High-level Status Summary

Technology	**Description (Institution)**	**Status**
Resilience	Containment Domains in SWARM (ETI)	Near Completed
Application Migration	MPI Interoperability (ETI)	Near Completed
Parallel Language	Evaluation of SPMD mode with NAS Parallel Benchmarks (UIUC)	Completed
Parallel Language & Parallelizing Compiler	Integration of PIL with R-Stream Compiler to generate SCALE code	Near Completed
Parallelizing Compiler	Support for distributed sparse block computations	In progress
Group Locality	Improved resource utilization (PNNL)	In progress

Table of contents:

Summaries of Quarterly Work (Q12)

ETI Work

During this reporting period (Q12: 06/01/2015 - 08/31/2015), 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 (near completed)

Task 10.1: Study MPI interoperability (near complete)

Resilience (containment domains): Task 8.1

In this quarter, we have continued our design studies on our containment domain approach for SWARM-like program execution and programming models as well as their efficient implementation in SWARM. Our major achievements include:

1. We explored the design space allowed within the confines of the containment domain framework within SWARM. Specifically, theses confines included the property of well-behavedness taken from data flow theory and the property of nesting in a method similar to loop constructs.
2. A case study has been used to verify the usefulness of these confines within SWARM. This case study takes two commonly used data flow schemas (merge, and loop operations) and implements the principles of containment domains within SWARM.
3. We performed a back-of-the-envelope analysis of the case study to determine where to inject preservation statements for containment domains.
4. We proposed a path forward to extend the semantics developed in our analysis leveraging the previous work proposed by Gao, et. al. in [TM104].
5. We recently submitted an extended abstract for publication to the Mini-Symposium on Energy and Resilience in Parallel Programming (ERPP 2015) to be held in conjunction with International Conference on Parallel Computing (ParCo 2015). In this work. we propose a resilience scheme that works with codelet-based runtimes and implement a prototype of our containment domain framework within SWARM. Finally, we demonstrate the feasibility of the approach by adapting a Cholesky decomposition to use our framework.

During Q12 Technical exchanges between my group and Mattan's at UT Austin have been continued, and is very helpful as expected in our original proposal.

The results of our resilience work is published as a ETI Technical Report 003. More technical details are included later in the Topics Details section in this report. Also, the references for that topic are mentioned at the end of that sub-section.

MPI Interoperability : Task 10.1

Exascale software will be unable to rely on minimally invasive system interfaces to provide an execution environment and hence 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 continued our proposal on SWARM interoperability with legacy MPI code - as outlined by several prior reports. We have consolidated our ideas and focused our work mainly on MPI+SWARM. to this end, we have extended our case studies beyond those reported in our prior work in Q11' report[1] and the ETI Technical Report 02[2]. Furthermore, we have conducted a comparative study with related work - that have helped us to position our work and identify its uniqueness. To this end, our major achievements in this quarter can be summed up as -

1. We continue our proposal as outlined in the ETI Technical Report 2[3] and consolidate our ideas in ETI Technical Report 004
2. We have extended our benchmarks for SWARM interoperability by adding additional examples for both MPI+SWARM which is inspired by our CODELET SWARM approach and our experience working with OCR[4].
3. We demonstrated the MPI interoperability with the practical example of Matrix Multiplication for the shared memory system. We also compared the execution time results of MPI, MPI+SWARM and SWARM for various configurations which helped us draw important conclusions.
4. We conducted a comparative study of MPI+X work in the field, and focused on the targeted literature that are most relevant with our theme MPI+X. We found that other teams with the exascale frameworks are also working towards common goal of

[1] "DynAx Quater 12 Report", https://xstackwiki.modelado.org/images/d/d7/DynAX-XStackQ11Report.pdf, June 1 , 2015.
[2] Sergio Pino, Guang Gao, "Legacy MPI Codes and its interoperability with fine grain task-parallel runtime systems for Exascale", https://xstackwiki.modelado.org/images/8/82/ETITechnicalReport02.pdf , June 1, 2015.
[3] Sergio Pino, Guang Gao, "Legacy MPI Codes and its interoperability with fine grain task-parallel runtime systems for Exascale", https://xstackwiki.modelado.org/images/8/82/ETITechnicalReport02.pdf , June 1, 2015.
[4] Open Community Runtime, https://xstCkwiki.modelado.org/Open_Community_Runtime

supporting interoperability. We composed and completed a new ETI technical report on the subject that includes relevant background and related work sections to summarize this study.

5. We have completed and published our work titled “Legacy MPI Codes and its interoperability with fine grain task-parallel runtime systems for Exascale” as ETI Technical Report 004

More technical details are included later in the Topics Details section in this report. Also, the references for that topic are mentioned at the end of that sub-section.

Work for No Cost Extension Period

ETI is expected to complete ETI’s portion of the Dynax project and burn out the ETI portion of the funds by August 31, 2015. ETI has requested additional 4 months to complete the documentation.

Please refer to appendix A for our correspondence with Dr. Sachs

Reservoir Work

During this reporting period (Q12: 06/01/2015 - 08/31/2015), Reservoir has been working on 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 the automatic parallelization of block-structured computations, a class of unstructured codes (Task 3.4). As presented last quarter, this relies on a smart layer of logical DMAs. We are presenting the approach and current implementation in the *Parallelization of block-structured codes* section below. The design enables the easy definition of data placement, helping us (and any user) contribute to Task 2.4.

We have also supported UIUC’s effort in targeting R-Stream (Task 5.7). A short description of the main issues encountered there and how they are addressed is presented in the *Supporting PIL as an R-Stream frontend* section below.

References for each topic are contained in their respective subsection.

Work for No Cost Extension Period

A two-months no-cost extension (NCE) to the DynAX contract has been requested. The work we will perform during that period of time will span tasks 2.4, 3.4 and also aminor item in 5.7.

Our main objective is to use the NCE to bring the R-Stream prototype runtime, backend and mapper for x86 clusters based on SWARM to a level of maturity that will enable automatic parallelization to codelets on clusters. We also plan to use R-Stream to automatically parallelize the core computation of the ExMatEx CoSP2 proxy app to clusters. CoSP2 represents a sparse linear algebra parallel algorithm for calculating the density matrix in electronic structure theory. We will produce a self-contained report to relate our work and results.

Please refer to appendix B for our correspondence with Dr. Sachs regarding the NCE.

UIUC Work

In Quarter 12, the UIUC team has completed the last two remaining tasks:

Task 5.3: Evaluation of the PIL implementation and API

Task 5.7: SCALE and R-Stream code generation

Performance Evaluation of the PIL implementation and API

We have reported the performance evaluation results of NAS parallel benchmark executing in SPMD mode in the last quarter, and we have also implemented a Cholesky factorization application to evaluate the benefits of asynchronous execution in SPMD. We found that when the SPMD mode is used in Cholesky factorization, the static distribution of data tiles results in load imbalance and bad scheduling, causing CPUs to be idle. In this quarter we experimented enabling nested parallelism to SPMD PIL in hope to solve this problem. The data tiles are still statically distributed to codelets representing processes, but adding an extra level of parallelization within each process allows processes to create and share finer-grained codelets when they are in the same memory address space. We present the mechanism and the results in the topic details section.

SCALE and R-Stream code generation

In the last quarter we have encountered an issue that requires changing the interface between the PIL side and R-Stream side of the code. In the original design, when PIL makes a call to the R-Stream optimized function, the PIL codelet making the call waits for the function to complete and does not release SWARM worker threads before the R-Stream function returns. However the R-Stream compiled function is a collection of codelets and they require freee SWARM worker threads to start running. When the number of PIL codelets exceeds the number of SWARM threads, the program will deadlock.

Our solution is to change the interface between PIL and SWARM. When the PIL codelet makes a call to the R-Stream function, it passes along a callback event to the R-Stream function. The call acts like an asynchronous function invocation and returns immediately. The PIL codelet then creates a continuation codelet, which depends on the callback event from R-Stream. It

then terminates itself to release the SWARM worker thread it occupies. In this way, the deadlock is prevented.

At the time the report is written, there is still a bug on the R-Stream side which causes deadlock sometimes. The Reservoir team is trying to figure out the problem. We will conduct experiments to show the performance improvement from the R-Stream optimization once the problem is solved and we plan to include the results in the Y3 report.

Work for No Cost Extension Period

We have received no cost extension until December 31, 2015 to subcontract that the university of Illinois has with ET international. We are requesting this extension to further investigate how applications written in HTA notation can run efficiently on the SWARM runtime system. We expect that by the end of this period we will have applications that could better demonstrate the scheduling ability of the SWARM runtime system.
Please refer to appendix C for our correspondence with Dr. Sachs

PNNL Work

Task 9 : Explore and evaluate the Power Efficient Data abstraction Layer (PEDAL) methodologies on fine grain runtime systems

During the last quarter, PNNL continued to develop the data restructuring methodology of our Group Locality framework. Under this methodology, the access patterns (represented under a Polyhedral framework) from both an actor (e.g. a thread) and a group of actors is taken into consideration when bringing and accessing the affected data structures. We start with a hierarchical tiled code for which data transformations are applied at each level to improve the data residence. The main components of this methodology include a collaborative data restructuring for group reuse and a low overhead transformation technique that exploits locality. We used an exemplar many core architecture, Tilera TileGX, to show improvements over optimized OpenMP code: improvements of up to 31% increase in GFLOPS; and even improving on our own previous work (the fine grained tiling techniques) up to 15% for selected kernels.

More technical details are included later in the Topics Details section in this report. Also, the references for that topic are mentioned at the end of that sub-section.

Work for No Cost Extension Period

Under NCTE, we would plan to use the additional time to complete the Tasks 9.1 and 9.2 as specified in the DyNAX Statement of Work. For Task 9.1, this includes (1) a comparative study of the Architected Composite Data Types (ACDT) framework developed under the program on

SWARM vs. an OCR implementation as part of measuring the overall impact of this technique across these runtimes; (2) the completion of the documentation with these new findings. For 9.2, this includes the completion of the documentation of the Group Locality (GL) compiler and the results of the applications that exercise it.

Topic Detail:

As mentioned earlier, this section includes more details on the work and results listed by each institution.

ETI:

The ETI work is divided in the two sub topics. They are described in details below -

Resilience (containment domains): Tasks 8.1

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 (SWift Adaptive Runtime Machine), 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 [2]. 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 leverage ideas from containment domain research performed by Mattan Erez and his team at University of Texas at Austin[4]. 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.

SWARM is a codelet-based runtime created at ETI [11]. 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.

The work that we found most related to ours are the various MPI based approaches to failure mitigation. Some of these are ad-hoc without coordination and others provide a more coordinated and holistic approach. They provide fault-awareness at the user level in some form much like our SWARM CD APIs. We discuss this work and other related work in detail in our related work section.

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 adapt a Cholesky decomposition program written in SWARM to use this API, showing that the 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

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 [9]. 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 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.

Containment Domains in SWARM

We have developed a containment domain API as a 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 include data preservation, user-defined fault detection functions, and re-execution of failed body functions. This feature set is realized by implementing a number of functions as follows:

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

`swarm_ContainmentDomain_begin(THIS, body, body ctxt, check, check cxt, done, done ctxt):`
Begin execution of the current containment domain denoted by THIS by scheduling the codelet denoted by body ctxt. When the codelet finishes execution, the codelet denoted by check ctxt is scheduled to verify results. If the result of the execution is TRUE then the codelet denoted by done cxt is scheduled.

`swarm_ContainmentDomain_preserve(THIS, data, length, id, type):`
In the containment domain denoted by 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 the first execution, data is preserved normally. On subsequent executions, data is copied in reverse (i.e. from the internal preservation into the data pointer). The CD in which the data is preserved is denoted by type. This can either be the currently activated CD or the parent CD.

`swarm_ContainmentDomain_finish(THIS):`
Close the current containment domain denoted by THIS, discard any preserved data, and make the parent domain active.

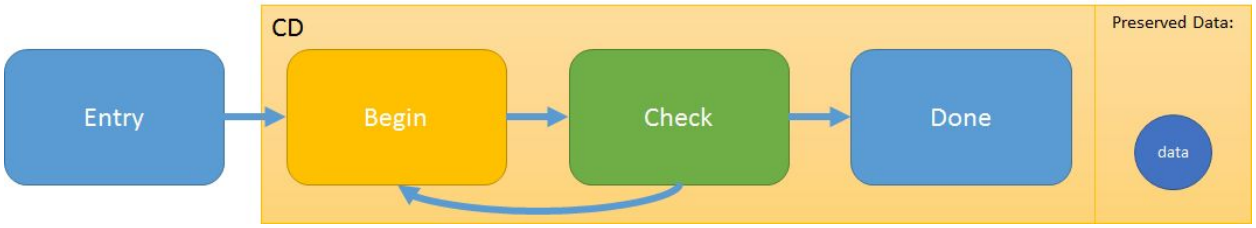

Figure 1. Simple Containment Domains Graph

```
#include <swarm/ContainmentDomain.h>
#include <eti/swarm_convenience.h>
#include <stdio.h>
#include <stdlib.h>

CODELET_DECL(entry); CODELET_DECL(begin);
CODELET_DECL(check); CODELET_DECL(done);

swarm_ContainmentDomain_t cd;

int gA = 17; int gB = 100;

typedef struct
{
        int *A; int *B; int *C;
} mult_t;

CODELET_IMPL_BEGIN_NOCANCEL(entry)
        int *C = malloc(sizeof(int));
        mult_t *ctxt = malloc(sizeof(mult_t));
        ctxt->A = &gA;
        ctxt->B = &gB;
        ctxt->C = C;
        swarm_ContainmentDomain_init(&cd);
        swarm_ContainmentDomain_begin(&cd, &CODELET(begin), ctxt,
        &CODELET(check), ctxt, &CODELET(done), ctxt);
CODELET_IMPL_END;

CODELET_IMPL_BEGIN_NOCANCEL(begin)
        mult_t *ctxt = THIS;
        swarm_ContainmentDomain_preserve(&cd, &gA, sizeof(int), 0,
        swarm_CONTAINMENTDOMAIN_COPY);
        swarm_ContainmentDomain_preserve(&cd, &gB, sizeof(int), 1,
        swarm_CONTAINMENTDOMAIN_COPY);
        *ctxt->C = *ctxt->A * *ctxt->B;
        swarm_dispatch(NEXT, NEXT_THIS, NULL, NULL, NULL);
CODELET_IMPL_END;

CODELET_IMPL_BEGIN_NOCANCEL(check)
        mult_t *ctxt = THIS;
        swarm_bool_t success = (*ctxt->C == *ctxt->A * *ctxt->B);
        swarm_dispatch(NEXT, NEXT_THIS, (void*)success, NULL, NULL);
        CODELET_IMPL_END;
        CODELET_IMPL_BEGIN_NOCANCEL(done)
        mult_t *ctxt = THIS;
        printf("done, result = %d\n", *ctxt->C);
        swarm_shutdownRuntime(NULL);
CODELET_IMPL_END;

int main() {
        return swarm_posix_enterRuntime(NULL, &CODELET(entry), NULL, NULL);
}
```

Figure 2. Simple Containment Domains Code

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. Figure 2 shows the accompanying code for the graph.

Figure 3.

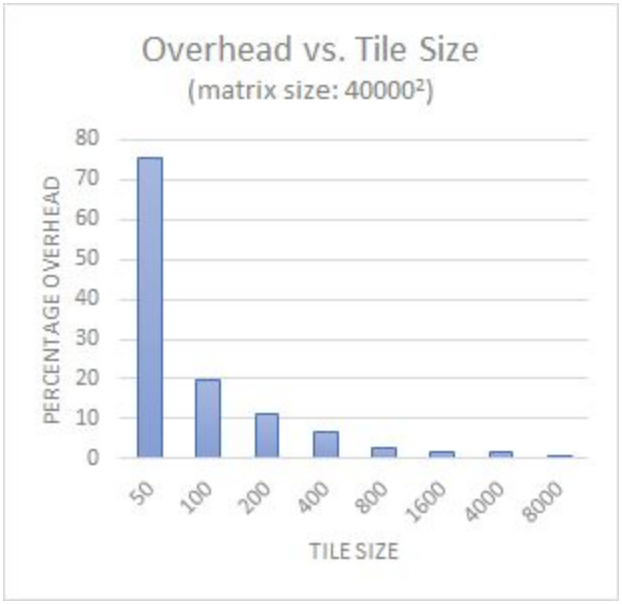

Figure 4.

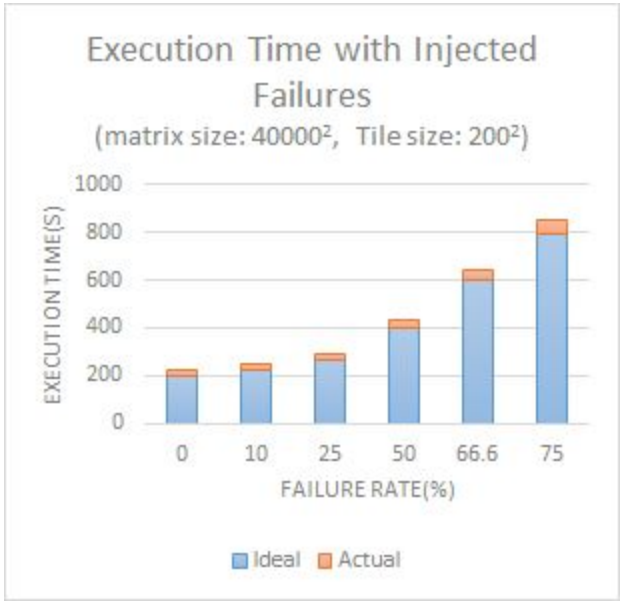

Figure 5.

Experimental Results

We evaluate our approach through three primary means: feasibility, efficiency, and resilience and make a number of key observations. To show that our prototype implementation has sufficient functionality, we instrument a Cholesky decomposition program in SWARM to use containment domains. For the experiments, the program was run on a dual-processor Intel Xeon system, using 12 threads. The workload sizes were confirmed to not exhaust the physical memory of the machine. Though we found insignificant variance between runs, we have averaged all times over 5 program runs due to the natural variation in run time due to extraneous system factors (such as scheduling differences).

Firstly, through the implementation of our framework in SWARM, and a working Cholesky application using said framework, we observe that it is feasible to adapt a codelet-based application to use containment domains. Secondly, Our implementation shows very low overhead.
Figure 3 shows the execution time for various tile sizes executing Cholesky of size 40000x40000. *Base* denotes the Cholesky kernel runtime without containment domains or data preservation. *CD* denotes the time spent within CD related API calls without preservation. *Preserve* denotes the time spent preserving data. One can see that the API itself adds negligible overhead and that the only significant overhead comes from actual preservation of data. Figure 4 shows the total overhead (preservation + API calls) relative to the base cholesky code without containment domains. The trends indicate that as tile sizes (workload per codelet) increase the overhead is mitigated and eventually becomes negligible. This trend is unsurprising given that the runtime overhead per API call is relatively constant and as the tile size increases less and increasingly larger data sized preservation calls are made to the runtime. Additionally, the cost of preservation (i.e. for data movement) increases at a much slower rate than the cost of the Cholesky computation as tile sizes are increased. Overall, this trend shows that there is a sweet spot in terms of granularity and that proper decomposition is key to mitigate preservation overheads and maximize performance.

Figure 5 shows simulated injected failures that result in codelet re-execution within the SWARM framework. The *idealized* case shows the projected base execution time without CD or preservation overhead for a Cholesky of size 40000x40000 and tile size of 200x200. The results are obtained by taking the execution time without failures and extrapolating for various failure rates where some total percentage of the codelets fail overall during the run (i.e. a failure rate of 75% would result in the runtime taking 4 times as long). The *actual* case shows the execution times actually obtained from running SWARM with injected failures. We note that there is around ~11% overhead without failures and that this overhead decreases to ~6% at a failure rate of 75%. This is because some allocation and API overheads are not present upon re-execution. Trend wise, we note that maximal overhead occurs when faults are not present in the system. We additionally note that the execution time follows reasonably well to that of the idealized case and that it would be possible to project with reasonable accuracy the execution time in the event of failures using data from actual runs without failure.

Related Work

As systems have become larger and larger, reliability has become an exceedingly active area of research over the last decade. Containment domains provide a flexible mechanism to ensure reliability at varying granularities. Additionally, CDs provide a novel bridge between reliability and programming models that facilitates reliability-aware or fault-aware application development, thus, giving the application programmer the ability to dynamically provide the runtime with the information needed to ensure reliability at the right granularities in terms of costs and benefits.

Much work in the field focuses on providing similar preservation and restoration capabilities, but generally lacks a programmer centric view of reliability. Some examples include global or local checkpointing [15] approaches, combining local and global checkpointing [21,18], as well as, multi-level [16,6] and hierarchical checkpointing [20]. There has been work to incorporate fault tolerance at the programmer level into MPI (Message Passing Interface) in the form of user-level failure mitigation (ULFM) [1]. ULFM provides mechanisms to inform and allow users to handle failed MPI processes on a per operation basis. Another approach, FA-MPI [10] seeks to provide fault-awareness to applications without significantly impacting the MPI interface. Similar to CDs, FAMPI incorporates transactional semantics to provide a less ad-hoc approach to reliability.

Still other work focuses on enhancing reliability through specialized scheduling techniques. Static approaches use offline analysis to provide fault tolerance scheduling for a fixed number of faults [12,7], however, these lack the flexibility to adapt to changing system resources or workloads. Dynamic approaches use system level monitoring to adapt to faults that occur during execution. These can put into several subcategories including system reconfiguration [2], workload assignment [8,4,9], or providing automatic or semi-automatic replication of tasks[17].

Containment domains address several weaknesses to prior forms for fault-tolerance. One fundamental difference between CDs and generic checkpointing is that CDs are not interval or time dependent [19]. This gives a flexibility and control lacking in prior fault tolerant schemes by allowing the programmer and system software to tune the location and method of preservation and the recovery to a desired level of reliability while also maximizing performance of the system. Furthermore due to the transactional characteristics of CDs, they are not susceptible to domino effect [13] that can cause full system rollback in the event of faults.

Acknowledgement

This material is based upon work supported by the Department of Energy [Office of Science] under Award Number DE-SC0008716.

Conclusion

In conclusion, we demonstrate 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 show that this is an approach worth pursuing. Due to the Cholesky program's very decentralized call graph, it was not feasible to add nested containment domains. Another example would provide additional insight. For future work, we plan to implement a Self-Consistent Field (SCF) program to evaluate nested containment domains.

References

[1] W. Bland, K. Raffenetti, and P. Balaji. Simplifying the recovery model of user-level failure mitigation. In Proceedings of the 2014 Workshop on Exascale MPI, ExaMPI '14, pages 20–25, Piscataway, NJ, USA, 2014. IEEE Press.

[2] C. Bolchini, A. Miele, and D. Sciuto. An adaptive approach for online fault management in manycore architectures. In Design, Automation Test in Europe Conference Exhibition (DATE), 2012, pages 1429–1432, March 2012.

[3] F. Cappello, A. Geist, W. Gropp, S. Kale, B. Kramer, and M. Snir. Toward exascale resilience: 2014 update. Supercomputing frontiers and innovations, 1(1), 2014.

[4] T. Chantem, Y. Xiang, X. Hu, and R. P. Dick. Enhancing multicore reliability through wear compensation in online assignment and scheduling. In Design, Automation Test in Europe Conference Exhibition (DATE), 2013, pages 1373–1378, March 2013.

[5] J. Chung, I. Lee, M. Sullivan, J. H. Ryoo, D.W. Kim, D. H. Yoon, L. Kaplan, and M. Erez. Containment domains: A scalable, efficient, and flexible resilience scheme for exascale systems. In the Proceedings of SC12, November 2012.

[6] S. Di, L. Bautista-Gomez, and F. Cappello. Optimization of a multilevel checkpoint model with uncertain execution scales. In Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '14, pages 907–918, Piscataway, NJ, USA, 2014. IEEE Press.

[7] L. Duque and C. Yang. Guiding fault-driven adaption in multicore systems through a reliability-aware static task schedule. In Design Automation Conference (ASP-DC), 2015 20th Asia and South Pacific, pages 612–617, Jan 2015.

[8] L. A. R. Duque, J. M. M. Diaz, and C. Yang. Improving mpsoc reliability through adapting runtime task schedule based on time-correlated fault behavior. In Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE '15, pages 818–823, San Jose, CA, USA, 2015. EDA Consortium.

[9] N. Fisher, J.-J. Chen, S. Wang, and L. Thiele. Thermal-aware global real-time scheduling on multicore systems. In Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE, pages 131–140, April 2009.

[10] A. Hassani, A. Skjellum, and R. Brightwell. Design and evaluation of fa-mpi, a transactional resilience scheme for non-blocking mpi. In Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on, pages 750–755, June 2014.

[11] C. Inc. Containment domains api. lph.ece.utexas.edu/public/CDs, April 2012.

[12] V. Izosimov, P. Pop, P. Eles, and Z. Peng. Scheduling and optimization of fault-tolerant embedded systems with transparency/performance trade-offs. ACM Trans. Embed. Comput. Syst., 11(3):61:1–61:35, Sept. 2012.

[13] R. Koo and S. Toueg. Checkpointing and rollback-recovery for distributed systems. Software Engineering, IEEE Transactions on, SE-13(1):23–31, Jan 1987.
[13] Z. Luo. Checkpointing for workflow recovery. In Proceedings of the 38th Annual on Southeast Regional Conference, ACM-SE 38, pages 79–80, New York, NY, USA, 2000. ACM.
[14] C. Lauderdale and R. Khan. Towards a codelet-based runtime for exascale computing: Position paper. In Proceedings of the 2Nd International Workshop on Adaptive Self-Tuning Computing Systems for the Exaflop Era, EXADAPT '12, pages 21–26, New York, NY, USA, 2012. ACM.
[15] Z. Luo. Checkpointing for workflow recovery. In Proceedings of the 38th Annual on Southeast Regional Conference, ACM-SE 38, pages 79–80, New York, NY, USA, 2000. ACM.
[16] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski. Design, modeling, and evaluation of a scalable multi-level checkpointing system. In Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, SC '10, pages 1–11, Washington, DC, USA, 2010. IEEE Computer Society.
[17] X. Ni, E. Meneses, N. Jain, and L. V. Kal´e. ACR: Automatic checkpoint/restart for soft and hard error protection. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, pages 7:1–7:12, New York, NY, USA, 2013. ACM.
[18] B. Panda and S. Das. Performance evaluation of a two level error recovery scheme for distributed systems. In S. Das and S. Bhattacharya, editors, Distributed Computing, volume 2571 of Lecture Notes in Computer Science, pages 88–97. Springer Berlin Heidelberg, 2002.
[19] J. S. Plank and M. G. Thomason. Processor allocation and checkpoint interval selection in cluster computing systems. Journal of Parallel and Distributed Computing, 61(11):1570 – 1590, 2001.
[20] T. Ropars, T. V. Martsinkevich, A. Guermouche, A. Schiper, and F. Cappello. Spbc: Leveraging the characteristics of mpi hpc applications for scalable checkpointing. In Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, SC '13, pages 8:1–8:12, New York, NY, USA, 2013. ACM.
[21] N. H. Vaidya. A case for two-level distributed recovery schemes. In Proceedings of the 1995 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS '95/PERFORMANCE '95, pages 64–73, New York, NY, USA, 1995. ACM.

SWARM and MPI Interoperability: Task 10.1

We expanded our work on SWARM interoperability with MPI by further exploring the 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.

In addition to the basic examples we had in ETI Technical Report 002 [1] , we continued our work on interoperability by demonstrations on examples like matrix multiplication. We have technical publication in preparation titled "Legacy MPI Codes and its interoperability with fine grain task-parallel runtime systems for Exascale". We focused on MPI+SWARM approach for this quarter and worked on extensive experimental analysis of our interoperability approach. We explored the related work in the domain in details with focus on interoperability of MPI with exa scale frameworks.
The results of work are summarized in the following report -

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.

Introduction

The DynAX project[2] 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[3], 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.

The work that we found most related to ours is by the OCR team[4] where they propose MPI-Lite on OCR. Also, The XPRESS team[5] has similar goals to achieve interoperability and migration from the legacy codes. We discuss this work and other related work in detail in our related work section.

Background

HPC programming is currently dominated by either a flat model with MPI Cross nodes as well as cores within a node, or a hybrid model with MPI Cross the nodes and OpenMP shared memory parallelism across the cores in a node. For the past 20 years most HPC systems were roughly isomorphic. However, the economics of HPC has lead industry to use the best, lowest-cost commodity parts whenever possible. Technology is driving designs towards slower cores with more parallelism to make up the performance slack.

The exa-scale frameworks have already adapted to these changing winds and the future exascale frameworks are built upon the presumption that the HPC systems will comprise of many-core sockets and GPU accelerators will impose increasingly difficult challenges in programming, efficiency, heterogeneity, and scalability for exascale computing.

However, 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. The interoperability of task parallel execution models with legacy codes can be characterized as -

- MPI+SWARM: An MPI program with SWARM calls added
- SWARM+MPI: A SWARM program with MPI calls added
- Codelet MPI: Creating an MPI compatibility layer in SWARM
- Porting MPI to SWARM: Fully rewriting an MPI program in SWARM

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.

Example

The following represents a minimal example to demonstrate this approach:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <eti/swarm_convenience.h>
#include <mpi.h>

CODELET_DECL(startup);
CODELET_DECL(hello);
CODELET_DECL(world);
CODELET_DECL(done);

void main() {
    int id, p;

    MPI_Init(&argc, &argv);
    MPI_Comm_size( MPI_COMM_WORLD, &p);
    MPI_Comm_rank( MPI_COMM_WORLD, &id);

    //Every process prints a hello using swarm
    MPI_Barrier(MPI_COMM_WORLD);

    if (0 != swarm_posix_enterRuntime(NULL, &CODELET(startup), NULL, NULL))
        printf("Error on pid%d\n", id);

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}
```

This example shows how the MPI hello world can be modified to perform the intra-node node computation for hello world in SWARM instead of the original one (printf("Hello, world! pid %d\n", id);).

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_Send and the other to perform asynchronous MPI_Recv. 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.

```
...
/*****************************************************************
 * Declarations for MPIX
 *****************************************************************/

//  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.

Example

The following represents a minimal example to demonstrate the use of the dynax_mpix.h library:

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <eti/swarm_convenience.h>
#include "dynax_mpix.h"

// Declare codelets
CODELET_DECL(testChar);
CODELET_DECL(dummy);
CODELET_DECL(done);

typedef struct {
    int rank;
    int numtasks;
} info;
static int tag=1;

void main (int argc, char *argv[]) {
    int numtasks, rank;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);

    info data;
    data.rank = rank;
    data.numtasks = numtasks;

    swarm_posix_enterRuntime(NULL,
                &CODELET(testChar),
                (void*) &data,
                NULL);
    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}
```

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

    char* msg = (char*) malloc(sizeof(char));
    *msg = 'x';
    // info for sending
    mpix_str* send_str = (mpix_str*) malloc(sizeof(mpix_str));
    send_str->rank = data->rank;
    send_str->buf = msg;
    send_str->count = 1;
    send_str->type = MPI_CHAR;
    send_str->dest_src = (data->rank + 1) % data->numtasks;
    send_str->tag = data->rank;
    send_str->comm = MPI_COMM_WORLD;

    // info for recv
    msg = (char*) malloc(sizeof(char));
    mpix_str* recv_str = (mpix_str*) malloc(sizeof(mpix_str));
    recv_str->rank = data->rank;
    recv_str->buf = msg;
    recv_str->count = 1;
    recv_str->type = MPI_CHAR;
    recv_str->dest_src = (data->rank + data->numtasks - 1) % data->numtasks;
    recv_str->tag = (data->rank + data->numtasks - 1) % data->numtasks;
    recv_str->comm = MPI_COMM_WORLD;

    static swarm_Dep_t dep = swarm_Dep_INITIALIZER(3,
                &CODELET(done),
                NULL,
                NULL);

    swarm_schedule(&CODELET(dummy), &dep, NULL, NULL, NULL);
    swarm_schedule(&CODELET(mpix_send), (void*) send_str, NULL,
       &swarm_Dep_satisfyOnce_CODELET, &dep);
    swarm_schedule(&CODELET(mpix_recv), (void*) recv_str, NULL,
       &swarm_Dep_satisfyOnce_CODELET, &dep);

CODELET_IMPL_END;

CODELET_IMPL_BEGIN_NOCANCEL(dummy)
    swarm_Dep_t* dep_ptr = (swarm_Dep_t*) (swarm_natP_t) THIS;

    int rem;
    if ((rem = swarm_Dep_getNrRemaining(dep_ptr)) > 1) {
         swarm_schedule(&CODELET(dummy), THIS, NULL, NULL, NULL);
         swarm_spin(1000000);
    } else if (rem == 1)
         swarm_Dep_satisfyOnce(dep_ptr);

CODELET_IMPL_END;

CODELET_IMPL_BEGIN_NOCANCEL(done)
    int rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    printf("pid: %d, Shutting down swarm\n", rank);
    swarm_shutdownRuntime(NULL);
CODELET_IMPL_END;
```

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).

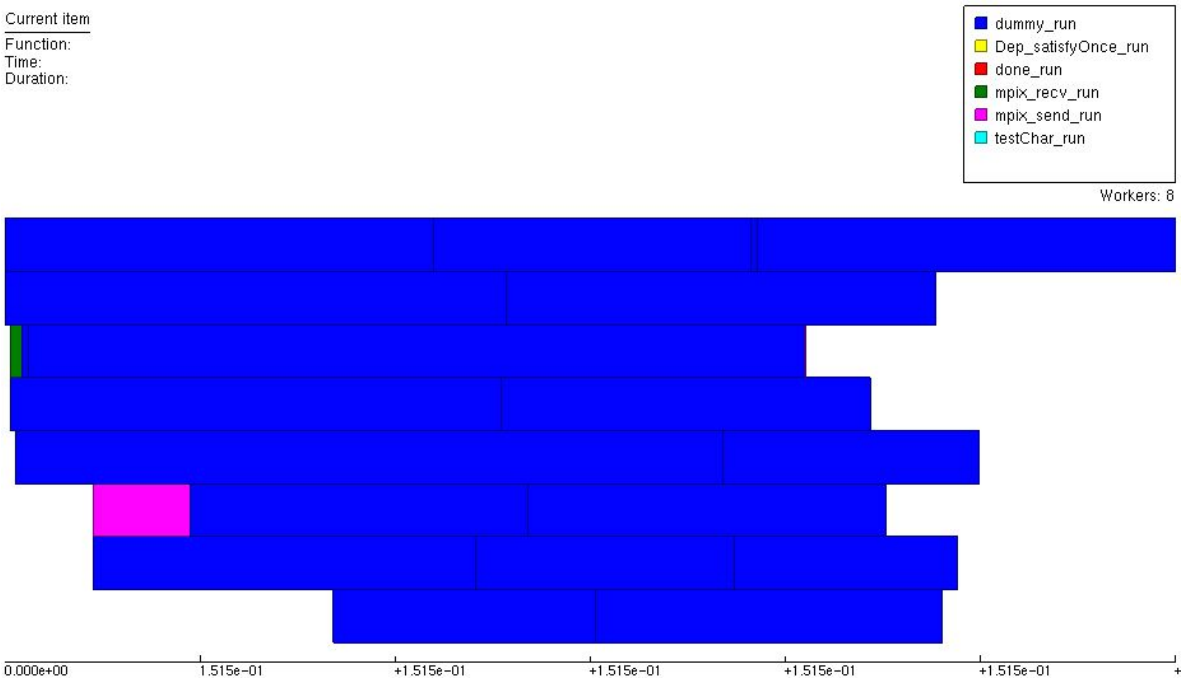

Fig 4: trCing of the program with all the codelets selected for visualization.

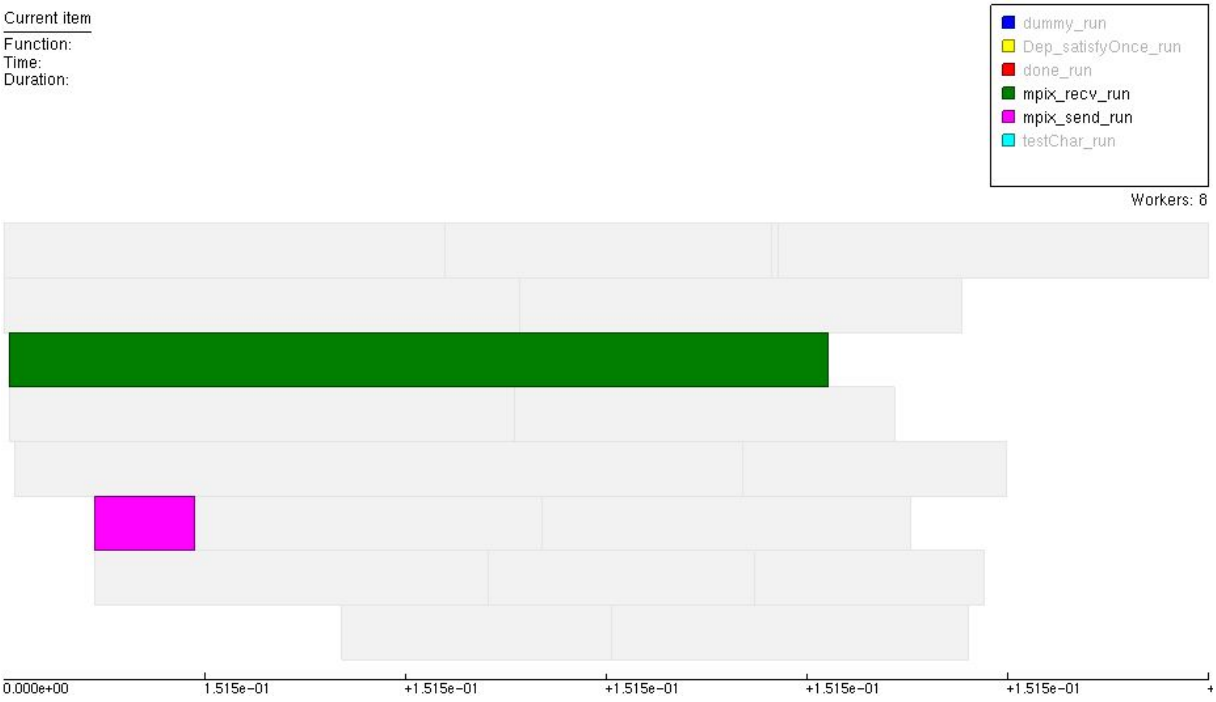

Fig 5: tracing of the program with just mpix_recv and mpix_send codelets selected for visualization.

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.

Experimental Results

This section describes setup of experiments conducted for this paper We decided to use matrix multiplication as test case to demonstrate interoperability between SWARM and MPI. This is mainly because matrix multiplication by nature can be easily parallelized and scaling can be shown clearly. We will be sticking to MPI+SWARM approach for the scope of this report. Also, we limited our matrices to square matrix and all element data types to double.

The goal of the interoperability study is to demonstrate progressive transition from legacy MPI code to exa-scale execution models like SWARM. To demonstrate this transition, we took legacy MPI code for matrix multiplication and replaced blocking MPI sends and receives by non-blocking SWARM calls as naïve approach to interoperability. Then, we also developed optimized matrix multiplication code and compared results of these three approaches. We conducted our experiments on a single node, shared memory system.

Experimental Setup
We evaluated our test studies on a 6 core compute node containing Intel Xenon X5650 processor clocked at 2.67GHz. The node is equipped with 96GBs of DDR3-1333Mhz system memory. Our test bed runs Linux version 3.2.0-77 with Ubuntu 12.04 LTS. Kernels are compiled with GCC version 4.6.3 with –O3.

Results
In this sub-section, we present the experimental results for matrix multiplication. We will compare 3 approaches – MPI, MPI+SWARM and SWARM results.

Figure 1 shows the scaling results for the legacy MPI code for matrix multiplication. Here, we can see three data lines for N=6,12 and 24, where N is number of processes. As we can see from the results, at lower N, the execution times are higher as the size of the matrix is increasing. This is mainly because resources are underutilized because we have restricted N to lower values. Similarly, we can also see that increasing N beyond certain threshold is also not useful as it overwhelms the system.

Figure 2 shows how the execution time scales for MPI+SWARM with increase in matrix size for N=2,4 and 8, where N is number of processes. Here, we can clearly see the underutilization of system resources are lower process count. In fact, this underutilization is amplified with MPI+SWARM. For N=4, we can see that system is well balances and scales steadily. However, at N=8, we were not able to get scaling results for matrix sizes greater than 800. This is mainly because of overutilization and fails in mpi sends and receives.

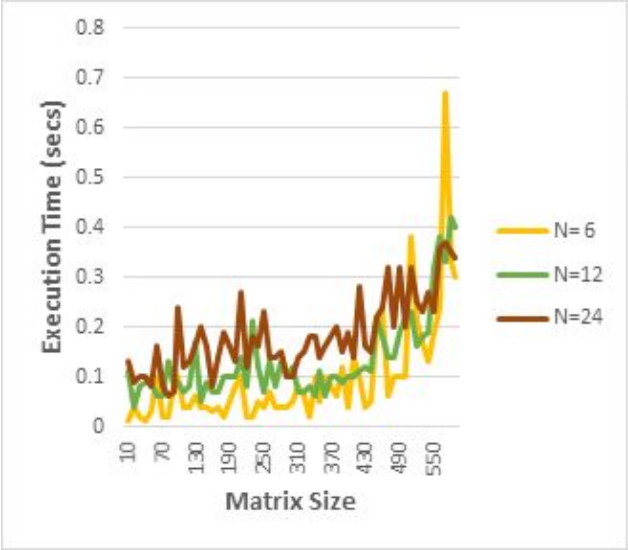

Figure 6 : MPI

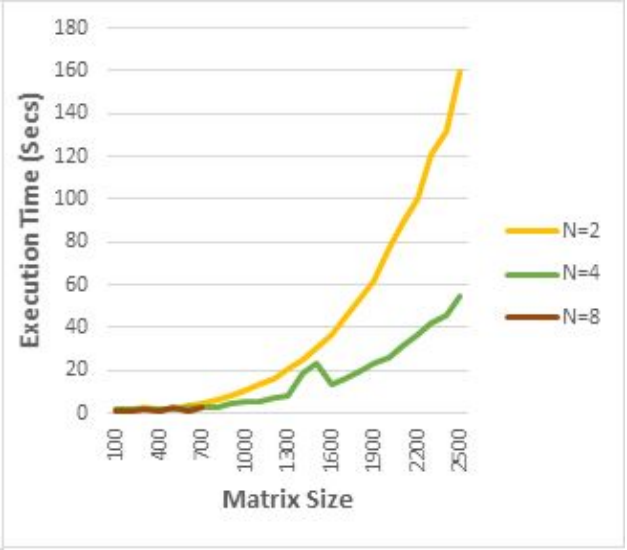

Figure 7 : MPI+SWARM

The figure 3 gives the comparative picture of execution times of all three MPI, MPI+SWARM and SWARM. To get this picture, We keep the number of processes constant. In this case, we keep N=4. The figure clearly shows that naïve changes to legacy MPI code can yield performance benefits. It also shows that optimized SWRM code has undoubtable performance benefits. The figure shows MPI scaling only upto matrix size 600 because legacy MPI implementation uses static allocation for matrices and as we increase number of processes, the limitations on stack size cause failure.

Figure 3 : MPI vs MPI+SWARM vs SWARM Scaling

Future Work

The current experiments clearly show that interoperability of MPI and SWAM is practical and naïve implementation also yields performance gains. Interoperability study can be further extended and we can throw light on many other aspects of it which we could not due to limited time span of the study. In this section, we will cover some of those areas in which can be improved.

Current implementation of MPI version of matrix Multiplication uses static allocation and hence it limits the maximum size of matrix due to per thread stack size. We would like to extend current implementation for dynamic allocation. This will yield to MPI scaling to huge matrix sizes and then the comparison can yield interesting insights.

We were able to get our shared memory matrix multiplication example working for multiple nodes. However, due to node configuration issues we were not able to run the required spectrum of experiments and get the satisfactory results in the given time window. The given study can be readily extended to the multi node version. Scaling to multiple nodes will certainly yield interesting results.

Matrix multiplication example lays good ground work. However, we would like to extend these experiments to more mature examples like Cholesky decomposition which have more opportunities of optimizations and comparison of that to some other interoperability framework can help give SWARM the required edge.

Related Work

There are ongoing efforts by other exa scale frameworks to introduce interoperability with the legacy MPI codes. In introduction and background sections we have briefly mentioned about these related works. The work that we found most related to ours is by the OCR team[4] where they propose MPI-Lite on OCR. Also, The XPRESS team[5] has similar goals to achieve interoperability and migration from the legacy codes. This section includes, in-depth description of major related work as well as some other related works. Also, later in this section, we highlight features that distinguish our work from these related works.

Major Related Work

In this sub-section, we will discuss the work which is most relevant and similar to our approach to interoperatibility.

OCR

In our study, we found that the interoperability work by the OCR team is closely related with that of ours. Open Community Runtime (OCR) Project[6] is a framework that explores new methods of high-core-count programming and acts as a tool that helps app developers improve the power efficiency, programmability, and reliability of their work while maintaining app performance.

The OCR team[4] has also proposed MPI-Lite on OCR in the recent presentation[7] by Mark Davis et al. As described in a presentation,the interoperability between OCR and MPI is achieved by spawning multiple MPI processes inside OCR and the joining them on their completion. The MPI_Lite library is used to communicate among these processes. Though the approach has its own limitations they claim that existing MPI code will run without modifications on their platform.

HPX

HPX[8] is a general purpose C++ runtime system for parallel and distributed applications. The team at LBNL is working on interoperatibilty between MPI and HPX. We have reached out Alice Koniges at Lawrence Berkeley National Laboratory. They have a set of simple MPI+HPX benchmarks[9] which help us understand how non-blocking MPI calls are used in this program. We were not able to find any other efforts to interoperate HPX with legacy codes.

Recently, a strong scaling of HPX and MPI AMR[10] has been reported. The authors compared an application which is using a 3-D adaptive mesh refinement (AMR) algorithm to solve the semi-linear wave equation. They have observed that In application performance experiments, the HPX runtime system substantially reduced starvation and latency effects which resulted in better load-balancing and better strong scaling than comparison code written using MPI. As levels of refinement were added to the simulation, strong scaling *improved* in the HPX version. The MPI comparison code showed the opposite behavior: strong scaling decreased as levels of refinement were added. The reduction in starvation and the mitigation of latencies when using the HPX runtime system comes at a cost of increased overhead and contention.

There are also reported effort in running HPX on an MPI Cluster[11] Under this environment, HPX comes with a Parcel port back end that is communicating via MPI. This has the advantage that HPX is now usable on any commodity cluster without special handling of hostnames or similar and the most appropriate networking implementation will then be chosen by the MPI implementation. User just has to start HPX application through mpirun/ mpiexec.

SHMEM

SHMEM[12] was developed originally by Cray for the Cray T3D and subsequently the T3E models. These systems typically consisted of a memory subsystem with a logically shared address space over physically distributed memories, a memory interconnect network, a set of processing elements, a set of input-output gateways, and a host subsystem.

There is detailed presentation[10] by Karl Feind of SGI which discusses how SHMEM can inter operate with MPI. The SGI documentation[13] explains Interoperability with the SHMEM programming model and states that one can mix SHMEM and MPI message passing in the same program. The application must be linked with both the SHMEM and MPI libraries. Start with an MPI program that calls MPI_Init and MPI_Finalize.

Also, we found work[14] where they compare the performance of the SHMEM MPI implementation with the native implementation. Micro-benchmark results show that the latency performance of the SHMEM implementation is faster for a range of small messages, while the bandwidth performance is comparable for a range of large messages.

Other Related work

In this sub-section, we will talk about some other related work which though not directly relevant but essential to put in the context of this work.

In his presentation, Tim Mattson from Intel Parallel Computing lab talks about two pathways to exa scale runtime research. There he calls MPI+X approach as being Evolutionary and CODELET based OCR approach as revolutionary. Using the example of "Proving that a shared address space program using semaphores is race free is an NP-complete problem", the presenter compares the efforts required for MPI compared to that of Multi- threading. Currently, The MPI is popular despite of its extra work upfront but easier optimization and debugging compared to that of Multi-threaded as debugging and optimization is time consuming there. This example makes it clear that pathway to Exa Scale has to provide interoperability with currently popular MPI.

The Fresh Breeze memory model[15] and system architecture is proposed as an approach to achieving significant improvements in massively parallel computation by supporting fine-grain management of memory and processing resources and utilizing a global shared name space for all processors and computation tasks. The Fresh Breeze memory model uses trees of fixed-size chunks of memory to represent all data objects, which eliminates data consistency issues and simplifies memory management. Low-cost reference-count garbage collection is used to support modular programming in type-safe programming languages.

The Delaware RunTime System - DARTS[16] is a runtime written for shared memory x86 architectures to implement the Codelet execution model. s. DARTS is written in C++, and takes advantage of object oriented programming. DARTS attempts to remain as accurate as possible to the aforementioned model with the intent of analyzing and further developing codelets.

We were unable to find work on interoperability between Fresh Breeze or DARTS with legacy MPI codes.

Acknowledgement

This material is based upon work supported by the Department of Energy [Office of Science] under Award Number DE-SC0008716.

Conclusions

In conclusion, we have demonstrated that SWARM and MPI runtimes can interoperate with the matrix multiplication example. The examples presented are simple enough to evaluate the feasibility of the approach and serve as basis to the creation of a more complex benchmarks 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.

References

[1] ETI Technical Report 002, https://xstackwiki.modelado.org/images/8/82/ETITechnicalReport02.pdf
[2] DynAx project, https://xstackwiki.modelado.org/DynAX
[3] SWARM : Swift Adaptive Runtime Machine,
http://www.etinternational.com/files/2713/2128/2002/ETI-SWARM-whitepaper-11092011.pdf

[4] Open Community Runtime, https://xstCkwiki.modelado.org/Open_Community_Runtime
[5] XPRESS, https://xstCkwiki.modelado.org/XPRESS
[6] Open Community Runtime, https://01.org/open-community-runtime
[7] Mpi-Lite on OCR, https://xstack.exascale-tech.com/wiki/images/e/e5/Mpilite_graph_v7.pdf
[8] Hartmut Kaiser, Thomas Heller, Bryce Adelstein-Lelbach, Adrian Serio, Dietmar Fey, HPX – A Task Based Programming Model in a Global Address Space, PGAS 2014: The 8th International Conference on Partitioned Global Address Space Programming Models (2014).
[9] MPI+HPX Benchmarks, https://www.nersc.gov/users/computational-systems/testbeds/babbage/hpx-on-babbage-and-edison/
[10] Strong scaling of HPC and MPI-AMR, http://stellar.cct.lsu.edu/2011/10/strong-scaling-of-hpx-and-mpi-amr/
[11] Running HPX on MPI cluster, https://github.com/STEllAR-GROUP/hpx/wiki/Running-HPX-on-an-MPI-Cluster
[12] Cray Research, Inc.: SHMEM Technical Note for C, SG-2516 2.3. (1994)
[13] MPI and SHEMEM, ftp://ftp.u-aizu.ac.jp/pub/SciEng/numanal/phase/o2k/technical_doc_library/docs/cug/mpi.pdf
[13]Interoperability with the SHMEM programming model, http://techpubs.sgi.com/library/tpl/cgi-bin/getdoc.cgi?coll=linux&db=bks&srch=&fname=/SGI_EndUser/MPT_UG/sgi_html/ch03.html
[14] A New MPI Implementation for Cray SHMEM, http://www.sandia.gov/~rbbrigh/papers/shmem-mpi
[15] J. Dennis. 1997. A Parallel Program Execution Model Supporting Modular Software Construction. In *Proceedingsof the Conference on Massively Parallel Programming Models* (MPPM'97). IEEE Computer Society, Washington, DC, USA, 50
[16] DARTS: A RUNTIME BASED ON THE CODELET EXECUTION MODEL, http://www.capsl.udel.edu/pub/doc/theses/master/Suetterlein-master.pdf

Reservoir:

This quarter, Reservoir has worked on the parallelization of block-structured codes on clusters, and on supporting UIUC's PIL as a front-end to R-Stream, as a way of constructing a hierarchical mapping. The following sections detail these contributions.

Parallelization of block-structured codes

As presented in the previous reports, R-Stream's parallelization to clusters relies on a nimble Partitioned Global Address Space (PGAS) approach, in which all communications that may cross node boundaries are performed through smart, logical DMAs. Computations performed by a node follow the GET-COMPUTE-PUT model[5].

In the case of dense tiled array, this model allowed us to use Global Arrays[6] (GAs) straightforwardly as a logical DMA (Direct Memory access) layer. GAs support automatic data distribution and tiling across nodes of a cluster. However, we have established that even the thin layer of abstraction provided by GAs would not permit an efficient implementation of a

[5] This model is also assumed by the PEDAL framework developed at PNNL.
[6] Jarek Nieplocha, Bruce Palmer, Vinod Tipparaju, Manojkumar Krishnan, Harold Trease and Edo Apra. 2006. "Advances, Applications and Performance of the Global Arrays Shared Memory Programming Toolkit." International Journal of High Performance Computing Applications, Vol. 20, No. 2, 203-231p.

logical DMA layer for distributed block-sparse arrays. Among the causes for this are the following limitations:

- array allocation is collective (and we want to dynamically allocate data tiles as they become non-empty),
- static data distribution introduces undesirable coupling between the number of non-zero data tiles and their distribution,
- data-dependence of the address of a data element makes that several implicit communications would occur even for one single DMA transfer
- the potential need for concurrent data tile creation involved collective synchronizations.

As a consequence, we decided to rely on a lower-level layer, ARMCI[7], which provides Remote Direct Memory access (RDMA), a fast mechanism for one-sided, asynchronous copy-less inter-node communication. It also enables local dynamic allocation of memory that can be targeted by RDMAs. Our basic design for distributed block sparse arrays does not imply any extra synchronizations, and it enables avoiding several types of communications.

Let us review its basic design and then go through the optimizations we are considering.

Basic design

There are two main components to our automatic parallelization scheme to distributed block-sparse computations:

- The R-Stream parallelization tool, which takes a sequential C code written as if the arrays were dense. It produces parallel code which accesses a distributed block-sparse version of the arrays through a logical array access API.
- The R-Stream block-sparse runtime, which implements the array access API. It is made of array creation methods and array access methods, under the form of an extended logical DMA interface.

Here we distinguish two sources of sparsity:

- Data sparsity, which relies on the existence of a highly dominant default value to compress the representation of data. In block-sparse arrays, the compression is obtained by not representing data tiles that do not contain any data. It is possible to further compress the data, which we do not consider here.
- Computational sparsity, which relies on the existence of a condition on the input data under which the computations to be performed are significantly cheaper. An optimal case of computational sparsity is when the computation becomes the identity for some input values, i.e., no data is changed. In this case, the computation can be skipped altogether.

Our goal is to take advantage of both sources of sparsity. Data sparsity is implemented by storing the distributed arrays as block-sparse. Computational sparsity is implemented by introspecting input data to tasks (obtained through a GET) and controlling the execution of the

[7] ARMCI was developed at PNNL as well as GAs. PNNL's implementation of GAs is based on ARMCI. http://hpc.pnl.gov/armci/

task based on the input values. For the purpose of demonstrating the approach, in the experiments performed in this project we are limiting this to the case when the computation is the identity whenever one of its inputs is zero. This particular case is frequent enough in sparse codes to be useful.

Data sparsity

In the R-Stream parallelizing tool, data sparsity is specified in the code through a pragma parameter. Since the computation to be parallelized is expressed in terms of dense arrays, but its parallelized version uses distributed block-sparse arrays, there is a formal mismatch between the actual distributed array and its version as specified in the sequential, dense version.
The R-Stream block-sparse runtime layer handles distributed block-sparse arrays through an identifier (similarly as files can be handled through an integer file descriptor in C). The user is required to bridge the representation gap by specifying the distributed block-sparse identifier that will correspond to each array to be distributed and sparsified.

The syntax is illustrated with a matrix multiplication code in Figure 1, where array A is defined as block-sparse with a data tile size of 32x32 and 4523 as its runtime array identifier. In the parallelized code, Initialization, reads and writes to the array corresponding to A will be done through the R-stream block-sparse array API, using identifier ‘4523’ as parameter. In the example in Figure 1, the value of N would most likely be very large.

```
#pragma rstream map block_sparse:A=32x32=4523
void matmult(real_t A[N][N], real_t B[N][N], real_t C[N][N]) {
  int i, j, k;
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      C[i][j] = 0.0;
      for (k = 0; k < N; k++) {
       C[i][j] += A[i][k] * B[k][j];
      }
    }
  }
}
```

Figure 1. Specifying sparse data-block structure in R-Stream input programs.

The R-Stream runtime layer API represents the data as a distributed set of dense data tiles. Data tiles are represented at the API level by their coordinate in the data space. Internally, a data rank function is generated by R-Stream, which injectively maps multi-dimensional data tile coordinates to an integer.

An **ownership** function defines which node is responsible for servicing data requests for each data tile. Default ownership functions are generated by R-Stream, and can be easily overridden by the user. The ownership function defines which node the requester of data needs to talk to in order to access the requested data. While it is often the case, ownership is *not equivalent* to distribution. The owner of a data tile knows where the data tile is, and makes sure that a request for data it owns is serviced. All nodes use the same ownership function. This way, no inter-node synchronization is necessary to know which node to talk to when performing communications. Also, this guarantees a maximum of two hops between a data requester and the node in which the data resides (which can be the owner or a third-party node known to the owner). Ownership can be updated lazily and without collective communications to optimize the number of hops.

The **distribution** function is thus generally dynamic and defined in a distributed way by the owners. This enables data load balancing in cases when the ownership function results in high data imbalance. However, in the scope of this project, we are only considering a static distribution function which matches ownership.

The **communication** API is a layer on top of a 2-D RDMA API. However, since the set of data tiles is sparse, and hence dynamic, the requester of data does not generally know:

- whether the data tile contains any non-default data, or
- what the base address of the data is (if its tile contains non-default values)

Hence, the general unoptimized communication pattern, expressed through an asynchronous get/put/wait DMA API is as follows.

A **dma_put**(src@, src_stride, dst@, dst_stride, nb_packets, packet_size, dst_tile, tag):

1. Checks whether there is any non-default values to send. If there are, increment (and create if non-existent) the dependence counter associated with *tag.* If there aren't, goes to END.
2. Uses the ownership o = own(dst_tile) function to determine the owner.
3. Remote-calls a service_dma_put() codelet on the owner with the DMA parameters as parameters.
4. service_dma_put():
    a. creates the tile if absent
    b. Computes the local, destination base address and stride *within tile* dst_tile
    c. Launches a RDMA_GET with the right parameters, using OFED. With HW that supports RDMA (such as Infiniband and RoCE), an optimal, copy-free transfer is taken care of by the network interface.
    d. Waits for completion of the RDMA_GET (with RDMA_wait) and remote-calls service_complete(tag) on the requester.
5. service_complete(tag) decrements the counter associated with *tag.* When the counter reaches zero, dma_wait(tag) can continue. SWARM has a TagTable which will ease this implementation.
6. END.

Similarly, a **dma_get**(src@, src_stride, dst@, dst_stride, nb_packets, packet_size, src_tile, tag):

1. (nothing)
2. Uses the ownership o = own(dst_tile) function to determine the owner.
3. Remote-calls a service_dma_get() codelet on the owner with the DMA parameters as parameters.
4. service_dma_get():
    a. If the tile is absent (i.e., it is full with the default value), remote-calls a service_zero_dma_put() codelet on the requester, and goes to 5.
    b. Computes the local, destination base address and stride *within tile* src_tile
    c. Launches a RDMA_PUT with the right parameters, using OFED.
    d. Waits for completion of the RDMA_PUT (with RDMA_wait) and remote-calls service_complete(tag) on the requester.
5. service_complete(tag) decrements the counter associated with *tag.* When the counter reaches zero, dma_wait(tag) can continue. SWARM has a TagTable which will ease this implementation.
6. END.

dma_wait counts the number of gets and puts associated with their tag. Remote RDMA_waits result in the decrement of the counter. The execution of the DMA_wait is triggered when the counter reaches zero.

All remote codelet spawns are performed through the SWARM network layer. Codelets are spawned into an existing SWARM runtime (basically, into a single process that runs several workers).

This design presents the following desirable characteristics:

- **Efficient**. Leverages hardware-supported asynchronous, zero-copy RDMA when available.
- **Asynchronous**. Presents an asynchronous API (with DMA_waits).
- **Codelet-based**. Meshes with a codelet/EDT-based runtime running on each node: the SWARM runtime.
- **Generic**. Virtualizes the sparsity: API users don't know whether the array is sparse or not. The base API has the same semantics as for a dense (tiled) array.
- **Adaptive**. The number of workers performing communications is *adapted to the load*. This is an important distinguishing factor from other codelet-based runtimes, in which the number of communication workers is fixed, making them either a bottleneck or a wasted resource most of the time.
- **Precise**. Communicates the exact set of data required by the user (as compared to, for instance, communication by copying entire tiles)

Several optimizations can be applied to it, discussed in the “Optimizations” section below. But first, we show how computational sparsity is achieved using the R-Stream parallelization tool in the next section.

Computational Sparsity

Computational sparsity is implemented as an optional R-Stream optimization and controlled through a command-line option.

The basic idea is that if the input data to a task satisfies the sparsity condition (which we are restricting here to “either input is zero”), the computation and update of the output data are skipped. We are relying on the get-compute-put structure of the computations as generated by R-Stream, and on the use of an asynchronous API, in which the completion of a groups of transfers is waited upon by a call to `dma_wait()`. In our computational sparsity optimization scheme, such wait operation is overloaded to also inform about the existence of non-zeros among the inputs it is waiting for. `dma_get` operations inspect incoming data and detect the presence of non-zeros. `dma_wait`, which returns after the relevant set of dma_gets are executed, checks this status and returns a boolean representing whether the sparsity condition has been met.

The boolean value controls whether the computation and the subsequent `dma_put`s are executed. The general structure of the code is presented in Figure 2.

```
rds_array_create(4523, 2, 32, 32, 1000000, 1000000);
```

```
for (i=...; ... ; ++i) {
  for j (j = …; …; ++j) {
    rds_dma_get(/*identifier=*/4523, src_addr1(i,j), src_stride1,
                dst_addr1(i,j), dst_stride1, size1, nb1,
                /*tag=*/1, i, j);
   }
}
int hasNonZeros = rds_dma_wait(/*tag=*/1);
if (hasNonZeros) {

  /* Computation code here */

  rds_dma_put(/*identifier=*/4523, src_addr2, src_stride2,
              dst_addr2, dst_stride2, size2, nb2, /*tag=*/1, i, j);
}
```

Figure 2. Avoiding identity computations through dma_wait overloading.

Now that we know how computational and data sparsity are achieved through a combination of code optimization and the use of a smart logical DMA layer, we can discuss optimizations.

Optimizations

We are presenting two types of optimizations in this block-sparse parallelization scheme: optimizations of the runtime (communication) layer, which aim at avoiding communications, and optimization offered by the parallelization tool, which aim at simplifying the data transfers.

Avoiding default-value transfers. Only data tiles containing non-default-values are encoded in our representation. Even in the basic design, when the transfer of data from a non-encoded (because empty) data tile is requested, the owner of the tile returns a short message which encodes that the full set of requested data is empty, as opposed to a set of zeros.
In addition to this, we are considering the following optimizations:

- **Inspection**. When performing a `dma_put`, data sets made of default (and optionally close-to-default) values are not sent, resulting in no communication. This requires a runtime inspection of data. This optimization is expected to fare well in the cases when positive cases are frequent.
- **Non-empty tile caching**. The address of a non-empty data tile is cached by the requester, which can then issue the RDMA directly instead of requesting an RDMA from the owner. In the even when the tile would become empty, the owner can broadcast an update to its potential cachers, and keep the tile around (filled with default values) until all cachers have updated their cache.

- **Empty-tile caching**. This is expected to save a large number of communications, but introduces a consistency problem, as data requesters need to know precisely whether a data tile is empty or not. Luckily, we have access to dependences among codelets (they are generated by R-Stream), as well as ownership functions. A empty-tile cache entry only needs to be updated if any dependence predecessor of the current codelet has created a new tile that is cached as empty, and if the tile will be read by the codelet. Since empty tile caching can be maintained at the node level, cache updates only need to be performed when codelet dependences cross owner boundaries. However, it is likely that naively maintaining and transferring a growing list of recently-created tiles whenever owner boundaries are crossed will increase communication volume and bookkeeping overhead. Hence we are considering the following relaxations, including the following:
    - Assuming tile creations are rare, we can enforce **broadcast cache updates** whenever a tile has been created. The broadcast update is enforced before the codelet that created the tile completes. This creates additional (but rare) one-to-many communications.
    - The granularity at which tile caching is performed can be coarsened, by **mapping each cache entry to a set of data tiles**. This is optimal in the case when large sets of tiles stay empty for long periods of time.
    - In the case of a traditional distributed mapping based on barriers, caches can be conservatively updated each time a barrier is reached.

Simplifying data transfers. We are considering two ways in which R-Stream optimization could simplify the job of the runtime.

- **Partitioning transfers by data tile** doesn't increase the number of RDMA communications, since data is tiled and the stride between consecutively transferred elements of a tile is different from the stride between consecutive elements that belong to different tiles. When using 2-D DMAs (as we do), a transfer that spans more than one tile has hence to be broken up into several smaller intra-tile transfers anyway by the runtime. As a result, if each of the dma transfers generated by R-Stream transfer data accesses data from the same tile, the job of breaking down the transfers by tile is saved.
- Assuming transfers are partitioned along data tiles, **DMA strides and base addresses** within a tile can be **computed directly by R-Stream**, rather than being translated by the runtime.

Implementation status

The specification of block-sparse arrays, the partitioning data transfers into data tiles and the generation of sparse computation through dma_wait conditionals are implemented but require a small amount of integration work. The sparse data block runtime is under implementation.

Supporting PIL as an R-Stream frontend

UIUC and Reservoir have been working on a two-level parallelization scheme. The goal of this work was to complement the R-Stream capability of inter-node parallelization to SWARM with a coarse-grain parallelization using PIL. The resulting parallelization is hence hierarchical: PIL creates N threads, which in turn get parallelized by R-Stream to N*M workers. Ideally, N*M would match the number of hardware threads T on the machine (over-provisioning in SWARM is not obtained by having more workers than threads, but more codelets than workers).

The R-Stream runtime was using SWARM's "callInto" mechanism as a bridge between sequential code and code parallelized with SWARM. Basically, the sequential thread "calls" a parallelized function by entering the SWARM runtime, and exits the runtime (without shutting it) when all the parallel codelets have completed. An issue with this mechanism is that the sequential thread is blockingly waiting on the SWARM workers to be done.
Hence, when combining PIL and R-Stream, the N threads created by PIL, which each use M workers, are wasted as they blockingly wait. An extreme case[8] is when PIL creates as many threads as the number of hardware threads on the machine. In this case, we have N=T and there are no threads left for the tasks parallelized by R-Stream, which results in a deadlock. But generally, having N unused workers out of N*M is wasteful, both in terms of energy and performance.

This problem required a fix on both the PIL and the R-Stream sides. On the R-Stream side, this issue was fixed by replacing the "callInto" mechanism with an asynchronous call, which doesn't require the blocking of a thread. However, it required the caller (the PIL-generated code) to perform an asynchronous call and associate the completion of the R-Stream parallel code with a continuation code.

As reported in the corresponding section by UIUC, at the time of writing, there is a remaining bug in the modified R-Stream runtime for SWARM, which prevents some examples from working. The problem is under investigation.

Reservoir Labs publications

Sanket Tavarageri, Benoit Meister, Muthu Baskaran, Benoit Pradelle, Tom Henretty, Athanasios Konstantinidis, Ann Johnson, Richard Lethin. "**Automatic Cluster Parallelization and Minimizing Communication via Selective Data Replication**", in proceedings of the 2015 IEEE High Performance Extreme Computing conference (HPEC'15).

[8] But this case is irrelevant in the context of hierarchical parallelization since the coarse-level parallelization would not leave any resources to finer-grain parallelization.

UIUC:

In this quarter the UIUC team focused the work on the performance evaluation of PIL/HTA and RStream integration with PIL. We present the details of the former topic here.

Performance Evaluation of the PIL implementation and API

In the first design of mapping SPMD program execution on top of the codelet runtime, the program starts with P codelets representing processes 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 process codelets do not share data directly. Whenever one requires data owned by another, they need to communicate with point-to-point messages. The design is illustrated in the following figure. The SPMD mode potentially has better performance (Figure 6(b)) when there are minimal dependences among codelets distributed to different processes.

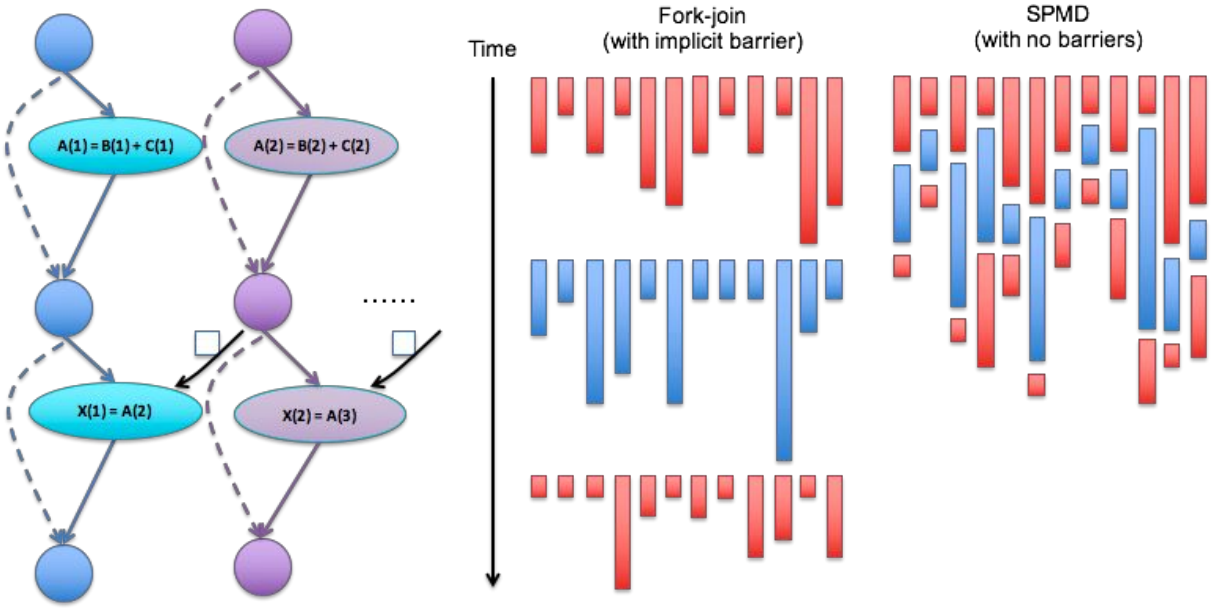

Figure 6: (a) SPMD program execution with codelets. (b) Executing three operations with load imbalance.

However, the data dependences among codelets distributed to different processes exist in almost any practical parallel application. To understand what are the effects of data dependences, we implemented a Cholesky factorization to benchmark the SPMD execution mode. In our preliminary experiments, we discovered that the fork-join execution performs better than SPMD execution, since the codelets created are usually more than the number of worker threads available (Figure 7(a)), and the codelet runtime can dynamically balance the workload among workers by work stealing. On the other hand, In SPMD mode, each process codelet deals with the work associated with their owned tiles one-by-one, creating a new codelet only when the last one created finishes (Figure 7(b)). Some process codelet could have less workload than others and run out of work. It has to idle and wait for other process codelets due to data dependence.

To resolve the problem, we implemented a new design that allows nested parallelism within processes. By having an extra level of parallelism in SPMD execution, the amount of exploitable parallelism is no longer limited to a fixed number P, because a process codelet can create all finer-grain codelets at once and thus making it possible to have much more codelets than worker threads. As long as the codelets are in the same memory address space, the runtime can dynamically schedule them to available worker threads, as illustrated in Figure 7(c).

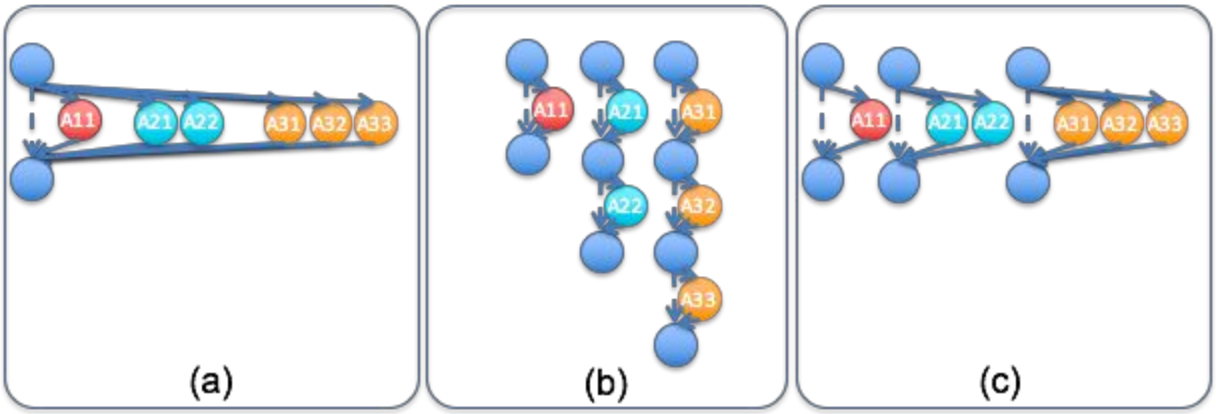

Figure 7: (a) Fork-join execution. (b) The initial SPMD design cannot benefit from the runtime's dynamic scheduling because existing codelets are always smaller than P. (c) SPMD with nested parallelism creates most finer-grain codelets at once so that there are more codelets to be dynamically scheduled

We implemented the nested parallelism support on the OpenMP backend (using OpenMP tasking) and the SWARM backend. The experiments were conducted with HTA Cholesky decomposition application running on a single node machine with 4 Intel Xeon E7 4860 processors (40 cores and 80 threads) to demonstrate the benefits of this scenario.

Figure 8 shows the OpenMP results of running Cholesky factorization in SPMD mode with nested parallelism support. As we expected, although tiles were statically distributed to process codelets, the fine-grain level codelets could be shared among all available worker threads. Thus, the overall performance is better than the original design when there is only a single level of parallelism. It also shows that in some configurations the performance scales better than the fork-join execution.

In the case of SWARM backend (Figure 9), we observed that the nested parallelism helps the performance in SPMD now that the load can be dynamically balanced. However, the way asynchronous tasks are generated in SWARM has too much overhead so the overall performance still cannot match with the fork-join results.

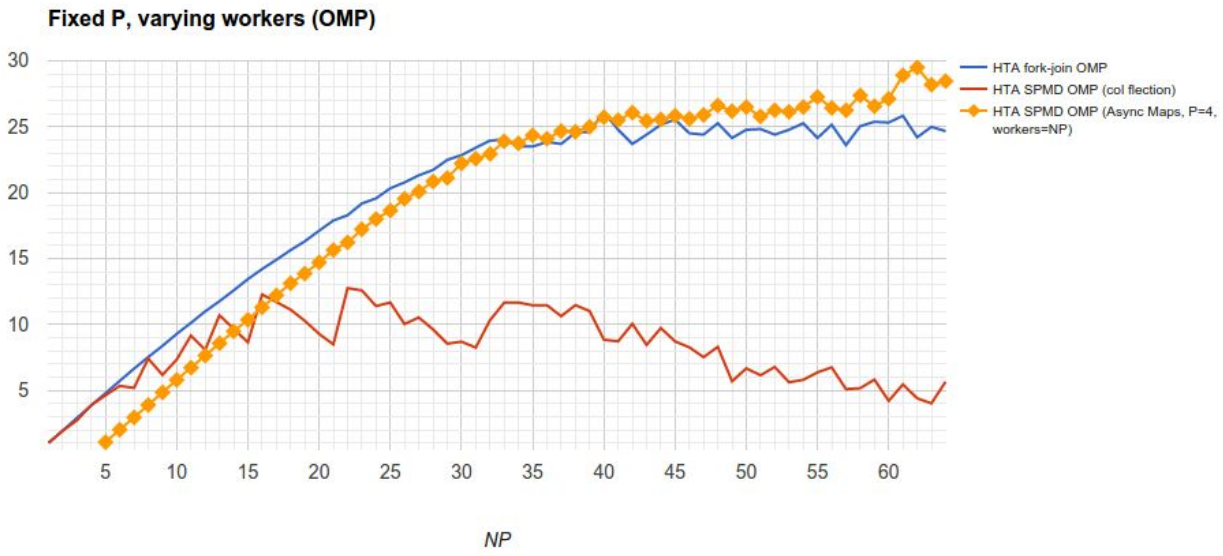

Figure 8: Nested Parallelism in SPMD mode with OpenMP backend

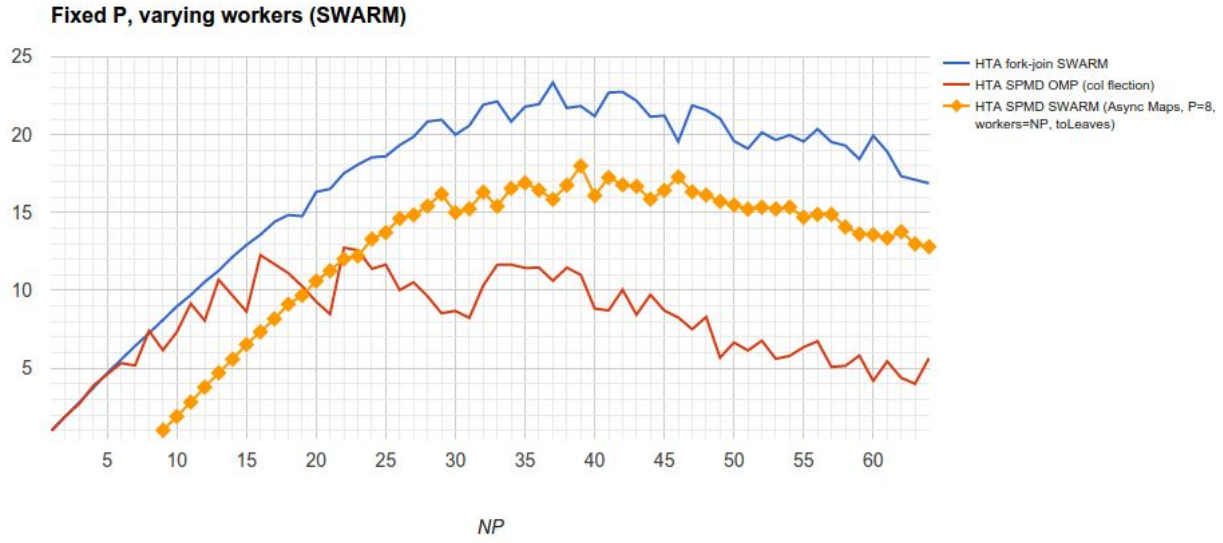

Figure 9: Nested Parallelism in SPMD mode with SWARM backend

There are some interesting future research topics left to explore. For example, given the number of processors on a shared memory machine, the optimal number P for the first level parallelism is application dependent. Also, by assigning priority to the EDTs according to the position of the tasks in the dependence graph, it is also possible to minimize the wait due to data dependences and results in better performance.

PNNL:

In this quarter the PNNL team focused on improving data locality further with low overhead data restructuring technique. Such technique is designed to convert strided access with high order of reuse to contiguous accesses.

Gregarious Data Restructuring:

Given two representative examples (Matrix multiply and a LU decomposition), we present our methodology and the operations needed to restructure the data structures based on their access patterns.

We use the matrix representation of the iteration space and access matrix to find the amount of reuse that each data structure has. We identify the depth of a loop nest, which is the iteration space dimensionality, and the rank of the access matrix given by the access dimensionality. If the depth>rank, it means that there exists a high order of reuse for that memory reference. For example, in our matrix multiplication example with size nxn matrices, accesses to matrices have a depth of 3 (i, j, k loop nest) and a rank of access matrix is 2. Hence the amount of reuse is $n^2(n-1)$ for each memory references. Using the iteration space vectors (i.e. access patterns) for each data structure, we can obtain reuse vectors that can be used by single (self-reuse) or multiple (group-reuse) parallel threads. Our goal is to find these reuse vectors and to reduce (by transforming) any inherent data access features (such as strides) that can reduce the locality inside the shared caches.

In our methodology, we create a storage space in which the transformed data structures can reside. A naïve methodology can create wasteful large spaces, but we are bounded by number and size of tiles in the shared dimension. Thus, our restructure space is given by multiplying the number of tiles in the shared dimension times the size of each.

To restructure the data, we take into consideration the specific layout of the structure plus its access pattern. The access pattern might work against locality because of its inherent characteristics. We use the information obtained from access matrices and layouts to perform transformations such that:

· All elements of outer tiles (i.e the inner tiles in a tiled hierarchy) stay together in memory. The mapping allows memory accesses to take advantage of open memory pages and also results in high residence at various cache levels.

· Elements of the innermost tiles are accessed in a contiguous fashion. This mapping allows better cache locality and reduces unnecessary conflicts.

To achieve this, we map the same restructuring space to disjoint (both in space and time) data structures so that parallel actors can take advantage of this data. Since the mapping strategy

involves the reshaping of the data within and across the tiles, such transformations are done on both the original and tiled domains. This can result in very expensive index remapping operations. We bypassed these costs, by calculating the offset of the indexes and constraining the access patterns to the following cases:

Condition 1: If the shared space (set of outer tiles) is arranged by rows and inner elements are accessed by column, then do the transformation with the calculated displacement and perform a copy transpose at the element level as shown in Figure 1. Initially, K' and J' are calculated by subtracting lower bounds K and 0 respectively. The respective lower bounds become the offset (since they are already relative to the original index K and J) and are translated to the values corresponding to the original domain by multiplying with the tile size. It is then followed by a transpose at element granularity to change access towards rows.

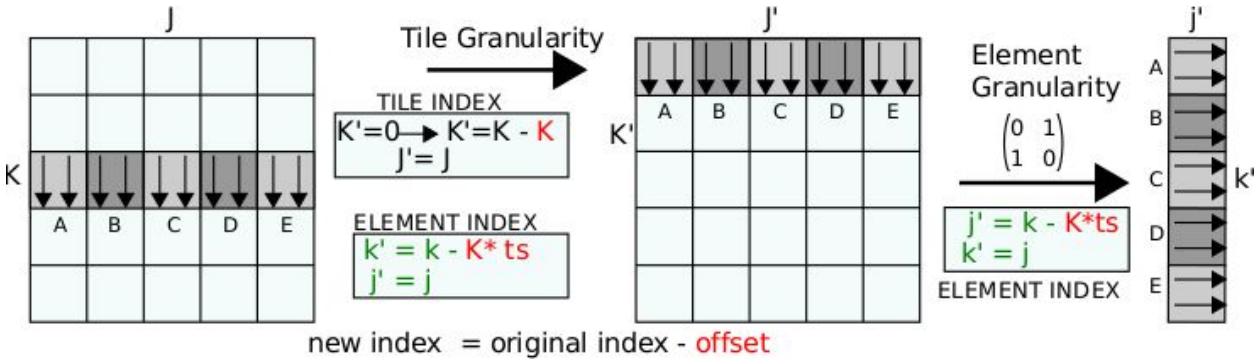

Figure 1: Transformation Condition 1: Strided access and row arrangement of parallel tiles (A-E) transformed to have contiguous access using calculated offset (shown in red)

Condition 2: If both outer tiles and inner elements are accessed by column, then do the transformation with the calculated offset. This is followed by transpose at element level as shown in Figure 2. Here first K' and J' are calculated by subtracting the lower bounds and transposing the tile index. The resultant value of K' and J' i.e. 0 and K, are then used to calculate the offset relative to the original index. In this case, relative to the original index K, K' becomes K-K and relative to J, J' become J+K-J. This gives the offset required to calculate the new indices. It is then followed by a transpose at element granularity to change access towards row.

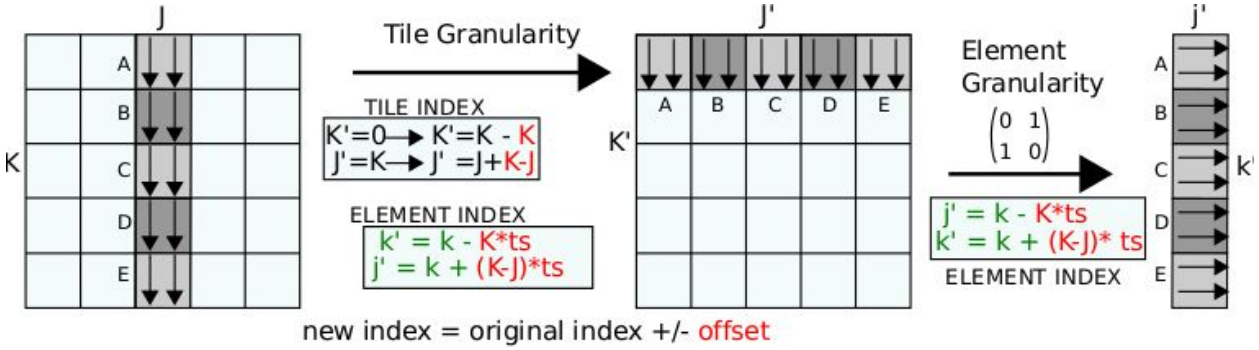

Figure 2: Transformation Condition 2: Strided access and column arrangement of parallel tiles (A-E) transformed to have contiguous access using calculated offset (shown in red)

In both cases, the amount of displacement required relative to the original index is calculated to perform targeted transformations. This approach allows calculation of the new indices in a single step without the use of expensive operations.

Using this methodology, parallel thread units grab different data sets according to their group identification and perform restructuring in a concurrent fashion.

We use the Tile-Gx36 architecture as our testbed for the restructuring methodology. It has 36 processor cores, each equipped with 32KB local 2-way L1 cache and 256KB 8-way L2 cache. All caches are inclusive. Each core also has access to the other core's L2 cache in the grid, giving an impression of a virtual L3 cache. accesses to L3 caches are much cheaper than accessing the memory (anywhere from 2x to 3x faster). Our framework uses intratile parallelism to exploit reuse within the grid and performs restructuring for better access strides.

On the software side, we select a vanilla matrix multiplication and a LU decomposition, to display the effectiveness of our technique.

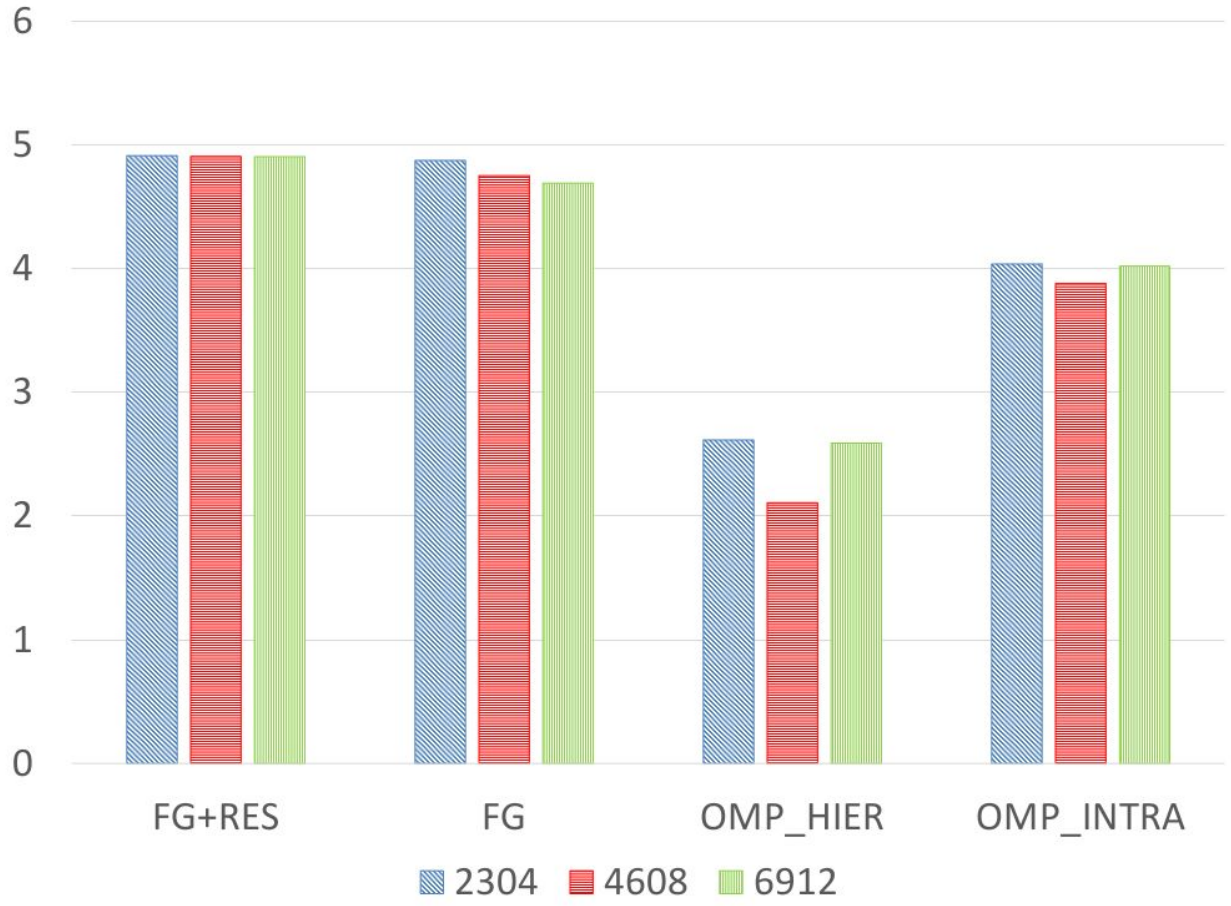

Figure 3: Matrix Multiply performance in GFLOPS. FG+RES is the fine grain tiling plus the restructuring techniques, FG refers to only fine grain tiling, OMP_HIER is the hierarchical version of an optimized OpenMP code and OMP_INTRA is the OpenMP version with inner TP

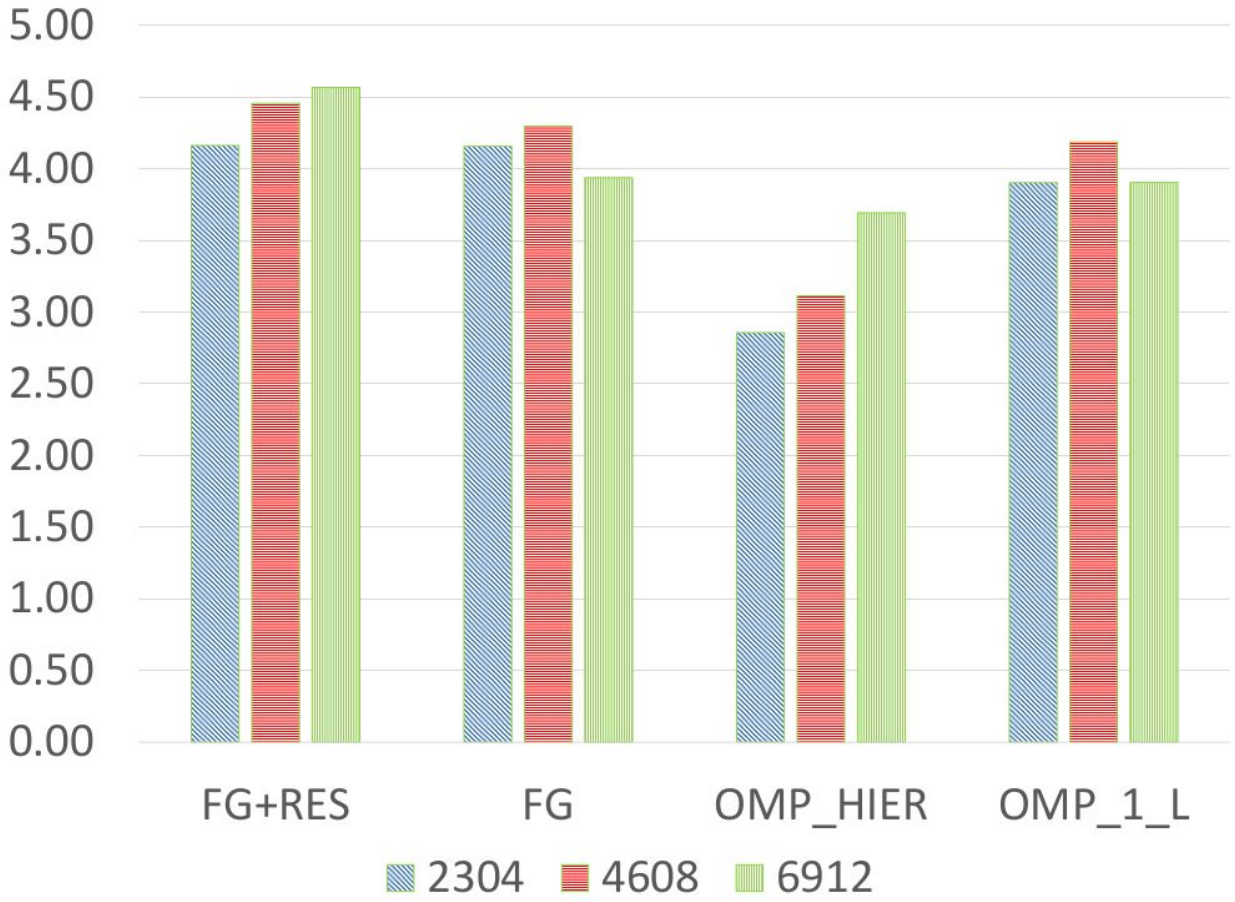

Figure 4: LU performance in GFLOPS. FG+RES is the fine grain tiling plus the restructuring techniques, FG refers to only fine grain tiling, OMP_HIER is the hierarchical version of an optimized OpenMP code and OMP_1_L is the OpenMP version with one level hierarchy

Figure 3 shows performance for matrix multiplication over different problem sizes. Our result shows an improvement of up to 26.50% over best case OMP code (OMP_INTRA as in OMP intratile). Additionally, it shows up to 4.5% improvement when compared against our own fine-grain grouping techniques.

Figure 4 shows performance for our second example, LU decomposition, at different problem sizes for fine-grain with/without restructuring and OMP code. For most cases, we use hierarchical and one level tiled OMP code instead of intra-tile OMP parallel code as reference as its performance is better. Our technique with restructuring has up to 31.4% advantage over OMP one level tiling and 15.9% advantage over fine-grain without restructuring.

Publication:

- Sunil Shrestha, Joseph Manzano, Andres Marquez, Stephane Zuckerman, Shuaiwen Leon Song and Guang Gao, "*Gregarious Data Re-structuring in a Many Core Architecture.*" In the 17th IEEE international conference on high performance computing and communications (HPCC 2015). August 24-25, 2016, New York, USA.

Appendix A

ET International Inc.
10 Montainview Drive
Newark, Delaware
302-738-1438 phone

August 14, 2015
via: email
Sonia Sachs
Program Manager
sonia.sachs@science.doe.gov

RE: NCE Request for DynAx DE-008716
Dear Dr. Sachs:

ETI would like to request a NCE on behalf of the X-Stack project called DynAx.

- Reservoir is making the request for an additional 2 months which will allow Reservoir team to continue to work on their part of the Dynax project and spending remaining fund for this purpose. You should have already received Reservoir Inc's request letter by email which contains all the details – which I also attached here for convenience.
- UIUC is making the request for an additional 4 months which will allow UIUC team to continue to work on their part of the Dynax project and spending remaining fund for this purpose. I am attaching UIUC's request letter by email which contains all the details.
- PNNL notified you of their unexpended funds to be carried over into FY16 via their yearly progress report submitted to you June 11, 2015. With work expected to be completed in early FY16, this will allow the PNNL team to continue to work on their part of the Dynax project and spend their remaining funds.

ETI is expected to complete ETI's portion of the Dynax project and burn out the ETI portion of the funds by August 31, 2015. ETI will request additional 4 months to complete the documentation.

Sincerely,

Guang R. Gao
Dynax PI
ET. International Inc.

Appendix B

Reservoir Labs
632 Broadway Suite 803
New York, NY 10012
212 780 0527 phone
212 780 0542 fax

August 10, 2015

via: email

Guang Gao
Principal Investigator
ggao.capls@gmail.com

RE: NCE Request for DynAx DE-SC0008716

Dear Pr. Gao:

Reservoir Labs would like to formally request a NCE to our portion of the DynAx project, part of the X-Stack program, award number DE-SC0008716

The balance remaining for Reservoir's portion as of July 31, 2015 is $65,120.46.

The proposed new date for completing the project is October 31, 2015.

We are making the request due to staffing constraints and the additional 2 months will allow us to enhance the quality of the deliverable.

With the additional time we will bring the R-Stream prototype runtime, backend and mapper for x86 clusters based on SWARM to a level of maturity that will enable automatic parallelization to codelets on clusters. We will use R-Stream to automatically parallelize the core computation of the ExMatEx CoSP2 proxy app to clusters. CoSP2 represents a sparse linear algebra parallel algorithm for calculating the density matrix in electronic structure theory.

The targeted features of the runtime include a multidimensional block-sparse distributed array API based on explicit asynchronous Remote Direct Memory Access (RDMAs) and adaptive communication/computation worker ratio.

The approach developed here differs from other codelet-based approaches in at least two useful ways:

- Only the exact data set required by computations is transferred across nodes (as opposed to coarser-grain blocks).
- The number of communication workers is a function of the number of communications to be performed at any given time. We conjecture that solutions proposing a fixed number of communication workers makes these workers either a bottleneck or a wasted resource during most of the program execution time.

Appendix C

UNIVERSITY OF ILLINOIS
AT URBANA-CHAMPAIGN

Department of Computer Science
201 North Goodwin Avenue
Urbana, IL 61801-2302 USA

August 13, 2015

Sonia Sachs
Program Manager
Office of Science
Department of Energy
Washington DC

Dear Dr. Sachs:

I am writing to request a no cost extension, until December 31, 2015, to the subcontract that the University of Illinois has with ET International, Inc. (ETI). This subcontract currently ends on August 31, 2015 and its goal is to conduct work for the X-Stack project entitled DynAx, which is sponsored by your office. We estimate that by August 31, we will have a balance of $51,016.

We are requesting this extension to be able to complete our work of implementing a high level notation, Hierarchically Tiled Array, using the SWARM system developed by ETi. While we now have an implementation that produces correct code, the performance at this point is not good enough to draw conclusions about the notation and the SWARM system. Our goal during the four months after August 31 would be to tune both the translator and help tune SWARM to demonstrate good scalability across a range of codes using a notation that enhance programmer productivity.

Sincerely,

David Padua

David Padua
Donald Biggar Willett Professor

telephone 217-333-3426 • *fax* 217-333-3501 • *email* admin@cs.uiuc.edu