures on a cluster.

An artefact of writing a program in a dense form is that the data structures created may also have to be dense as opposed to sparse. Therefore, it becomes essential that after the compiler has extracted dependence information, the sparse data structures be stored in a compressed form to achieve efficient memory use. Towards that end, we are developing an R-Stream runtime layer functionality to automatically perform array compaction. The runtime layer API has three main functions: rstream_sparse_array_create, rstream_sparse_array_dma_get, and rstream_sparse_array_dma_put which are explained in detail below.

The data space is tiled and arrays are stored as data tiles. Data tile sizes are configurable. One of the main advantages of data tiling for a sparse data structure is that it introduces a degree of control of data localization, which would otherwise be fixed in the case of non-tiled sparse formats. Another advantage is that quick emptiness tests can be performed at the tile level for any subregion of a tile. Also, the risk of contention on metadata is lower since the metadata itself becomes distributed. Furthermore, when data are to be retrieved, the number of indexes that need to be traversed would be potentially lesser.

i) rstream_sparse_array_create
The incoming data structures are inspected and stored in a coordinate format: only non-zero data tiles and their indexes are held in memory.

ii) rstream_sparse_array_dma_get
When the parallelized program accesses data -- to fetch operands needed for computation, the runtime layer performs re-indexing of arrays to retrieve data from compacted data structures.

iii) rstream_sparse_array_dma_put
In addition to providing access to existing data blocks, the runtime layer would have to store newly created non-zero data-tiles. To store the new non-zero blocks, a separate list per array is maintained and the data-tiles are attached to the list. Periodically, when the number of free-standing data blocks reaches a certain proportion of the total number of data-blocks of the array, the array is re-organized to merge the data blocks in the list with the rest of the data blocks.

Compiler support: The other component (first is efficient data storage) of our integrated data and computation efficient mapping of unstructured codes to parallel computers is, to eliminate redundant computations in the output code. Examples of it include, multiplication between a zero and a non-zero block in a sparse matrix-vector or matrix-matrix multiplication. The compiler will perform semantics analysis to derive conditions under which the computation can be skipped. E.g., if the operation is multiplication and one of the operands is a zero, then the result of the operation is known *a priori* to be zero and therefore, the computation can be shorted. For more intricate sequence of program statements, the user can provide *skip* conditions in the form of compiler directives and the R-Stream compiler would process them and insert those conditions in the generated parallelized code.

R-Stream support for PIL parallelization

Glossary:

- HTA: Hierarchically Tiled Arrays, a notation to specify data-parallel computations on arrays and their decomposition
- PIL: Parallel Intermediate Language, the intermediate representation of UIUC's HTA parallelization tool. It is possible to compose programs in PIL directly, hence it also its own language (called PIL as well).

- R-Stream: Reservoir Labs' automatic parallelization tool. Takes sequential C codes that process arrays and produces a parallel version of the codes based on SWARM (other backends are available).

UIUC and Reservoir are collaborating on a hierarchical parallelization path for HTA (and, by extension, PIL) programs, in which a first, coarse-grain parallelization is produced by generating parallel calls to a sequential function. The sequential function is in turn parallelized by R-Stream, this time targeting hardware threads.
Most of the work here is taking place at the interface. The current "contract" with the R-Stream user is that R-Stream parallelizes the computations of a function (or a set of functions). No assumption is made on the caller of the function, which forces R-Stream to preserve the synchronous nature of the function call, as illustrated in Figure 1.

```
#pragma rstream map
float foo(...) {
  // sequential code, parallelized by R-Stream
}
int main() {
    ...
    x = foo(...)
   // Synchronous call: control returns here after foo has completed.
}
```

Figure 1: Calling a function parallelized by R-Stream

The drawback of preserving the synchrony of the call is that the caller thread is waiting while *n*-1 threads are working (*n* being the total number of threads used). Since *n* goes up to hundreds on current nodes, it is acceptable to "waste" a thread.
In the context of the "HTA to R-Stream to SWARM" parallelization path, HTA produces *m* threads that call a sequential function to be parallelized by R-Stream. Once parallelized at both levels, the cost of maintaining a synchronous call at the boundary (the "foo" function in the example) is now that *m* threads are idling. This imposes an artificial limitation to the amount of parallel instances that can be generated from HTA. An extreme case is when *m*>=*n*, in which case this scheme would result in a deadlock. Since it is always wasteful, we proposed an extension of the R-Stream runtime layer for SWARM to support asynchronous calls. Instead of calling foo(), the caller (generated by HTA) initializes an event on which the continuation to the foo() call will depend. The R-Stream runtime ensures that this event is triggered only when all the codelets in foo() are completed.

Reservoir Labs Publications

Sanket Tavarageri, Benoit Meister, Muthu Baskaran, Benoıt Pradelle, Tom Henretty, Athanasios Konstantinidis, Ann Johnson, Richard Lethin. “**Automatic Cluster Parallelization and Minimizing Communication via Selective Data Replication”**
Submitted to HPEC’15 (High Performance Extreme Computing) conference for publication

UIUC

In the previous quarters, we implemented mechanisms to compile the HTA programs for different execution modes to run on SWARM runtime and OpenMP. In the fork-join execution mode, the sequential part of an HTA program is executed by the master codelet, and multiple slave codelets are created whenever an HTA operation is invoked (forking). The master codelet then busy waits for the for all slave codelets to complete (joining) before it goes on executing the next program statement. In this mode, barriers are implicit and they happen with each HTA operation. The style is suitable for bulk synchronous applications with balanced workload.

In contrast, in the SPMD execution mode, the program execution starts with P codelets simultaneously executing the sequential part of an HTA program. When an HTA operation is invoked, each of the P codelets creates one slave codelet to perform the work, and continues only after the slaves finish the parallel work. The initial P codelets corresponds to the definition of “process” in the conventional SPMD model. They do not share data directly. Whenever one requires data owned by another, they need to communicate with point-to-point messages. The communications are implicitly performed by the HTA library implementation and thus hidden from the programmer. The SPMD model is advantageous in that barriers are avoided, so it is better suited for unbalanced workload since a process holding more workload than others will affect only others who depend on its output, but not every process. It can also be easily extended to work on distributed memory machines.

NAS Parallel Benchmarks

We studied the performance results of six NAS Parallel Benchmarks (NPB): EP, IS, FT, MG, CG, and LU. The experiments are conducted on a cluster node with 2 Intel Xeon E5-2690 CPUs (32 threads in total).

Fork-join Execution

For the fork-join mode, both HTA/OpenMP and HTA/SWARM get comparable results and are within 10% difference of the NPB OpenMP version. In some cases, we get better results than the pure-OpenMP version due to different memory layout/tiling effects. The same application code running on SCALE (HTA/SWARM) results in slightly worse performance (5%~10% in average) than HTA/OpenMP. We believe this is because in the case of HTA/OpenMP, when parallel operation is forked, workload is evenly splitted and statically scheduled to slave threads. However, in the case of HTA/SWARM, the master codelet forks by inserting newly created slave codelets into its work queue, and these codelets are not executed until the other available SWARM worker threads steal work from the work queue. A mechanism to directly schedule slave codelets into work queues of the other known free worker threads could improve performance in this situation.

SPMD Execution

Concerning the SPMD mode, comparable results with pure-OpenMP version are observed for both HTA/OpenMP and HTA/SWARM. In general, they are not as good as fork-join results (10%~20% slowdown) because of the need to communicate between "processes" causing extra memory copies on a shared memory machine. While most of the NPB algorithms have balanced load for each thread, they are inherently more suitable for bulk-synchronous execution in the fork-join mode, and relying on the fast barrier synchronization is better than performing point-to-point synchronization which causes extra memory copies.

Cholesky Factorization

In order to understand the execution behavior of applications that are not bulk-synchronous, we studied the dense tiled Cholesky factorization algorithm in different modes. The pseudocode is shown below:

```
for (k = 0; k < nBlocks; k++) {
    DPOTRF(Akk);
    for(i = k + 1; i < nBlocks; i++) { // Column update
        DTRSM(Akk, Aik);
    }
    for(i = k+1; i < nBlocks; i++) {   // Submatrix update
        for(j = k+1; j <= i; j++) {
            if(i == j)
                DSYRK(Aik, Aij);
            else
                DGEMM(Aij, Aik, Ajk);
        }
    }
}
```

To implement this algorithm with HTA, the outer for loop is seen as the sequential part of HTA program. The first inner loop for DTRSM operation is written as a HTA_cmap_h2() function which performs DTRSM operations on the column tiles in parallel. The second inner loop is also written as a HTA_cmap_h3() function that conditionally perform either DSYRK or DGEMM operation in parallel.

We ran the experiment again on a cluster node with 2 Intel Xeon E5-2690 CPUs (32 threads in total). The fork-join mode outperformed SPMD mode in most cases. We know this is because in the fork-join mode, both inner loops fork tasks to be shared among all available workers, and they dynamically fetch new tasks. On the other hand, in the SPMD mode, each process owns statically distributed tiles, and by strictly following “owner-compute” philosophy, idling workers are not aware of other workers’ workload and cannot dynamically steal work from others. The following execution traces demonstrate the difference between the two. We are investigating on how to relax the restriction caused by static tile distribution on a single node with shared memory address space.

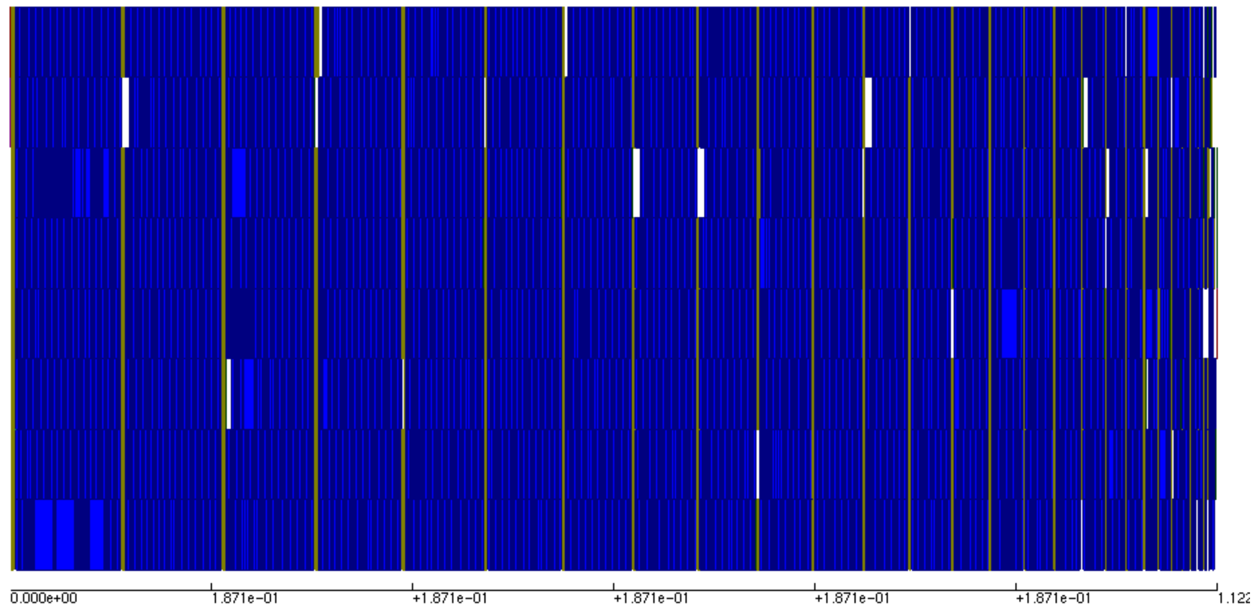


Figure 2: trace of the fork-join execution. The yellow stripes are the DTRSM tasks, and the blue stripes are DSYRK and DGEMM. There are very few places where a worker thread is idle (white).

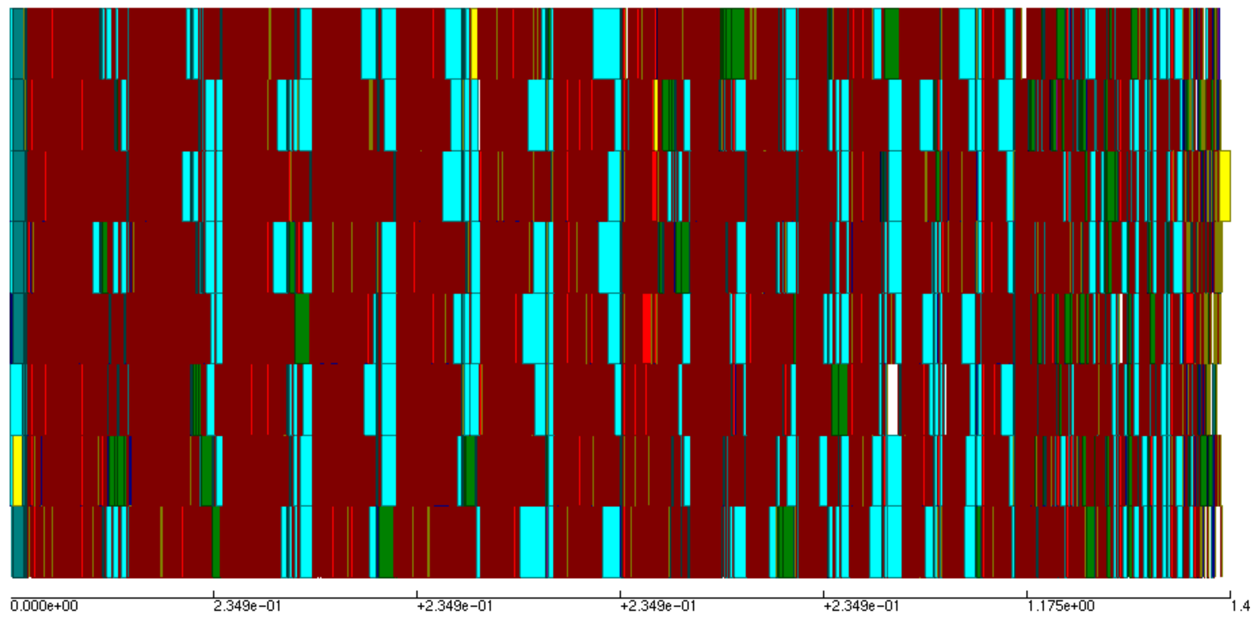


Figure 3: trace of the SPMD execution. The red blocks are DSYRK and DGEMM tasks, and the light blue blocks represent the busy waiting due to data dependence. The green blocks represent sending data. The load imbalance due to static data distribution causes the busy waiting.