TACC Technical Report IMP-24

IMP as a vehicle for software/hardware co-design

Victor Eijkhout and John McCalpin*

September 27, 2016

* {eijkhout,mccalpin}@tacc.utexas.edu, Texas Advanced Computing Center, The University of Texas at Austin

Abstract

Overview It is becoming increasingly clear that continued performance scaling of parallel computers at the rates to which we have been accustomed (i.e., 500x or more per decade for the performance of a machine at any rank on TOP500 list) will require a fundamental rethinking of architecture and implementation of both hardware and software. This not merely a problem for exascale systems, but is pervasive across the HPC ecosystem – increased application performance at any scale now depends primarily on increasing the number of processor cores used, rather than on using cores of increasing performance.

For example, from the TOP500 results of 1993 to 2003, almost 80% of the increase in aggregate performance came from increases in per-processor performance, while for the decade of 2003 to 2013 over 70% of the increase in aggregate performance came from simply increasing the number of processor cores.

Exascale computers will only be feasible if major components of the software/hardware stack are fundamentally redesigned to reduce both purchase price and power costs.

Based on an extrapolation of recent hardware trends, exascale systems in the near term (e.g., 2018) could cost well over $1B to purchase and could consume in excess of 100MW of power. while the core counts of even the smallest systems in the TOP500 list already exceed 10,000 and are growing at more than 50% per year.

Limitations on power density dictate that increased performance be obtained via increased thread parallelism using increased numbers of relatively loosely-coupled, using relatively simple and relatively low-frequency processing cores. In the past, this sort of parallelism was only one of many contributors to performance increases, with processor frequency and processor instruction-level parallelism making comparable contributions. In the future, ”loosely-coupled” parallelism may be required to deliver more than 100% of the desired performance increases, as frequencies are decreased to reduce energy consumption per unit of computational work. Single-core performance is approximately stagnant in high-performance systems, so to first order, all future performance increases must come from increased parallelism.

This paper argues that current processor and system design is poorly suited to efficient large-scale parallelism, and that this is a direct result of retaining architectural assumptions that are fundamentally misaligned with the complexity, performance, and power balances of the current (and projected) technology base. As a consequence, more and more resources (design, silicon, and power) are required to deliver even small improvements in end-user performance, price/performance, or power/performance from

the continuing improvements in semiconductor process technology, while the difficulty of programming large-scale parallel systems continues to increase.

We have identified several performance and functionality limiters that are due primarily to architectural decisions – mostly related to the inability to either "see" or control data motion in the memory hierarchy and data motion related to communication.

Thus, a fundamental rethinking of hardware and software appears overdue. First, we propose fundamental changes to the hardware architecture related to data motion through the memory hierarchy and data motion associated with communication and synchronization. – making transfers through the memory hierarchy explicit and controllable via higher level semantics, and separating the architectural functionality of communication and synchronization operations from those of private memory transfers. Second, we propose that the newly developed *Integrative Model for Parallelism* programming model can significantly increase programmability while effectively exploiting the capabilities introduced by the revised hardware architecture.

Rather than raising the abstraction level this model skews it: models need to become more global in expression, but need the ability to explicitly control data motion and exploit fine-grained synchronization.

The following IMP reports are available or under construction:

IMP-00 The IMP Elevator Pitch
IMP-01 IMP Distribution Theory
IMP-02 The deep theory of the Integrative Model
IMP-03 The type system of the Integrative Model
IMP-04 Task execution in the Integrative Model
IMP-05 Processors in the Integrative Model
IMP-06 Definition of a 'communication avoiding' compiler in the Integrative Model (under construction)
IMP-07 Associative messsaging in the Integrative Model (under construction)
IMP-08 Resilience in the Integrative Model (under construction)
IMP-09 Tree codes in the Integrative Model
IMP-10 Thoughts on models for parallelism
IMP-11 A gentle introduction to the Integrative Model for Parallelism
IMP-12 K-means clustering in the Integrative Model
IMP-13 Sparse Operations in the Integrative Model for Parallelism
IMP-14 1.5D All-pairs Methods in the Integrative Model for Parallelism (under construction)
IMP-15 Collectives in the Integrative Model for Parallelism
IMP-16 Processor-local code (under construction)
IMP-17 The CG method in the Integrative Model for Parallelism (under construction)
IMP-18 A tutorial introduction to IMP software (under construction)
IMP-19 Report on NSF EAGER 1451204.

IMP-20 A mathematical formalization of data parallel operations
IMP-21 Adaptive mesh refinement (under construction)
IMP-22 Implementing LULESH in IMP (under construction)
IMP-23 Distributed computing theory in IMP (under construction)
IMP-24 IMP as a vehicle for software/hardware co-design

1 Introduction

This proposal addresses the integrated problem of revising the fundamental assumptions of hardware architecture to address the increasing energy and performance costs of data motion and the increasing importance of communication and synchronization in parallel applications, while simultaneously developing a rigorously formulated programming model that facilitates the development of correct programs and that is amenable to automatic program transformations.

Scientific computing has an undiminished appetite for high performance, currently aiming for exaflop performance around the year 2020. The hurdles to be overcome on that road concern both software and hardware:

- Software has no integrated way of dealing with the diversity of current parallel computer designs, which consists of a mix of distributed memory, shared memory, attached co-processors, light-weight threads, SMT threads, et cetera. While all these modes of parallelism are individually programmable, a unified solution is both lacking and needed.
- Large-scale systems are increasingly power-hungry, to the point that an extrapolation of current power budgets projects an exascale machine at hundreds of megawatts. A redesign is needed that reduces the both cost and power demand by significant factors, while improving scalability to allow increased overall performance.

Most of the traditional approaches to increase performance have reached plateaus. For example, comparing the TOP500 lists of 1993 to 2003, we find that almost 80% of the increase in aggregate performance came from increases in processor (core) performance, while for the decade of 2003 to 2013 over 70% of the increase in aggregate performance came from simply increasing the number of processor cores. The total number of cores represented by the systems on the TOP500 list has been increasing at almost 55% per year for the last decade and there is little reason to believe that application scalability has been able to keep up with such rates. At the same time, the increasing complexity of processors and the increasing disparity between processor and memory performance has reduced the extent to which user applications can exploit the peak performance improvements and increases in core count that are provided with new product generations.

We argue that the inability to exploit the reduced power consumption and increased core count of HPC systems is due to a mismatch between architectural assumptions and technological realities: **Programming models continue to assume a "flat-memory" cache-coherent architecture with weakly interacting Von Neumann processors, and hardware continues to present this interface, but the actual implementations are increasingly unlike this idealized model**. Hence we need to rethink both hard-

ware and programming: hardware needs to be endowed with a new set of semantics, and programming models need to allow expression of both data motion and data dependence, while supporting provably correct program transformations to map to the variety of target system configurations.

As examples of this architectural mismatch, we note: (1) Current hardware architectures do not provide visibility or control over data motion through the memory hierarchy – even though memory latency is now 100 times more expensive (relative to computational rates) than it was two decades ago. Data transfer is also becoming increasingly costly in energy use compared to computation [11]. (2) Current hardware architectures provide no explicit mechanisms to support communication and synchronization operations. Synchronization using shared memory is extraordinarily inefficient, with optimized implementations on idle systems requiring hundreds of cycles – even between cores sharing a last-level cache on a single die. Under more typical circumstances these values quickly rise into the thousands or tens of thousands of cycles.

Optimization of these expensive operations will require both a programming model that can specify data motion and inter-process communication, and efficient hardware mechanisms (with exposed interfaces) to implement them.

Our Integrative Model for Parallelism (IMP) naturally expresses data dependencies, which can be interpreted as either data motion or task synchronization. In [**?**] we argued how the IMP model can be interpreted in terms of several existing programming systems. In particular, an algorithm described in our model can be translated to any of these systems. Because of this we propose to adopt and adapt the IMP model to exploit data motion and synchronization primitives that redesigned hardware exposes to user space programming.

> **We argue that hardware can be made more efficient by making existing hardware functions visible and controllable.** Exposing these hardware primitives requires them to be programmable, so **we propose to adapt the IMP software model to enable it to exploit these primitives**. This proposal argues that a joint software and hardware angle of attack is needed; our research will be to investigate the co-design of hardware capabilities and software expression, and the trade-offs involved.

Most of our proposal will focus on the data motion aspects of the co-design of the user-level programming model and the hardware architecture responsible, taking into account both data motion through a memory hierarchy and data motion related to communication and synchronization across parallel tasks. To a lesser extent we consider execution scheduling and threading issues.

In this proposal we will study the redesign of software and hardware, leading to **prototype sofware, and hardware simulators**, that evaluate the extent to which plau-

sible modifications to existing architectures might provide the desired functionality and performance. We note that our goal is analysis leading to insight, rather than the production of complete new hardware designs or complete software stacks.

2 Illustrations of the Impact of Current Architectural Shortcomings

We argue that current hardware architecture is not well matched to the current technology in several important ways. Many of the problems fall into the categories of controlling "vertical" data motion through the memory hierarchy and controlling "horizontal" data motion associated with communication between processes.

In the next sections we present examples of some of the ways in which current architectures do not provide an adequate base to deal with the performance and power issues associated with vertical data motion (sections 2.1 and 2.2) and horizontal data motion (section 2.4). This is not intended to be a comprehensive review, but as an illustration of why we have concluded that these problems are architectural, rather than being shortcomings of specific implementations.

2.1 Analysis of "vertical" data motion

Over the past two decades, the "cost" of data motion (in terms of execution time) has steadily risen in relation to the "cost" of arithmetic. The historical data provided by the STREAM benchmark results database [16] documents this clearly for the case of long vector operations, with current systems capable of performing 30-60 floating-point operations in the time required to transfer one 64-bit word from memory. When considering latency, the ratios are much worse, with current cores able to perform 500-1000 floating-point operations in the time required for a non-prefetched access to main memory. Despite these huge performance ratios, data transfers through the memory hierarchy have still not been made visible or explicitly controllable in current architectures.

We can review the consequences in more detail by considering a specific processor. Here we use values for the 'Intel Xeon E5-2680 processor' from the TACC Stampede supercomputer. This is an 8-core, dual-socket processor based on Intel's "Sandy Bridge" processor core. Each Sandy Bridge core typically runs at 3.1 GHz (the maximum "Turbo" mode frequency supported by this model) and can execute up to 8 double-precision floating-point arithmetic operations per clock cycle (one 4-element-wide addition and one 4-element-wide multiplication) for a peak of 24.8 GFlops per core.

If we assume a DAXPY kernel ($x_i = x_i + s \times y_i$), each iteration performs two 8-byte reads and one 8-byte write, or 24 Bytes for 2 FP operations. Therefore a single core would require almost 300 GBytes per second of load plus store traffic to fully utilize the arithmetic units (for this kernel).

At the DRAM level, each processor chip has 4 DDR3/1600 DRAM channels, providing an aggregate of 51.2 GByte/s of peak memory bandwidth. For the DAXPY kernel, this peak bandwidth of 51.2 GB/s would support 4.266 GFLOPS – only 17% of the peak performance of one core, and only 2% of the peak performance of the eight cores.

In an ideal world, one would be able to use one core to run DAXPY at full speed while the other cores rest in a low-power state or work on other computational tasks. Unfortunately one core cannot generate enough concurrent cache transactions to fill the memory pipeline. At a nominal (idle) load latency of 79 ns and a bandwidth of 51.2 GB/s, basic queuing theory dictates that $79 \times 51.2 = 4044.8$ bytes of traffic be "in flight" at all times. This corresponds to 64 cache lines – but a single core can only directly support 10 outstanding L1 cache misses[1]. Therefore at least 7 of the 8 cores must be used to generate enough L1 cache misses to tolerate the memory latency. A more detailed analysis includes the effect of the L2 hardware prefetch engines, the difference in buffer occupancy between loads and stores, and the increase in buffer occupancy due to contention in the memory system, but extensive experimentation has shown that these factors mostly cancel out and that the maximum bandwidth achievable (about 76% of peak) requires the use of 6-8 cores per chip, even though only one core is needed to perform the arithmetic.

We argue that this is not necessary, and that including data motion as a primary architectural feature would allow a single core to request the full bandwidth available to the chip (as well as the full bandwidth available on the chip-to-chip and IO links).

> We conclude that the current data transfer mechanism is ill-suited to both energy-efficient performance and effective utilization of the processing resources. Our proposed solutions will involve architecturally visible mechanisms to specify data motion (with significantly higher-level semantics than the existing transparent mechanisms), and abandoning transparent caches (and their implicit 'pull' mechanism) as the primary mechanism for data transfer in favour of managed local storage and a push mode. Of course, to exploit this we need a new programming model that makes data motion explicit.

2.2 The Power Cost of "Vertical" Data Motion

In section 2.1 we saw that using individual cache misses to generate adequate memory concurrency carries a heavy penalty – as many as **7 out of 8 cores are being "wasted"**,

using most of the power consumed by the system and unable to contribute to other computational tasks.

Power measurements using the Intel "RAPL" (Running Average Power Limit [4]) show a consumption of 100 Watts when running the STREAM benchmark, with 72 Watts consumed by the 8 cores, 17 Watts by the "uncore" (L3 cache, memory controller, QPI interconnect, IO controller, and other management logic), and 11 Watts used by the DRAM. An architecture capable of specifying adequate concurrency using a single core could power down seven of the eight cores, for a savings of up to 56 Watts (56% of the total power consumption).

Once the un-needed cores can be powered down, other power consumption issues can be addressed. The next largest consumer of power is the "uncore", at 17 Watts (39% of the remaining 44 Watts). We can identify several mechanisms that are not needed in the improved architecture and which can be disabled or removed – detailed analysis of these opportunities is part of the proposed research. These un-needed mechanisms include cache coherence, large memory reorder buffers, and hardware prefetch engines.

After the "uncore", the next largest remaining consumer of power is the DRAM. The measured efficiency of 39 GB/s for 11 Watts is about 26 mW/Gbs (or 26 pJ/bit), which is good level of efficiency for this technology. With improved scheduling, we will demonstrate that the **DRAM utilization can be increased** from the observed 76% to well over 95% with a modest improvement in efficiency.

2.3 Power cost of instruction handling

Finally, the single core needed to perform the arithmetic does not require the 9 Watts measured in the Xeon E5-2680 system. As much as **half of the power used by the core is expended by the scheduler and instruction retry mechanisms** [3]. These would not be needed by a core exploiting programmed data motion for which unpredictable memory delays are much less frequently on the critical path. Power consumption could also be reduced by reducing the frequency of the remaining core until its cache bandwidth and compute capability are no higher than what is needed to process the incoming data. Simpler cores such as the ARM Cortex-A9 could be augmented with double-precision floating-point hardware to deliver arithmetic rate required by the available bandwidth in a budget of 1-2 Watts.

> Between more efficient data transfer and in-order calculation, power savings of 80%-90% should be attainable. We will argue below how hardware with redesigned semantics and a new programming model facilitate this.

2.4 "Horizontal" Data Motion – Synchronization and Communication

Parallel programs must include some type of communication and some type of synchronization, yet these are not present as basic concepts in current processor architectures. Although communication and synchronization can be implemented as side effects of ordered memory references, such an approach is necessarily inefficient and extremely difficult to optimize in hardware.

Communication Historically, cache coherence protocols have been optimized to maximize exploitation of spatial and temporal locality for a single thread of execution. They do this reasonably effectively by keeping recently used data "close" to the processor. Cache coherence protocols allow operation of shared memory systems by serializing access to modified cache lines, and by providing a set of ordering rules that make it possible to write code that reliably conveys data with controllable ordering between processes.

The *ability* to communicate via shared memory does not mean that it is an *efficient* means of communication. The properties desired for optimizing communication are almost the inverse of those desired for exploiting spatial and temporal locality. Instead of keeping data "close", a protocol for communication should "push" data from the producer to the consumer as quickly as possible. Instead of serializing access to modified cache lines and requiring multiple coherence transactions to "hand off" data from a producer to one or more consumers, a protocol for communication should support "single-hop" messages, broadcasts, and higher-level constructs such as FIFOs. These are either not possible, or incur high development and validation costs, if they are layered on top of memory operations, rather than being included as an independent set of functions by the architecture.

Synchronization Synchronization is another fundamental concept in any model of parallel programming, that is not included as a first-class concept in current processor architectures. Although implementable via side-effects of ordered memory references on standard systems, synchronization via shared memory is intrinsically inefficient and suffers from poor scalability. The resulting overheads of several thousand cycles to several tens of thousands of cycles have contributed to the failure of previous programming models based on fine-grained parallelism. The existence of extremely efficient synchronization mechanisms in hardware (with O(1) cycle latency) makes it clear that this is an issue of architecture, not of physics, and is therefore a critical topic that we will address. (We should note that such concepts were proposed decades ago, e.g, [6], but seem to have been dropped along the way.)

This mandatory coherence incurs costs in several dimensions. In addition to power and complexity, **mandatory coherence increases memory latency** and thereby increases the amount of concurrency required to fill the memory pipeline. As an example, the Xeon E5-2680 processors used in the previous examples have a local memory latency of 79 ns when running at their maximum speed of 3.1 GHz in their dual-socket configuration. The latency would be about 20% lower in a single-socket configuration, and could be reduced another 15% if memory accesses did not have to check the caches before going to memory. The combined effect would reduce the concurrency required to fill the memory pipeline by almost 1/3. As an illustration that a large core count, single socket, design is not the solution, the Intel Xeon Phi SE10P with 61 cores on a single chip has a local memory latency of 275 ns, largely due to the complex multi-level protocol required to maintain coherence for the 61 caches.

3 Possible solutions to the hardware problem

We will now outline several approaches that may alleviate the problems discussed above. We will see that, to a varying extent, solving the hardware problems requires changes to the programming model, while several desirable changes to the programming model require changes to the hardware to be effective.

3.1 Incremental Hardware Improvements

When criticizing a system, it is generally wise to consider incremental solutions, rather than complete redesigns. Incremental solutions are typically considered to be cheaper to implement and easier to exploit by incremental modifications to the existing code base.

Incremental approaches are possible for several of the issues mentioned above. Some approaches to explicit control over cache coherence (e.g., [5]) could provide a modest reduction in memory latency, a corresponding decrease in concurrency requirements, and a decrease in the the power consumption associated with cache coherence. Other approaches (e.g., [15, 10]) could improve communication and synchronization performance.

However, implicit approaches are largely at the point of diminishing returns. For example, processors could also be designed with support for a larger number of cache misses, but a significantly larger miss handling structure could easily increase the L2 cache hit latency, which is likely to be unacceptable for many applications. Hardware prefetchers could be made more aggressive, but this can also easily lead to performance degradation due to false prefetches and eviction of needed data from the caches.

The bandwidth/latency/concurrency examples above were both optimistic (based on idle-system latency), and pessimistic (not assuming cache reuse or aggressive hardware prefetchers).

We can improve the situation by make the optimistic numbers more reliable, for instance by **reducing the latency to memory**. The figure of approximately 80ns can easily be reduced to 60 by losing coherence between the multiple sockets. Coherence between cores is an important cost factor in high-core-count chips such as the Intel Xeon Phi, and relaxing coherence there can greatly reduce latency.

This type of coherence is rarely needed in scientific computations. However, to guard against its accidental use, one needs a programming model that avoids it.

One could also increase the actual concurrency by addressing the **LFB!**s (**LFB!**s). Increasing the number of them would increase the L2 latency, which is undesirable since the L2 is typically the major source of reuse. Wider **LFB!**s have been tried in the DEC Alpha processor, and are certainly possible. Some issues with cache line alignment, overlap with other **LFB!**s, and crossing page boundaries exist, but these are not major. Optimistically, a programming model that keeps explicit track of data can address these issues.

3.2 Software approaches

Much work has been done in the past decade on programming in a way that more effectively uses the existing hardware. (Cache and register reuse [7], prefetch streams [14], DRAM banks [8].) The problems with this are twofold. Practically, reports of performance improvement are all *proof of concept*, and can not be integrated into applications without considerable effort. Secondly, as with the above incremental approaches, they **both do not come close bridging the enormous gaps described** in section 2.

3.3 The need for fundamental solutions

Our major problem with incremental solutions is that they do not directly address the shortcomings of the underlying architecture, and therefore do not put us on a path toward effective exploitation of projected hardware technologies. Although some of these approaches may provide some benefit, they provide it in most cases by making the most complex existing features of the architecture even more complex. Due to the complexity of current processor designs, very few of the proposed mechanisms in the literature are actually implementable, and once the detailed implementation is accounted for, few of the remaining approaches would be helpful if implemented. In the end, physics wins, and only architectures designed to provide control over the

most expensive operations (in power or latency) are likely to be suitable foundations for cost- and power-effective large-scale parallel systems.

The solutions just mentioned do not address the fundamental shortcomings of the underlying architecture. Therefore, **we advocate redesigning hardware and software in tandem**, achieving both higher hardware efficiency and programmer productivity.

In the hardware case this means making visible essential concepts, and exposing already existing mechanisms to programmability in user space. For software is means developing a vocabulary to address these now exposed features.

We briefly tabulate the aspects of our software/hardware co-design research.

Hardware redesign	Problem addressed	Programming model capability	Research
data motion semantics	power efficiency, effective bandwidth	semantic aggregation	5.1
non-coherent caches	latency, power, complexity	explicit treatment of data distribution	5.2
managed local memory	latency, power, precise control	explicitly known data dependencies	5.3
low overhead synchronization	fine-grained threads	explicit description of parallelism and dataflow	5.4

We also summarize in Figure 1 our envisioned processor design: the basic block diagram of the processor stays the same (although we probably reduce the number of cache levels to one), but all components get a different semantics. Programming these semantics is the work of the programming model that we will discuss next.

3.3.1 Explicit Control of Data Motion

Technology projections suggest that (once we adopt more efficient processor architectures) the energy cost of data motion will become the dominant factor in overall power consumption, and perhaps also in total lifetime system cost [13].

We therefore propose to improve the efficiency of data motion by making data motion architecturally explicit, allowing various components of a system to communicate about data motion at a relatively high semantic level.

Rather than making a sequence of individual requests for the physically addressed cache lines containing data in various virtually addressed data structures, a processor should be able to convey a description of the entirety of its data needs for a phase

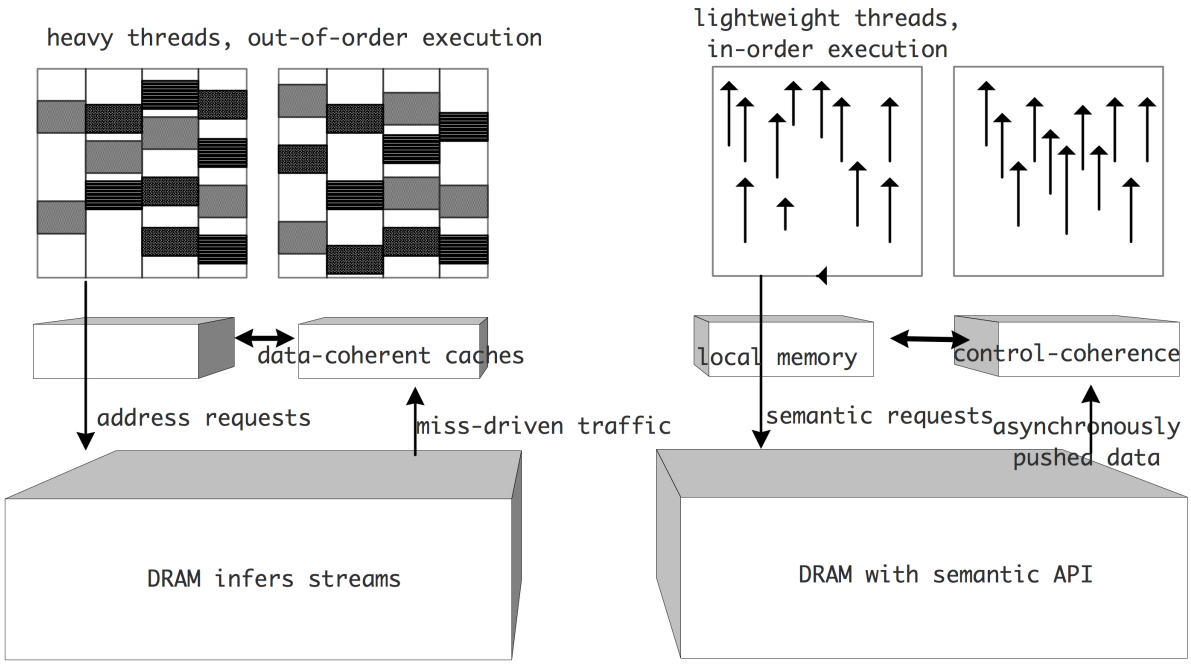

Figure 1: Old and new processor structure

of execution to other processing elements. These "other processing elements" may include (potentially heterogeneous) general-purpose processors, (potentially heterogeneous) specialized processors, memory controllers, IO controllers, communications controllers, and power management hardware.

One way to make data movement far more efficient is to add considerable semantic information to it. Above we already remarked that prefetch streams are able to alleviate our above analysis. Unfortunately, these streams are in general inferred by the memory system, rather than programmed in user space. Although software prefetch instructions exist, they provide little semantic content and they interact with the (minimally-documented) hardware prefetchers in complex and often unpredictable ways. Most importantly, the software prefetch instructions do not increase the maximum memory concurrency available to a processor, nor do they provide guarantees that the prefetched data will still be in cache when it is actually needed.

(The IBM Power architecture is an exception here.) *John, how about the x86 prefetch instruction?*

Possible hints to the prefetches could include the number of independent streams and their lengths, whether the stream is for read or write. More sophisticated annotations

are possible: for instance in a sparse matrix-vector product the stream of indices, consisting of 32-bit integers, is consumed at half the rate of the 64-bit floating point data.

3.3.2 Managed local memory

The most far-going solution to our problem is largely to abandon the notion of caches, and replace that with explicitly managed 'scratchpad' local memory, or 'local memory' for short. (Local memory being explicitly managed, the notion of coherency naturally disappears.) Such local memory can have similar bandwidth and latency characteristics as caches, as evinced by the TI TMS320 family of DSP chips or the Cell processor that was used in the Los Alamos Roadrunner supercomputer.

Local memory solves the above signalled memory bottleneck problems:

- Data movement is no longer effected by cache misses generated by an operation; data can be placed ahead the operations that need it, reducing or eliminating the latency to memory.
- Data is loaded from local memory with uniform latency/bandwidth behaviour, so complicated out-of-order instruction scheduling can be abandoned. This will make the individual core simpler and thereby more power-efficient.
- Cores no longer need complicated mechanisms for maintaining coherence. This reduces latency, and simplifies their design.

3.3.3 Synchronization

In section 2 we mostly concentrated on bandwidth and power issues for a single instruction stream. However, processors are gradually becoming more threaded, with the Intel Xeon Phi having hardware support for four threads, and larger numbers likely in the future. Below we will see how the IMP programming model can generate fine-grained tasks organized in a DAG; for this we want efficient exposed synchronization mechanisms.

Note that GPUs handle fine-grained tasks only in the sense of fast context switching; they lack efficient fine-grained control over data dependencies between threads.

4 Solutions through a programming model

In this section we will discuss the Integrative Model for Parallelism (IMP) model for parallel programming; see [9, **?**] for further details. Here we limit ourselves to the aspects of the model that interact with hardware design.

4.1 Brief introduction

IMP is a theoretical model for expressing parallel algorithms. It operates on a high level of abstraction, but it requires the programmer to specify enough about parallelism to make these concepts translatable to efficient code; see below for proof of this contention.

The IMP model is based on distributions of data and work. Unlike many other programming systems, IMP has distributions as first-class objects and provides an algebra of operations on and transformations between distributions. Thus, **IMP translates to an intermediate representation expressed in a rich vocabulary of data motion concepts**. The IMP model is flexible enough to handle traditional Cartesian structures, but also sparse data and even distributions with redundant duplication. (That latter option means that resilience is formally expressible in IMP through manipulating the distribution mechanism.)

The theory behind IMP derives data dependencies through formal transformations between distributions. Specifically, each parallel operations (a 'kernel') is based on the distributions on the input and output objects, as well as a distribution that is defined by the structure of the calculation. Reconciling these three distributions gives tasks and task data dependencies as formally derived objects.

Since IMP is based on formal reasoning, it is not dependent on any parallelism model, and in fact it can be shown to generalize both distributed memory message passing and shared memory task models. We summarize this workflow in figure 2. As a

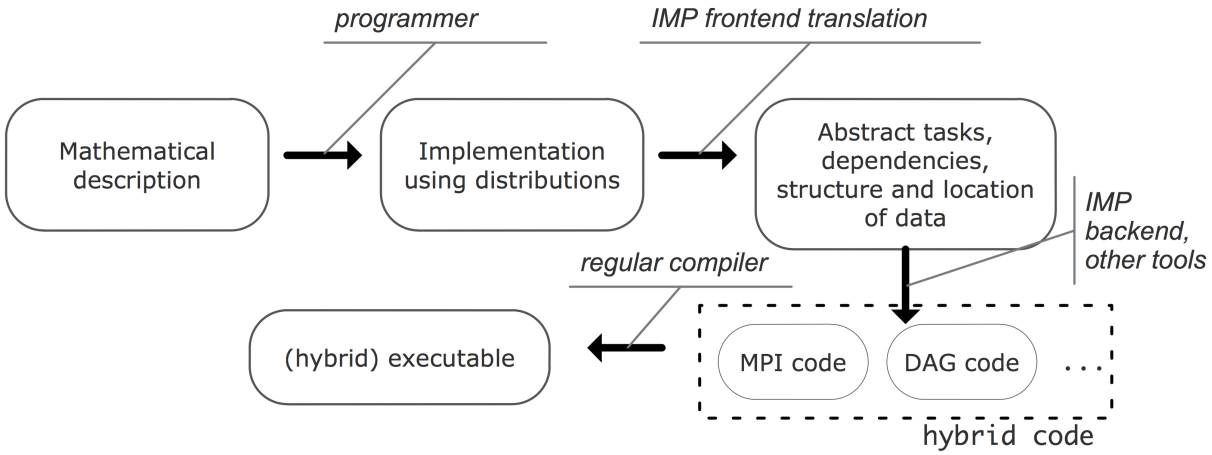

Figure 2: Block diagram of programming with IMP

proof-of-concept of the IMP framework we implemented a simple example, the one-

dimensional heat equation, as a C++ class library along the IMP formalism, and compiled it twice, once as MPI code, and once using OpenMP tasks [**?**]. We compared this code to manually produced reference codes. Figure 3 shows that both with OpenMP

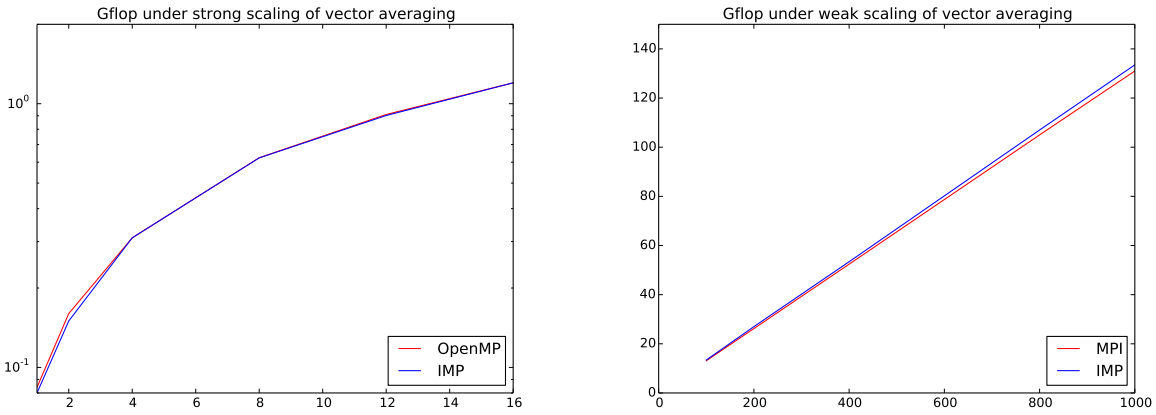

Figure 3: Scaling behaviour of OpenMP and MPI realizations of the example IMP code, compared to manually produced reference codes.

and MPI the performance behaviour, possibly after startup phenomena, is identical within 2–3%.

While we target two parallel programming backends here, the IMP model can as easily accomodate hybrid forms of parallelism. In that case, IMP code would be realized as a mix of these backends, and not using any PGAS system.

4.2 IMP as a basis for co-design

Above we reasoned that the poor performance on contemporary processors is due to deficient semantics of the processor/memory interface. The IMP model can remedy this, since it can formally derive all data dependencies in a parallel algorithm.

For instance, an IMP analysis of an algorithm not only knows what data is needed in a kernel, it also knows its structure and when it is available. This means for instance:

- An IMP kernel can provide semantic informa-

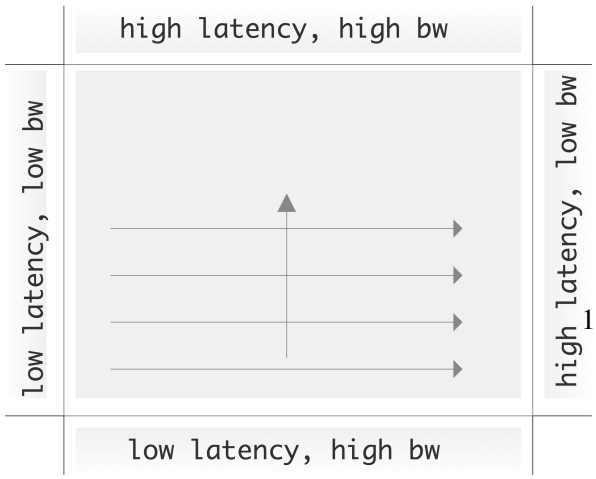

Figure 4: Illustration of different transfer demands on halo data for lexicographic stencil update

tion on requested data to the DRAM controller, allowing it to schedule an efficient transfer. Possible DRAM semantics could include whether data is contiguous or strided, but also latency and bandwidth properties such as its anticipated absorption rate; see figure 4.

- Rather than data being transferred, triggered by cache misses, IMP can realize a 'push' model of data transfer. This replaces current strategies of 'latency hiding' by '**latency prevention**'. As a result, we can replace caches with much more energy-efficient local memory, and we can largely dispense with out-of-order instruction handling, which is to extent motivated by the unpredictable manner in which data traditionally arrives at the functional units.

The analysis of a single IMP kernel yields a full description of all of its data dependencies: this includes the specification of what tasks or processors produced the data, and an aggregate description of the layout in storage. This latter description makes it possible to provide the DRAM controller with a semantically rich description of data to be provided.

The work of Co-PI McCalpin in [8] indicates that addressing DRAM inefficiencies can markedly improve performance. Our IMP programming model can supply the necessary information; **our proposed research will address enhanced DRAM semantics**: the design of an API for this information, a formalism for deriving it, and a design and simulation of an improved DRAM controller.

4.3 Related software efforts

The IMP model shares characteristics with several other programming models. We highlight the following.

Charm++ (see for a recent exposition [12]) is a system based on active messages. Thus it provides a mechanism for dataflow and synchronization analogous to IMP. Concurrent Collections [2] and Quark [20] are programming systems for explicit dataflow programming.

While such systems typically incorporate heuristics for minimizing data movement, they do not offer the possibility of explicitly supplying information to guide this analysis. Furthermore, these systems require explicit programming of the dataflow, rather than deriving it from a high level description. This means they are both semantically poorer and harder to program than IMP.

5 Proposed research

We propose the following research activities as the focus of our software/hardware co-design.

5.1 Explicit Control of Data Motion through the Memory Hierarchy

The key enabling technology for improving hardware utilization and power efficiency is to provide explicit control for data motion through the memory hierarchy. I.e., rather than making a sequence of individual requests for the physically addressed cache lines containing data in various virtually addressed data structures, a processor should be able to convey a description of the entirety of its data needs for a phase of execution to other processing elements. These "other processing elements" may include (potentially heterogeneous) general-purpose processors, (potentially heterogeneous) specialized processors, memory controllers, IO controllers, communications controllers, and power management hardware.

In addition to its direct benefits, providing explicit control of "vertical" data motion opens up opportunities to architect more efficient mechanisms to exploit locality than traditional transparent caches.

Research Topic 1: Through analysis and simulation we will investigate the costs and benefits of enabling a processor to convey additional semantic information about memory transfers to caches, memory controllers, and other processor cores. For bandwidth-limited codes, the goal is to fully utilize memory bandwidth while activating as few

computational resources as possible. Full memory bandwidth utilization requires both exposing enough concurrency to tolerate latency and providing the memory controller with enough information about future access patterns so that it can schedule effectively. Exposing concurrency is simple if the access patterns are known in advance. DRAM scheduling is a bigger challenge, as it is well known that DRAM scheduling often degrades as the scheduling window is widened. The research challenge is to design an **architecture that enables information to be conveyed** from the application and the compiler to the memory controller, and a **programming model to generate this information**.

Research Topic 2: For compute-limited codes, the goal of explicitly managed data transfer is to minimize loss of compute cycles due to unexpectedly delayed data (e.g., cache misses), and to minimize latency in cases where it is physically unavoidable – i.e., when the algorithm is unable to compute an address until immediately before it is needed. Thus the proposed system will be evaluated for absolute latency as well as for throughput.

Research Topic 3: It is often the case that a finely tuned system lacks robustness. We will evaluate the ability of the high-level semantic interface and co-adapted memory scheduler to tolerate unexpected requests (e.g., RDMA requests from remote processors) of various sizes and priorities.

5.2 Coherence

Cache coherence is a fundamental attribute of current systems, and proposals to remove cache coherence – even in carefully proscribed circumstances – are typically met with considerable resistance. Therefore an important requirement of this research project is to document the costs of cache coherence as accurately as possible.

Research Topic 4: We will continue our studies of the impact of cache coherence on main memory latency and use the results to evaluate the impact of this added latency on performance and power consumption of high-bandwidth application kernels.

Research Topic 5: Through detailed analysis of fundamental algorithms and selected HPC codes of importance to the TACC user base, we will investigate the programmatic need for cache coherence, and how the IMP model can circumvent this.

5.3 Explicitly Managed Local Memory

Explicit control of data motion and explicit control of cache coherence can alleviate some of the power and performance limitations of current architectures, but explicitly managed local memory is expected to provide significantly better controllability and significantly lower power.

Research Topic 6: Using available simulators (such as CACTI [19]), we will evaluate the differences in latency, throughput, and power consumption for explicitly managed local memories in comparison to caches.

Research Topic 7: We will study the extent to which the input/output dependence functionality of IMP can be used to specify and control data motion through an explicitly managed memory hierarchy and evaluate its expressiveness for common algorithms used in HPC.

5.4 Synchronization

We observed in section 2.4 that user space synchronization is very expensive, but that fast synchronization mechanisms are embedded in the hardware of every CPU. There is no physical reason that this performance can not be made available in user space, as was shown over 20 years ago [6].

The current architectural insistence on transparent caching and side-effect-free memory access has blocked efforts to revisit these approaches. We are willing to break these assumptions, which will open up opportunities for approaches that are more than incremental. Hardware-based FIFOs, hardware barriers, loads with full/empty (or valid/invalid) bits, user-level interrupts, and hardware work queues may all play a role. We will focus on relatively high-level mechanisms that can be implemented efficiently in hardware and which directly support programs based on specification of data dependencies.

Research Topic 8: We will review and analyze the semantic requirements for synchronization in the context of fine-grain parallel programs, evaluate the mechanisms listed above for their applicability to different algorithms, and research the generation of such semantics in the IMP model.

5.5 Cost model

One critical long-term goal of this project is to enable the development of hardware and software that supports accurate cost modeling. Having an accurate cost model for a parallel code makes it possible to have software layers that engage in optimization, for instance through reordering memory requests, task scheduling and migration, or redundantly duplicating tasks. The two factors standing in the way of this are hardware and software, and we aim to address both.

Current hardware behaves unpredictably, making a priori cost estimates of operations hard to impossible to make. User level software expresses communication insufficiently as independent entities, Our research will lead to **hardware with analyzable semantics** and, as a result, software will be capable of doing analytic optimization, rather than heuristic or empiric.

We will investigate what aspects of **cost can be efficiently modeled and dynamically scheduled**. This will rely on the hardware simulations that we will develop in this project, as well as development in the IMP model.

6 Evaluation plan

This project aims to show the feasibility of more efficient hardware, and software to exploit it. Thus we will engage in design, exploration, and simulation. Neither actual hardware design or the delivery of a full software system fits this scope; instead we will gauge the success of our research by **producing hardware simulators, and prototype software**.

The success of our co-designed software/hardware stack can be evaluated as follows.

Programming model capabilities The end product of our software research will be a vocabulary of concepts expressing data motion and task synchronization. We will show that such concepts can be derived in the framework of the IMP model, both by theoretical analysis and by proof-of-concept prototype software.

Hardware semantics Our hardware research will show the feasibility of efficient hardware that implements more explicit, and higher level, semantics than is currently used. We will simulate this hardware using existing and newly written simulators.

In particular, we will deliver detailed computations of the power required by our design, contrasting this with existing processor designs at similar effective performance.

Co-PI McCalpin has performed this type of analysis in industry as well in recent publications [17, 18].

References

[1] *Intel 64 and IA-32 Architectures Optimization Reference Manual*.

[2] Michael G. Burke, Texas Kathleen Knobe, Ryan Newton, and Vivek Sarkar. The concurrent collections programming model. Technical Report TR 10-12, Department of Computer Science Rice University, 2010.

[3] Jie Chen, Guru Venkataramani, and Gabriel Parmer. The need for power debugging in the multi-core environment. *IEEE Computer Architecture Letters*, 11:57–60, 2012.

[4] Intel Corporation. *Intel64 and IA-32 Architectures Software Developer's Manual, Volume 3: System Programming Guide*, 2012.

[5] W.J. Dally. System and method for explicitly managing cache coherence, March 29 2012. US Patent Application 13/243,948.

[6] W.J. Dally, J.A.S. Fiske, J.S. Keen, R.A. Lethin, M.D. Noakes, P.R. Nuth, R.E. Davison, and G.A. Fyler. The message-driven processor: a multicomputer processing node with efficient mechanisms. *Micro, IEEE*, 12(2):23–39, April 1992.

[7] Jim Demmel, Jack Dongarra, Victor Eijkhout, Erika Fuentes, Antoine Petitet, Rich Vuduc, R. Clint Whaley, and Katherine Yelick. Self adapting linear algebra algorithms and software. *Proceedings of the IEEE*, 93:293–312, February 2005.

[8] J. Diamond, M. Burtscher, J.D. McCalpin, Byoung-Do Kim, S.W. Keckler, and J.C. Browne. Evaluation and optimization of multicore performance bottlenecks in supercomputing applications. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 32 –43, april 2011.

[9] Victor Eijkhout. A theory of data movement in parallel computations. *Procedia Computer Science*, 9(0):236 – 245, 2012. Proceedings of the International Conference on Computational Science, ICCS 2012, Also published as technical report TR-12-03 of the Texas Advanced Computing Center, The University of Texas at Austin.

[10] S. Fide and S. Jenks. Architecture optimizations for synchronization and communication on chip multiprocessors. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–8, April 2008.

[11] International Technology Roadmap for Semiconductors. International technology roadmap for semiconductors, 2012 update. Technical report, International Technology Roadmap for Semiconductors, 2012.

[12] Laxmikant Kale and Jonathan Lifflander. Controlling concurrency and expressing synchronization in Charm++. Technical Report 13-36, Parallel Programming Lab, UIUC, 2012.

[13] Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, Katherine Yelick, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Jon Hiller, Stephen Keckler, Dean Klein, R. Stanley Williams, and Katherine Yelick. Exascale computing study: Technology challenges in achieving exascale systems. Technical Report TR-2008-13, University of Notre Dame, 2008.

[14] Jaekyu Lee, Hyesoon Kim, and Richard Vuduc. When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1):2:1–2:29, March 2012.

[15] J.D. McCalpin and P.N. Conway. Push for sharing instruction, January 17 2012. US Patent 8,099,557.

[16] John D. McCalpin. Stream: Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2013. A continually updated technical report.

[17] Ardavan Pedram, John McCalpin, and Andreas Gerstlauer. Transforming a linear algebra core to an FFT accelerator. In *Proceedings of the 2013 IEEE 24th International Conference on Application-Specific Systems, Architectures and Processors (ASAP)*, pages 175–184, 2013.

[18] Ardavan Pedram, John McCalpin, and Andreas Gerstlauer. A highly efficient multicore floating-point FFT architecture based on hybrid linear algebra/FFT cores. Revised manuscript in review at the Journal for Signal Processing Systems, 2014.

[19] S. Thoziyoor, Jung-Ho Ahn, M. Monchiero, J.B. Brockman, and N.P. Jouppi. A comprehensive memory modeling tool and its application to the design and analysis of future memory hierarchies. In *Computer Architecture, 2008. ISCA '08. 35th International Symposium on*, pages 51–62, June 2008.

[20] A. Yarkhan, J. Kurzak, and J. Dongarra. QUARK users' guide: Queueing and runtime for kernels. Technical Report ICL-UT-11-02, University of Tennessee Innovative Computing Laboratory, 2011.