Preliminary scaling results on multiple hybrid nodes of Knights Corner and Sandy Bridge processors

Tan Nguyen and Scott B. Baden
Department of Computer Science and Engineering
University of California, San Diego, La Jolla, CA 92093 USA
Email: {nnguyent, baden}@ucsd.edu

***Abstract*—We discuss our experience in optimizing a stencil method on an Intel Xeon Phi-based cluster. We describe our solutions to three challenges: tolerating the high cost of inter-node communication, mapping program parallelism to multi-core and many-core processors, and balancing workloads on-node across heterogeneous resources. We present results on TACC's Stampede system. While details of how to optimize stencil methods will differ from that of other applications motifs, the 3 issues we have brought up will apply as well.**

***Keywords*—MIC, Intel Xeon Phi, data-driven, virtualization, latency hiding, load balancing, heterogeneity.**

I. Introduction

At present, it appears that further improvements to High Performance Computing (HPC) systems will mainly come from enhancements at the node level [1]. Node architectures are changing rapidly, and a heterogeneous design that uses a device (i.e. coprocessor or accelerator) to amplify node performance is gaining traction. However, heterogeneous nodes complicate application design which challenge the application programmer. First, performance amplification significantly raises communication costs relative to computation. Thus, we are required to tolerate communication delays [2], [3], avoid them [4], or both. Second, processor mapping is challenging due to the heterogeneous design. This task also requires significant application redesign to (1) work within the limitations of the interface to the memory subsystems and (2) to use the interface in a way that utilizes the resources efficiently. Third, the performance differential between the processors and devices introduces the need to solve a load balancing problem within the node. This is true even if the application has no inherent load balancing problem.

In this paper we discuss our experiences in using a directive-based translator to overcome such challenges on Stampede, a system employing Intel Xeon Phi coprocessors and Sandy Bridge processors. We apply our techniques to a 7-point stencil solver, an important application motif [5]. The starting point for our study is a conventional MPI+OpenMP code. We transform this code using our custom source-to-source translator called Bamboo [2]. Bamboo interprets the MPI API and transforms the MPI input code into an equivalent program represented by a task precedence graph and running under a data-driven execution model. The task graph virtualizes the MPI processes, enabling us to control granularity as needed.

Among the four different MIC execution models currently supported on Stampede [1], we found that symmetric mode (utilizing both MIC and Sandy Bridge) always yielded the highest performance. However, this is the most difficult mode to use due to the three obstacles outlined above. Thus, in this paper we only show our experiences in optimizing for symmetric mode, though some of the optimizations apply to other modes as well. Experimental results on up to 32 nodes[2] show that our techniques improve performance by up to 32%. This improvement comes as the result of tolerating significant communication delays. To facilitate the task of mapping program parallelism to heterogeneos computing resources, we propose a *rectification* strategy for symmetric mode. We demonstrate experimental results showing that the proposed model can efficiently employ both types of processing resources, traditional Sandy Bridge processor and MIC. For our stencil method, it turns out that the two Sandy Bridge processors deliver nearly identical performance to the MIC coprocessor. However, this situation may or may not apply to other hardware configurations or applications. Thus, we built a variant of the 7-point stencil application that enables the user to adjust workload imbalance, enabling us to emulate a node where the device runs the application at a different speed than the host. Our data-driven formulation of the modified application was able to mitigate the load imbalance we introduced and also tolerate communication delays, thereby improving performance on 64 nodes[3], varying from 10% to 40% depending on the amount of load imbalance.

The contributions of this paper are as follows.

- A significant performance benefit from using a data-driven execution model to tolerating communication. Note that IMPI is unable to mask communication using a split phase implementation based on asynchronous, non-blocking communication [2], [6], [7].
- A novel execution model based on a reinterpretation of the MPI process model to easily manage the heterogeneous parallelism of a hybrid node.
- A simulation that demonstrates that our proposed approach can also efficiently distribute regular workloads onto heterogeneous resources within a node.

II. A case study and experimental Testbed

A. Stampede

Stampede, located at the Texas Advanced Computing Center (TACC), comprises 6400 compute nodes, each equipped with 1 Intel Xeon Phi XE10P (Knights Corner) coprocessor and 2 Intel Xeon E5 8-core processors (Sandy Bridge). Each compute node includes 32GB (4 x 8GB DDR3) of host memory (NUMA) and 8GB of on-board DDR5 device memory. The coprocessor is connected with the host via a PCIe connection. Nodes communicate via a Mellanox FDR

[1] *Reverse offload* is not yet supported on Stampede

[2] At present, symmetric mode is limited to 32 nodes.

[3] The simulation employs MIC-MIC mode, which is at present limited to 64 nodes.

InfiniBand interconnect with a 2-level fat-tree technology. In order to employ the MIC processors, we use the Intel C++ Composer XE 2013 suite (version 13.0), which includes the Intel C++ compiler `icpp`. We use Intel's MPI implementation, IMPI.

At the time of this writing, Stampede employs the initial offering of the MIC architecture, Intel Phi, AKA Knights Corner (KNC). KNC consists of 61 cores, each an Intel Pentium-like processor (2 pipe in-order superscalar design). Each core includes a 512-bit SIMD ALU that can perform 8 double-precision floating-point operations per clock cycle. Processor cores communicate, and access on-chip DRAM, via a 512-bit wide, bi-directional ring bus and 8 memory controllers. Each core has 32 KB of private L1 and a partitioned 512KB L2. To hide memory latency, MIC supports up to 4 hardware threads (AKA contexts) per core. The Intel documentation states that a minimum of 2 contexts is required to maximize performance.

B. A Jacobi iterative solver using a 7-point stencil

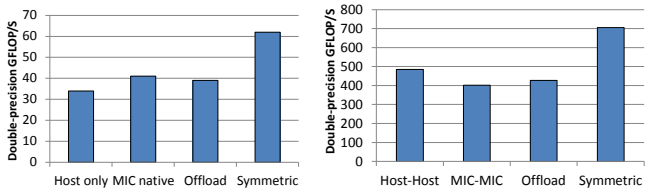
Our motivating application solves the 3D Poisson Equation using Jacobi iterations and a 7-point stencil. The unoptimized kernel of the application comprises 4 loop nests as shown in the following code snippet.

```
1 // V, U, and rhs are N x N x N grids
2 for step = 1 to num_steps{
3  for k = 1 to N-2 //Z
4   for j = 1 to N-2 //Y
5    for i = 1 to N-2 //X: the leading dimension
6      V[k,j,i]= alpha *(U[k-1,j,i]+U[k+1,j,i]+U[k,j-1,i]+U[k
           ,j+1,i]+U[k,j,i-1]+U[k,j,i+1])-beta*rhs[k,j,i]
7   swap(U,V)
8 }
```

7-point Jacobi solver is a well-known memory bandwidth bound application. Although this stencil kernel could be aggressively optimized to become compute bound, we observed that the overall performance at scale is strongly limited by the inter-node communication. As a result, we applied the following optimizations only. We use the OpenMP *collapse* clause to parallelize along the Y axis as well as the Z axis, since parallelizing along the Z axis only is insufficient to fully harness the available parallelism. To exploit the parallelism on the leading dimension, we SIMDize the stencil operations along X. We then apply spatial blocking, dividing the problem into many small tiles so that the working set of each one fits on cache. We further restructure the code with a communication avoiding optimization that unrolls the stencil formula once in the time domain and applies copy propagation optimization to reduce the demand for memory bandwidth [8]. The effect is to reduce the number of compulsory cache misses by a factor of 2, which more than overcomes the added cost of redundant floating-point operations and more L1 references.

C. Exploring the performance of execution models on Stampede

Stampede supports 5 execution modes[4] : *host-host*, *MIC-MIC*, *symmetric*, *offload*, and *reverse offload*. Since *reverse offload* is currently unavailable, we will evaluate the first four modes. *Host-host* mode executes all computations on the hosts and does not use the MICs at all. In *MIC-MIC* mode, the program allocates data and performs computations locally on MICs. *Symmetric* mode supports heterogeneous processing by considering MICs and hosts as peers in one large SMP (symmetric multiprocessor) node. *offload* and *reverse offload* mode work on one resource (device or host, respectively) while migrating computation to the other.



(a) Performance on 1 node. Grid size = 256x512x512

(b) Performance on 16 nodes. Grid size = 1024x1024x1024

Fig. 1. MPI performance with different execution modes. Symmetric mode works best, but could be further optimized.

Fig. 1 presents the performance of the 4 execution modes on 1 and 16 Stampede nodes and shows that symmetric mode significantly outperforms the others. However, we immediately perceive the two limitations of using symmetric mode naively. First, on 1 node the performance of symmetric mode is lower than the total performance on host and MIC. This is because the performance gap between host and MIC can't be averaged with a regular distribution. Second, symmetric mode on 16 nodes only yields 70% of the maximum GFLOPS that it could realize. This modest efficiency demonstrates that the communication overhead is already significant at such a small scale.

D. Transforming MPI code into data-driven form

We use a directive-based solution supported by Bamboo [2] to overcome limitations of symmetric mode. The code snippet in Fig. 2 shows how we annotated the 7-point Stencil with 3 pragmas (*dimension*, *send*, and *receive*) and an optional *olap* pragma. While *dimension* helps Bamboo generate correct code to virtualize MPI processes, *send* and *receive* pragmas indicate that sending and receiving ghost cells are independent activities within a process. For more details of the Bamboo programming model, see [2]. Code generated by Bamboo is a new program represented as a task graph that runs under a data-flow like execution model. This program relies on a task graph library, *Tarragon,* to define, manage, and execute task precedence graphs [3], [9]–[11]. The nodes of a TaskGraph correspond to tasks to be executed. The edges correspond to data dependencies among tasks. Each MPI process will be effectively transformed into a set of tasks that are executed by a group of *worker threads*. The library supports task execution via one *service thread*, including scheduling.

```
1 #pragma bamboo dimension 3
2 #pragma bamboo olap layout Nearest Neighbor (optional)
3 for step = 1 to num_steps {
4  #pragma bamboo receive
5  {     MPI_Irecv and unpack all messages
6  }
7  #pragma bamboo send
8  {     pack data and MPI_Isend all messages
9  }
10 MPI_waitall
11 Stencil update on local grid
12 }
```

Fig. 2. Annotating a synchronous MPI variant (MPI-sync) of 7-point stencil with Bamboo pragmas

[4]Intel uses different terms for some of these modes, but we will stick to the conventions described in TACC documentation.

III. Hiding inter-node communication

In Fig. 1 we observe that communication is costly and this motivates the need to mask it. Our first attempt to manually hide communication fails to improve performance, since unlike other MPI implementations we have worked with [2], IMPI is unable to realize overlap. We then compare Bamboo against the original MPI code.

A. Overlapping mechanism under the hood of Bamboo

Since, at this time, IMPI immediate mode calls do not support asynchronous non-blocking data transfer, we cannot overlap within the framework of MPI alone. We need to extend the MPI framework with multithreading support that can realize the overlap we require [3], [9]–[11]. Bamboo helps us avoid the need to manually restructure the application to use the framework. The resulting code runs as a data-driven program that automatically overlaps communication with computation. The execution of this data-driven code is controlled by distributed runtime systems (RTS).

Within a node, we configure 2 RTSs on each MIC and 1 RTS on each Sandy Bridge processor, where RTS is a multithreaded MPI process. On MIC, we bind the 2 service threads, which take care of communication, of 2 runtime systems to core #0 and #1 respectively. For each runtime system, we then create a worker thread and bind it to a set of 116 virtual processors, which are mapped to cores #2 to #59 in a cyclic fashion. The configuration of the RTS on each Sandy Bridge processor is simpler as follows. We dedicate one core for the service thread and use the rest 7 cores for the worker thread.

B. Experimental evaluation

We first conduct a weak scaling study, fixing the problem size at 384x384x384 per node. Thus, we increase the problem size in proportional to the number of nodes. Fig. 3 shows the results of the 7-point stencil running symmetric mode on up to 32 nodes. The figure compares the results obtained from Bamboo, by applying it to a traditional bulk synchronous implementation (MPI-sync) that uses blocking communication and cannot overlap communication with computation. Since the problem size is sufficiently large, MPI-sync realizes good performance and scalability. In this situation, however, the benefit of Bamboo is still significant: 24% on 16 MICs and 20% on 32 MICs.

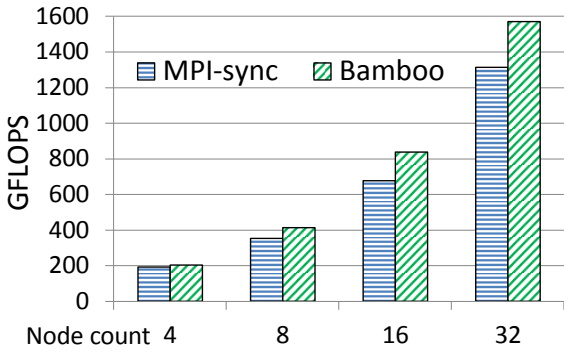
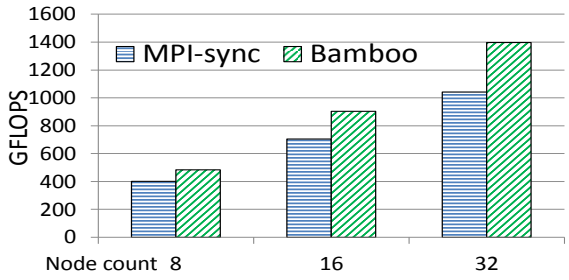


Fig. 3. Weak scaling study: base problem size = 384x384x384. We increase the size equally on 3 dimensions, i.e. the problem size on P nodes is $384P^{1/3} \times 384P^{1/3} \times 384P^{1/3}$. We perform a 3D decomposition in this study.

While we use weak scaling to adapt to large problem sizes and maintain good performance, strong scaling serves as a stress test for the performance of an application when communication cost grows. To this end, we fix the problem size at 1024^3. Results are shown in Fig. 4. In this study, Bamboo plays a more significant role in keeping the application scalable. In particular, *MPI_sync* performs well on 8 nodes. However, this variant slows down on 16 nodes and does not realize good performance on 32 nodes. Performance is now bound by communication, so speeding up the computation time by doubling the number of nodes is offset by the added communication costs. With Bamboo, the increasing cost of communication can be hidden as long as the computation is sufficiently significant. Indeed, Bamboo improves *MPI_sync* by 29% on 16 nodes and 32% on 32 nodes.



(a) Strong scaling performance. Problem size = 1024x1024x1024

Fig. 4. Strong scaling study: we test with two cubical problem sizes: 1536^3 and 1024^3. Since the leading dimension is large, a 3D decomposition is required. We observe that performance under strong scaling is generally lower than under the weak scaling due to increased surface to volume effects.

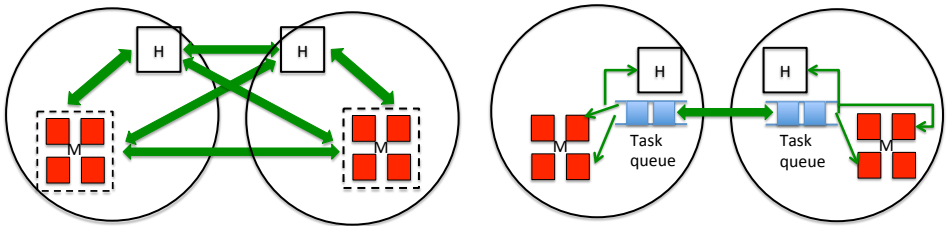
IV. A rectified symmetric mode to manage processor mapping and scheduling

Although hosts and coprocessors coincidentally operate 7-point stencil at comparable rates on Stampede, this fact can change in other applications and on different systems, such as Tianhe-2, which has 3 Intel Phi and 2 Ivy Bridge processors per node [12]. In this section, we propose a new execution model to rectify the host-coprocessor discrepancy so that we can continue using regular distribution effectively on both hosts and coprocessors altogether.

Fig. 5(a) depicts the use of symmetric mode with 2 nodes, where hosts (H) and MICs (M) serve as SMP nodes. The effectiveness of this configuration relies on the assumption that hosts and coprocessors deliver similar performance. However, for reasons stated above, this assumption may not hold.

Since the imbalance occurs within a single node only we can avoid the costs of migrating tasks, which is a more challenging solution to implement. Fig. 5(b) presents a modified version of symmetric mode. This scheme employs only one MPI process running on the host to communicate across nodes. Load balancing within node relies on the virtualization provided by Bamboo and on the dynamic scheduling supported by its runtime. In particular, Bamboo virtualizes MPI processes into many smaller homogeneous tasks. The runtime system employs a single queue per node to handle these tasks. This queue can be configured as a first-come-first-serve or a priority queue. Coprocessor and host serve as workers and keep picking tasks until there is no available task in the queue.

Since the latency between host and coprocessor could be significant compared to the cost of computation, binding tasks at at run



(a) Symmetric mode

(b) A proposed scheme to rectify symmetric mode

Fig. 5. Automatic load balancing in symmetric mode. Virtualization plays a significant role in load balancing regular distributions on heterogeneous processors

time would be crucial to maintain good performance. Specifically, in the first few iterations, host and coprocessor dynamically take tasks from the queue to determine a balanced ratio. Tasks are then bound to host or coprocessor in later iterations to eliminate the overhead of migrating task's data. For irregular problems, hints from the programmer may be useful to perform a optimal task binding or redistributing.

V. Load balancing with rectified symmetric mode

Implementing rectified symmetric mode requires that the host and co-processor residing on the same node see a single unified address space. Implementing such a single address space, either on hardware or by a software solution, requires siginificant efforts. We next present our first step in demonstrating the proposed scheme via a simulation using coprocessors only, which enables us to avoid the thorny implementation issues in supporting the single address space, while demonstrating the utility of the approach.

To simulate the unified address space between host and coprocessor, we extract 1/4 number of cores on each coprocessor to use as host. As a result, each simulated coprocessor consists of 3-time cores more than its simulated host, a reasonable fraction to distinguish multi- and many-core processors. To simulate the fact that each core of the simulated host is faster than each core of the coprocessor, we dynamically add dummy computations to tasks at the time they are scheduled on the simulated coprocessors. We configure Tarragon with 4 worker threads, each spawning 60 OpenMP threads. We use the notion *slowdown factor* σ to denote how slower a worker thread on coprocessor is compared to that on the host. If $\sigma = 1$, all worker threads run with the same rate, meaning that coprocessor is 3-time faster than host. If $\sigma = 6$, then the coprocessor runs at only 3*(1/6)= 1/2 of the host's rate.

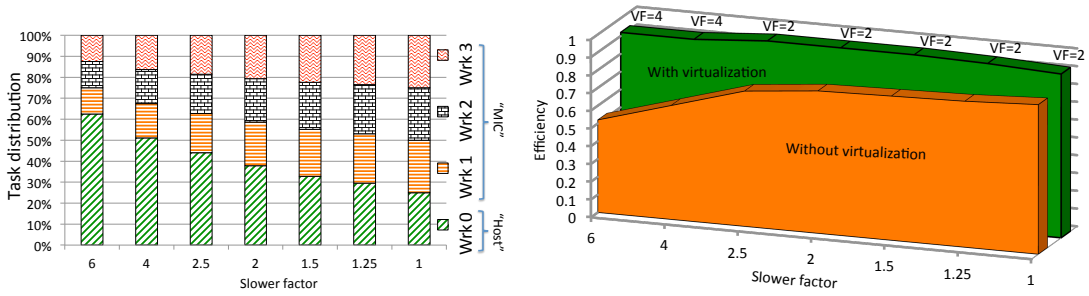
Fig. 6(a) presents the task distribution on worker threads of simulated hosts and coprocessors when the σ varies. We select values for σ so that the coprocessor can be slower or faster than its host. We can see that when $\sigma = 6$, the host's worker thread is very fast compared to those of the coprocessor. In such a scenario, the scheduler dynamically assigns more tasks to the host's worker thread, thereby balancing the workload to maintain good performance. It is important to note that the task scheduling is purely driven by workload and the slowdown factor, and there is no intervention from the programmer. We also observe that the task distribution results shown in Fig. 6(a) hold for both single and multi-node configuration.

Fig. 6(b) demonstrates that latency hiding does not negatively impact load balance, but rather works in synergy with load balancing activity to improve performance. In particular, when $\sigma = 1$, entire performance benefit is due to latency hiding. With higher values of σ the load balancing scheme contributes additional benefit, increasing the amount of performance improvement. In this study, we also see the vital role of virtualization in both hiding latency and balancing the workload. The degree of virtualization is denoted by *virtualization factor* (VF), the total number of tasks divided by the total number of worker threads. Fig. 6(b) shows that a virtualization factor of 2 is sufficient when the discrepancy between worker threads is not significant. However, when the host's worker thread becomes much faster, a virtualization factor of 4 is required. This makes sense since we need more tasks to fill the larger performance gap among worker threads.

VI. Related work

Overlapping communication and computation is a popular technique to hide communication overheads, with a long history [3], [9]–[11], [13]–[22]. However, writing code to achieve overlap is not a trivial process, and this task is even more challenging without proper support from MPI implementations. For example, Witmann et al. noted a significant impact of communication on the performance of a 3D stencil solver on up to 64 nodes of Xeon 5550 processors [6]. However, the authors did not apply any optimization to overlap communication with computation since the Intel MPI implementation (IMPI) that they used did not support asynchronous non-blocking primitives.

Bamboo virtualizes MPI processes into tasks and uses static analysis to extract information embedded in communication primitives to generate a dynamic firing rule. Virtualized tasks then execute under a data-driven execution model provided by Tarragon [9], [10], [22], i.e. running and waiting upon the arrival of data and the firing rule generated by Bamboo. MPI/SMPSs [23] also employs a source-to-source translator and pragma annotations to realize task parallelism using SMPSs [24]. Under MPI/SMPSs programmers *taskify* MPI calls that may then run in parallel with computation, adding additional burden on the programmer. Similar to Tarragon, SMPSs separates communication from task computations. As a result MPI/SMPSs also supports asynchronous transfer and communication-computation overlap. Danalis et al. [25] implemented transformations that realize communication-computation overlap in MPI collectives. β-MPI [26] generates the runtime dataflow graph of an MPI program, in order to assess communication volume. It overloads the MPI calls using macros, but does not perform source code analysis or code restructuring. Adaptive MPI [27], built on top of Charm++ [28]



(a) Virtualization help balance the workload across non-uniform processors

(b) Efficiency (performance/sustainablePerformance) with and without virtualization on 64 simulated nodes

Fig. 6. The role of virtualization in enabling homogeneous programming in a heterogeneous computing environment

supports communication overlap by virtualizing MPI processes, but but performs no translation.

VII. Conclusion

The current trends in node architecture pose a challenge to both application development and maintenance. We have shown that even at a small scale, an accelerator based platform significantly increases communication overheads, and we have proposed a means of mitigating these overheads in stencil methods. We have also shown that employing both host and coprocessor can significantly improve the performance. The use of this approach, however, is complicated due to the heterogeneity of the host-coprocessor communication structure. We proposed a novel execution model based on task virtualization and dynamic scheduling to load balance computations within each node. This solution not only solves the load balancing problem but also hides communication delays. Nevertheless, future hybrid systems will require vastly improved interconnect in order to operate at scale.

Acknowledgment

This research was supported by the Advanced Scientific Computing Research, the U.S. Department of Energy, Office of Science, contracts No. DE-ER08-191010356-46564-95715 and DE-FC02-12ER26118. Tan Nguyen is a fellow of the Vietnam Education Foundation (VEF), cohort 2009, and was supported in part by the VEF. This work used the Extreme Science and Engineering Discovery Environment (XSEDE), which is supported by National Science Foundation grant number OCI-1053575. We would like to thank Stampede's consultants, who provided quick and thoughtful responses to our questions while conducting this research.

References

[1] J. Shalf, S. Dosanjh, and J. Morrison, "Exascale computing technology challenges," in *Proceedings of the 9th international conference on High performance computing for computational science*, ser. VECPAR'10. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 1–25.

[2] T. Nguyen, P. Cicotti, E. Bylaska, D. Quinlan, and S. B. Baden, "Bamboo: translating mpi applications to a latency-tolerant, data-driven form," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. Los Alamitos, CA, USA: IEEE Computer Society Press, 2012, pp. 39:1–39:11.

[3] S. B. Baden and S. J. Fink, "Communication overlap in multi-tier parallel algorithms," in *Proc. of SC '98*, Orlando, Florida, November 1998.

[4] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick, "Avoiding communication in sparse matrix computations," in *Proceedings of International Parallel and Distributed Processing Symposium (IPDPS)*, 2008. [Online]. Available: http://bebop.cs.berkeley.edu/

[5] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2006-183, Dec 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html

[6] M. Wittmann, G. Hager, and G. Wellein, "Multicore-aware parallel temporal blocking of stencil codes for shared and distributed memory," in *Parallel Distributed Processing, Workshops and Phd Forum (IPDPSW), 2010 IEEE International Symposium on*, 2010, pp. 1–7.

[7] G. Hager, G. Schubert, T. Schoenemeyer, and G. Wellein, "Prospects for truly asynchronous communication with pure mpi and hybrid mpi/openmp on current supercomputing platforms," in *Proc. Cray Users Group Conference 2011 (CUG 2011)*, Fairbanks, AK, May 2011.

[8] T. M. Chipeperekwa, "Caracal: unrolling memory bound stencils," Department of Computer Science and Engineering, University of California, San Diego, Tech. Rep., 2013.

[9] P. Cicotti and S. Baden, "Latency hiding and performance tuning with graph-based execution," in *Data-Flow Execution Models for Extreme Scale Computing (DFM), 2011 First Workshop on*, 2011, pp. 28–37.

[10] P. Cicotti, "Tarragon: a programming model for latency-hiding scientific computations," Ph.D. dissertation, Department of Computer Science and Engineering, University of California, San Diego, 2011.

[11] S. J. Fink, "Hierarchical programming for block–structured scientific calculations," Ph.D. dissertation, Dept. of Comput. Sci. and Eng., Univ. of Calif., San Diego, 1998.

[12] J. Dongarra, "Visit to the national university for defense technology changsha, china," Oak Ridge National Laboratory, Tech. Rep., June 2013. [Online]. Available: http://www.netlib.org/utk/people/JackDongarra/PAPERS/tianhe-2-dongarra-report.pdf

[13] A. Sohn and R. Biswas, "Communication studies of DMP and SMP machines," NAS, Tech. Rep. NAS-97-004, 1997.

[14] A. K. Somani and A. M. Sansano, "Minimizing overhead in parallel algorithms through overlapping communication/computation," ICASE, Tech. Rep. 97-8, February 1997.

[15] A. Gupta, G. Karypis, and V. Kumar, "Highly scalable parallel algorithms for sparse matrix factorization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 8, no. 5, pp. 502–520, 1997.

[16] J. D. Teresco, M. W. Beall, J. E. Flaherty, and M. S. Shephard, "A hierarchical partition model for adaptive finite element computation," *Comput. Methods. Appl. Mech. Engrg.*, vol. 184, pp. 269–285, 2000.

[17] K. Teranishi, P. Raghavan, and E. Ng, "A new data-mapping scheme for latency-tolerant distributed sparse triangular solution," in *Supercomputing '02: Proceedings of the 2002 ACM/IEEE conference on*

Supercomputing. Los Alamitos, CA, USA: IEEE Computer Society Press, 2002, pp. 1–11.

[18] P. Raghavan, K. Teranishi, and E. Ng, "A latency tolerant hybrid sparse solver using incomplete cholesky factorization," *Numer. Linear Algebra Appl.*, vol. 10, pp. 541–560, 2003.

[19] N. Chrisochoides, K. Barker, J. Dobbelaere, D. Nave, and K. Pingali, "Data movement and control substrate for parallel adaptive applications," *Concurrency Practice and Experience*, pp. 77–101, 2002.

[20] C. Bell, D. Bonachea, R. Nishtala, and K. Yelick, "Optimizing bandwidth limited problems using one-sided communication and overlap," in *Proceedings of the 20th international conference on Parallel and distributed processing*, ser. IPDPS'06. Washington, DC, USA: IEEE Computer Society, 2006, pp. 84–84. [Online]. Available: http://dl.acm.org/citation.cfm?id=1898953.1899016

[21] J. Sorensen and S. B. Baden, "Hiding communication latency with non-spmd, graph-based execution," in *Proc. 9th Intl Conf. Computational Sci. (ICCS '09)*. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 155–164.

[22] P. Cicotti and S. B. Baden, "Asynchronous programming with tarragon," in *Proc. 15th IEEE International Symposium on High Performance Distributed Computing*, Paris, France, Jun. 2006, pp. 375–376.

[23] V. Marjanović, J. Labarta, E. Ayguadé, and M. Valero, "Overlapping communication and computation by using a hybrid mpi/smpss approach," in *Proceedings of the 24th ACM International Conference on Supercomputing, ICS '10*, 2010, pp. 5–16.

[24] J. Perez, R. Badia, and J. Labarta, "A dependency-aware task-based programming environment for multi-core architectures," in *Cluster Computing, 2008 IEEE International Conference on*, 2008, pp. 142 – 151.

[25] A. Danalis, K.-Y. Kim, L. Pollock, and M. Swany, "Transformations to parallel codes for communication-computation overlap," in *Proceedings of the ACM/IEEE SC 2005 Conference*, November 2005, pp. 58–68.

[26] R. Silva, G. Pezzi, N. Maillard, and T. Diverio, "Automatic data-flow graph generation of mpi programs," *Computer Architecture and High Performance Computing, 2005. SBAC-PAD 2005. 17th International Symposium on*, pp. 93–100, 24-27 Oct. 2005.

[27] C. Huang, O. Lawlor, and L. Kalé, "Adaptive mpi," in *Proc. 16th International Workshop on Languages and Compilers for Parallel Computing (LCPC 03)*, 2003.

[28] L. V. Kale and S. Krishnan, "Charm++: a portable concurrent object oriented system based on c++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, ser. OOPSLA '93. New York, NY, USA: ACM, 1993, pp. 91–108.