Analysis of Parallel Programming Models for Exascale Computing

Sergio Martin

Abstract—High-performance computing has become an essential part of developments in scientific and technologically important problems. However, as we approach the age of exascale computing, developers of large scale scientific applications will have to face new challenges. In this study, we focus our attention on reducing the costs of communication in supercomputers and analyze the parallel programming models and tools aimed at helping programmers address this challenge.

I. INTRODUCTION

Supercomputers are used in a wide range of fields such as: weather forecasting [54], medicine [59], computer aided design [25], military [57], and simulation of natural disasters [28]. As applications become larger and more sophisticated, so does their need for increasingly powerful supercomputers. Fortunately, the continuous improvement in both computer architecture and interconnect has led to an exponential growth in performance.

Fig. 1 shows how the performance of the top 500 supercomputers has increased since 1995, growing by an order of magnitude in performance roughly every decade, closely following Moore’s law [45]. Currently, the most powerful computer, Sunway TaihuLight located at the National Supercomputing Center in Wuxi, China, delivers a staggering peak performance of 125 petaflops (10^{15} floating point operations per second). The coming milestone – the exaflop (10^{18} floating point operations per second) supercomputer is on the horizon.

Exascale computing poses new challenges for scientific application programmers [6]. We focus our attention on challenge of reducing the costs of communication. Addressing this challenge will be essential for exascale performance and involves two aspects: the cost of internal communication and the cost of network communication.

We refer to internal communication to all data transferred through main memory or cache structures of an individual computer. An inefficient use of these structures can reduce memory bandwidth and have a significant impact on performance. The cost of internal communication plays an important role in the performance of individual computers, and thus in the performance of supercomputers as a whole.

We refer to network communication to all data transferred among nodes through a supercomputer’s interconnect. As interconnects become larger and more complex, it is expected that the average routing latency among any two computers in exascale systems will increase way beyond that of current petascale systems. It will therefore be necessary for programmers to implement mechanisms to reduce or hide this cost as much as possible in order to realize exascale performance.

Although optimizations to ameliorate internal and network communication costs can be introduced manually by a programmer, this could entail an excessive amount of effort. In this study, we survey the parallel programming models and tools (language extensions, translators, and compilers) that have been proposed for providing these optimizations with a minimum effort. We analyze the mechanisms implemented by each model and how they contribute in realizing efficient applications for exascale computing.

The rest of this paper is organized as follows: in §II we provide an overview of the current state of supercomputing. In §III we explain the costs of communication in supercomputers. In §IV we analyze the Message Passing model and its limitations. In §V we present alternative models based on the Message Passing model. In §VI we introduce the Partitioned Global Address Space model. In §VII we introduce the Asynchronous PGAS model. In §VIII we introduce dataflow models. In §IX we discuss how these models will help realizing exascale computing. Finally, in §X we present our conclusions.

II. SUPERCOMPUTING OVERVIEW

Fig. [1] shows how the performance of the top 500 supercomputers has increased since 1995, growing by an order of magnitude in performance roughly every decade, closely following Moore’s law [45]. Currently, the most powerful computer, Sunway TaihuLight located at the National Supercomputing Center in Wuxi, China, delivers a staggering peak performance of 125 petaflops (10^{15} floating point operations per second). The coming milestone – the exaflop (10^{18} floating point operations per second) supercomputer is on the horizon.

Exascale computing poses new challenges for scientific application programmers [6]. We focus our attention on challenge of reducing the costs of communication. Addressing this challenge will be essential for exascale performance and involves two aspects: the cost of internal communication and the cost of network communication.

We refer to internal communication to all data transferred through main memory or cache structures of an individual computer. An inefficient use of these structures can reduce memory bandwidth and have a significant impact on performance. The cost of internal communication plays an important role in the performance of individual computers, and thus in the performance of supercomputers as a whole.

We refer to network communication to all data transferred among nodes through a supercomputer’s interconnect. As interconnects become larger and more complex, it is expected that the average routing latency among any two computers in exascale systems will increase way beyond that of current petascale systems. It will therefore be necessary for programmers to implement mechanisms to reduce or hide this cost as much as possible in order to realize exascale performance.

Although optimizations to ameliorate internal and network communication costs can be introduced manually by a programmer, this could entail an excessive amount of effort. In this study, we survey the parallel programming models and tools (language extensions, translators, and compilers) that have been proposed for providing these optimizations with a minimum effort. We analyze the mechanisms implemented by each model and how they contribute in realizing efficient applications for exascale computing.

The rest of this paper is organized as follows: in §II we provide an overview of the current state of supercomputing. In §III we explain the costs of communication in supercomputers. In §IV we analyze the Message Passing model and its limitations. In §V we present alternative models based on the Message Passing model. In §VI we introduce the Partitioned Global Address Space model. In §VII we introduce the Asynchronous PGAS model. In §VIII we introduce dataflow models. In §IX we discuss how these models will help realizing exascale computing. Finally, in §X we present our conclusions.
A. Node Architecture

Modern supercomputers are built using a large set of nodes connected through a high-speed interconnect. A node is the minimal set of physical components capable of functioning and performing computation. A node also represents the minimal addressable unit in the network, with its own IP address and hostname. Nodes in a supercomputer are typically server-grade computers that contain: (a) one or more multicore and/or manycore processors, (b) a main memory (RAM) system, and (c) network interconnect.

Fig. 2 represents the processor topology and memory hierarchy of a compute node in the Edison supercomputer located at the National Energy Research Scientific Computing Center. We use Edison as a representative example for our explanation in this section.

Edison’s nodes contain two 12-core Intel “Ivy Bridge” processors each, identified as sockets P#0 and P#1. A socket is the physical placeholder for a processor chip on the node’s motherboard that provides connectivity to other components of the node (e.g. RAM, PCI-e channels, network device).

Each core in Edison is able to run two threads simultaneously. A thread represents a stream of instructions to be executed, plus its execution status (i.e. registers, stack pointers, program counter). The ability of processor cores to execute two or more threads simultaneously is called Simultaneous multi-threading [69]. Edison defines the logical placeholders for the execution of threads inside each core as Processing Units (PU), numbered from 0 to 47, two per core. The operating system uses the PU number to assign the mapping of threads to PUs.

Each core contains its own L1 (instructions+data) and L2 cache, while a common L3 cache is shared between cores if the same socket. All cores have access to the entire main system memory space. However, given physical constraints, processor cores have different access times to/from different segments of main memory, called Non Uniform Memory Access (NUMA) domains.

Cores allocated in the same NUMA domain are guaranteed to have the same access time (assuming no contention). However, cores accessing data residing in a different NUMA domain will suffer from performance degradation. In the case of Edison nodes, main memory (64Gb) is divided into 2 (NUMA) domains, each with 32Gb of memory and all the cores from a socket belong to the same NUMA domain.

B. Many-Core Devices

One of the reasons why the peak performance of top supercomputers have consistently increased in recent years is the addition of massively parallel (many-core) processors into their computing nodes. These devices base their potential in implementing processor chips with a large number of cores, many more (thousands in GPUs) than conventional multi-core processors (dozens). In order to fit such numbers of cores in the processor dye, many-core devices have simpler core designs that span less area than conventional cores.

Many-core devices are specially targeted for compute-intensive, highly parallel algorithms. That is, algorithms that can divide the problem domain in many fine-grained partitions that can be computed in parallel with minimal additional synchronization or communication overheads. Algorithms requiring that all cores execute the same instruction over different sets of data represent ideal cases.

Although many-core processors are key to increasing the peak performance of supercomputers, harnessing their power still represents a challenge for the following reasons: (i) They represent an additional layer of complexity for programmers. (ii) Communication between the host and the device requires an extra copy of the data since they cannot access each other’s memory space. (iii) The increase in computational power puts additional pressure on main and device memories. While we do not address this challenge in our study, it could provide the basis for an extensive analysis.

C. Interconnect Design

Supercomputers comprise a huge number of interconnected computing nodes. For this reason, network and systems architects need to implement efficient network topologies. That is, find ways to organize nodes, routers and cabling in order to minimize the cost of communication while keeping power and money budgets constrained. Folded Clos (fat-tree) [40] network topologies, proven to be efficient in smaller scales, would incur a prohibitive cost for exascale supercomputers because of the number of routers and cabling complexity required. Such network would dominate the costs of a supercomputer, both in budget and energy costs [37].
Several highly-scalable topologies for petascale and exascale supercomputers have been proposed to reduce the complexity of the interconnect. The Flattened Butterfly [38] and Dragonfly [39] topologies have proven to be less costly in the number of nodes (see Fig. 3). The use of optical cables in global channels allow the Flattened Butterfly and Dragonfly topologies achieve a similar bandwidth and latency as folded Clos networks, with highly reduced router and cabling costs.

Edison uses a dragonfly topology, organized in a four rank hierarchy [49]. Rank 0 consists of 4 nodes allocated inside one blade. Communication is routed through a high-speed custom-designed integrated circuit (ASIC) that serves as the main gateway for all 4 nodes; Rank 1 consists of 16 blades allocated inside one chassis. Communication is routed through high-bandwidth wires across a circuit board that connects all blades ASICs; Rank 2 consists of 3 chassis allocated inside one cabinet. Communication is managed by a blade router connected by copper cables, and; Rank 3 represents the entire supercomputer (5576 nodes, in total) and uses high-bandwidth optical cables and routers.

The dragonfly topology in Edison guarantees that data packets sent from node to node perform a maximum of 4 hops. A hop represents each transmission of a data packet between intermediate routers and end-nodes. The average number of hops in the execution of a distributed application is an important factor in our analysis of the cost of communication in the next section.

D. Classification of Scientific Algorithms

The main use for modern supercomputers is in simulating physical phenomena. Although there exists a broad range of scientific applications, researchers from the University of California, Berkeley and the Lawrence Berkeley National Laboratory identify 7 common computation motifs, known as Phil Colllela’s 7 dwarfs, that can be used to classify them [15, 5]. Each one of these motifs have different patterns for communication, computation and storage, as described below:

(i) Structured Grids. These algorithms partition the problem domain (multidimensional space) into a grid of discrete elements. The elements of the grid are distributed as a mesh of rectangles. The solution is approached by iteratively applying a mesh sweep over every element, where computation and communication patterns are regular. That is, all elements have the same amount of neighbors that they communicate with, and demand the same amount of computation. A special kind of structured grid, called Adaptive Mesh Refinement, shown in Fig. 4 (left) allows a finer division of the grid in areas of particular interest. The complexity of these algorithms is typically $O(I * N^d)$, where $I$ is the number of solver iterations, and $N$ is the number of elements in a dimension (assuming a square grid), and $d$ is the number of grid dimensions.

(ii) Unstructured Grids. This is a variation of (i) where the grid is partitioned based on the physical elements being modeled. The density and shape of grid elements can be tailored to fit the shapes of physical objects (e.g. the 3D mesh of a surface of an airplane wing), as shown in Fig. 4 (right).

(iii) Dense Linear Algebra. These represent operations between matrices and vectors, where all the elements of such data sets are involved in the calculation. The complexity of these algorithms vary from $O(N)$ for vector-scalar operations, to $O(N^3)$ for matrix-matrix operations, where $N$ is the number of elements in a dimension (assuming square matrices).

(iv) Sparse Linear Algebra. This is a variation of (iii) in which algorithms are optimized for matrices/vectors filled with mostly zero values. Since the presence of zeroes reduce the amount of computation required, these algorithms implement compression techniques to reduce the memory footprint, and keep track of the distribution and location of zeroes.

(v) Particle / N-Body Methods. These algorithms simulate the interaction between discrete points. In general, the properties (position/speed) of every point is calculated as a function of the properties every other point (position/mass/charge). For this reason, these algorithms have a $O(N^2)$ complexity, where $N$ is the amount of points. Variations of this motif, however, exploit the spatial locality of points to reduce the complexity of these algorithms to $O(N \log N)$ [24].

(vi) Spectral Methods. These represent numerical algorithms that transform data in the frequency domain to/from time/spatial domains. The typical example of these methods is the Fast Fourier Transform and its inverse, with an $O(N \log N)$ complexity, where $N$ is the amount of elements to transform.

(vii) Monte Carlo Methods. These algorithms compute statistical result of random trials. Monte Carlo algorithms scale well with the number of processors because random trials are independent from each other and can be executed in parallel with negligible communication.
III. COST OF COMMUNICATION IN EXASCALE

Due to the unprecedented number of processing elements involved and the complexity of memory hierarchies, achieving exaflop performance will demand extraordinary efforts from system designers, interconnect architects, and programmers alike. In this section we analyze the sources of communication overhead that programmers will need to address to realize exascale performance.

A. Internal Communication

We refer to internal communication as data transfers between cores of the same node through main memory or cache structures inside an individual compute node. In this section we analyze the sources of internal communication that affect the performance of large-scale applications.

While nodes keep incorporating more processors and cores, the performance of main memory technology fails to scale accordingly. This is known as the memory wall, and represents a limiting factor for many scientific applications. The memory wall is caused by the growing performance gap between processors and main memory, as illustrated in Fig. 5. The performance of an application can be severely reduced by limitations in memory bandwidth and latency. This problem is exacerbated by multicore processors, since each additional core puts additional pressure on the main memory. Current petaflop systems are particularly affected by this problem, and it is expected that this will be a performance bottleneck in exascale systems as well.

The roofline performance model [72] is an intuitive way to visualize the maximum attainable performance of an application over a specific memory/CPU architecture. This model shows how memory bandwidth can limit an application to perform below the peak performance of the CPU.

A roofline diagram, shown in Fig. 6, uses the arithmetic intensity of an application (measured in flops/byte) as input. This value indicates how many floating point operations are executed per byte transmitted from/to memory. The point at which arithmetic intensity meets the roofline function – delimited by the maximum memory bandwidth and CPU peak performance – indicates the application’s attainable performance (measured in flop/s).

The example estimates the performance of two applications. Application A has a higher arithmetic intensity (right side of the diagram) and thus is only limited by the peak CPU Gflop/s. On the other hand, application B has a low arithmetic intensity (left side of the diagram) and requires a higher transfer rate from/to memory. In this case, performance is limited by memory bandwidth.

Different computational motifs have characteristic arithmetic intensity patterns. Dense linear algebra algorithms involving matrix operations, for instance, perform $O(N^3)$ floating point operations, while loading $O(N^2)$ elements from memory. This means that, for large enough matrices, the number of floating point operations is larger than the bytes loaded from memory. For this reason, these algorithms are mostly bound by the processor performance. Structured and unstructured grid algorithms, on the other hand, involve close to one floating point operation per data element. For this reason, these algorithms are typically bounded by memory bandwidth.

One way to increase the performance of memory-bound algorithms is to make an efficient use of intermediate cache structures. Cache structures are much faster than main memory, and get faster the closer they are located from cores (e.g. In Edison, L1 cache is 5x faster than L3 cache). A good use of cache can decrease the number of accesses to main memory, increasing the attainable performance given a certain arithmetic intensity. One well known technique, cache blocking, involves changing an algorithm’s memory access patterns to reuse data in cache lines as much as possible before it is replaced. The effect of implementing this optimization is that the attainable performance of an application would move closer to the Cache Bandwidth line in Fig. 6 allowing, for instance, application B to attain a higher performance.

While optimizations like cache blocking are specific to each algorithm, there are factors that can affect memory performance in general. The choice of parallel programming model, for instance, can have a significant impact on the arithmetic intensity of an algorithm. In our analysis of programming models, we identify two main causes for this:

(i) Data duplication. This occurs in models that require the use of (send/receive) buffers for the communication of messages between threads. In case no data hazards (read-after-write, write-after-read, or write-after-write) exist in
the program semantics, the data could be copied/accessed by multiple threads without the need of buffering. Not only unnecessary buffering reduces the memory bandwidth used for actual computation, but also the efficiency of cache structures.

(ii) Misuse of shared memory. When two threads execute in the same node or NUMA domain, they can be optimized to work on the same address space without the need of explicit communication operations. Using a programming model that enables the use of shared memory can reduce the pressure on main memory. However, some programming models work under the assumption that no threads share the address space, making it difficult to use shared memory.

These problems are main contributors to the cost of internal communication and addressing them will therefore be crucial for exascale performance and therefore a central point of discussion throughout our study.

B. Network Communication

Despite the success in creating scalable interconnects, there is still much space for improvement. As the size of a network grows, so does the number of average routing hops ($H$) that messages require to reach their destination, increasing overall latency. The cost of latency will become an important challenge in exascale computing for scientific applications.

The time taken for a message ($T$) to be transmitted from one node $a$ to another node $b$ can be estimated with the formula in Eq. 1.

$$T_{a,b} = L_{a,b} + \frac{S_m}{B_{max}}$$  \hspace{1cm} (1)

Where $L_{a,b}$ is the network latency between the two nodes, $S_m$ is the size of the message, and $B_{max}$ is the maximum bandwidth of the network. We can see that, for small $S_m$, the cost of latency dominates the overall cost of communication.

The latency between nodes $a$ and $b$ can be calculated as a function of the number of routing hops between them ($H_{a,b}$) times a cost per hop ($h$), and a fixed overhead per message ($s$), as shown in Eq. 1.

$$L_{a,b} = H_{a,b} \times h + s$$  \hspace{1cm} (2)

The fixed overhead per message can be caused by both software (e.g. a communication library can require filling a software buffer before sending) and hardware (e.g. per-data packet startup time). A large cost of $s$ can significantly impact performance on algorithms that require sending many small-sized messages. On the other hand, the average value of $H_{a,b}$ can become significant in large-scale executions, thereby making latency a predominant component in the cost of communicating each message.

Some computational motifs are susceptible to the cost of latency. Spectral methods (FFT), for instance, require all-to-all communication patterns where all nodes need to communicate with each other regularly, making these algorithms difficult to scale efficiently. Other motifs, such as Monte-Carlo methods have a negligible communication cost and can scale almost perfectly with the number of nodes.

Although new technologies will continue to improve interconnect performance in the coming years (e.g. Intel OmniPath [60] promises a 40% reduction in latency for new petascale systems), it will still be necessary for programmers to explicitly reduce the cost of communication.

To achieve this, programmers can implement mechanisms that enable Computation/Communication Overlap (over- lap, for short) [1] [61]. This optimization can increase CPU usage and reduce the impact of communication and I/O delays by keeping cores performing useful computation while data is being transmitted.

Some algorithms, such as communication-optimal dense matrix multiplication and LU factorization [63], have been manually optimized to realize overlap. The problem with implementing optimizations for overlap by hand is that this requires an excessive amount of effort by programmers. The alternative is to use tools that can implement mechanisms to support overlap automatically or with little effort from programmers. We found that mechanisms such as oversubscription (i.e. executing more processes than processing cores, discussed in [VIII-C]) and data dependency-driven execution (i.e. out-of-order execution of code based on the availability of data, discussed in [VIII-C]) can be used to automatically enable overlap, without a painstaking refactoring of a program’s code.

In the following sections we discuss the parallel programming models that have been proposed for large-scale computing. We analyze their features and limitations, and how they address the challenges presented in this section. We start with the common approach which is based on the Message Passing Model. In subsequent sections, we analyze alternative models.

IV. THE MESSAGE PASSING MODEL

Because large-scale supercomputers do not provide a physically coherent global memory address space, programmers need to handle communication between processes residing in different address spaces across the network. Several communication models and libraries have been proposed to build distributed applications, with Message Passing being the most widely used.

Message Passing Interface (MPI) [21] is the de facto standard for writing high performance applications on distributed memory computers. Its first specification, MPI-1, presented in 1995, contained a set of 128 C/Fortran functions that provided a basic support for message passing communication between processes. Subsequent releases (now at MPI-3.1 [22]) were aimed at expanding the model in response to new ideas and research made in more recent parallel programming models. In this section we discuss the principles of operation and limitations of MPI.

A. Spatial Decomposition of MPI Programs

An MPI program instantiates as a set of processes that execute autonomously, allowing them to realize parallelism both within and across computer nodes indistinctly. Thanks to this autonomy, an MPI application that executes correctly in a multicore processor can – assuming no bugs or semantic errors – also execute correctly in the millions of cores of a supercomputer [7].

Another reason why MPI has become so widely used is that it makes it easy to develop scientific applications. In its simplest configuration – where all processes execute the same program – MPI provides a natural way to describe applications under the Single Program, Multiple Data (SPMD) execution model.

2Not to be confused with the Simple Instruction, Multiple Thread (SIMT) model used in GPUs.
double U, Uprev;
allocate(U,(N\times N) / ProcessCount + GhostCellsSize);
allocate(Uprev, (N\times N) / ProcessCount + GhostCellsSize);

for (int step = 0; i < iterations; step++)
{
    MPI_Isend(&Uprev[BoundaryCells], up, down, left, right);
    MPI_Irecv(&U[GhostCells], up, down, left, right);
    MPI_Waitall();

    for (int i = 0; i < N; i++)
    {
        for (int j = 0; j < N; j++)
            U[i][j] = \( U[i-1][j] + U[i+1][j] + U[i][j-1] + U[i][j+1] - 4 \times U[i][j] \);
    }
    swap(&U, &Uprev);
}

Fig. 8: Pseudocode of an MPI iterative Jacobi solver, simplified for clarity.

SPMD is a commonly used model to describe the spatial decomposition of scientific applications. The idea behind SPMD is that the workload is divided into smaller parts, which can be computed in parallel using the same program.

Structured grid algorithms such as partial differential equations solvers represent typical examples of SPMD applications. These solvers use finite difference methods, which calculate a new value of a particular point in the grid as a function of the value of itself and its neighboring points.

While a sequential algorithm would sweep over the whole grid applying the stencil one element at a time, an SPMD parallel version of the program divides the grid into smaller parts that can be calculated in parallel by every process using the same solver.

Fig. 7 illustrates a parallel execution of the solver using, for example, 9 MPI processes to solve a 2D grid. In this case, the grid is decomposed equally into sub-grids of size \( N/3 \times N/3 \) that are distributed among all 9 processes. At every iteration, each process applies the sequential solver to all the elements of its sub-grid and communicates boundary cells to/from neighboring processes, required to satisfy the data dependencies expressed by the stencil.

Fig. 8 shows the MPI code of a Jacobi solver based on a 5-point stencil (up, down, left, right, and center points). In lines 2-3, each process allocates and initializes its \( U \) and \( Uprev \) buffers. \( U \) stores the sub-grid for the current iteration, and \( Uprev \) stores the sub-grid for the previous iteration. Each sub-grid contains an extra set of rows/columns, called ghost cells, used to store the boundary cells of neighboring sub-grids. Boundary cells are exchanged at the start of each iteration (lines 7-8) and execution waits until all communication has finished (line 9). Finally, every process applies the stencil over its own sub-grid (lines 11-14) and swaps its sub-grid pointers (line 16).

B. Communication in MPI

To exchange boundary cells, our example uses the two-sided communication protocol. In a two-sided operation, both sender and receiver processes need to explicitly participate in the exchange of a message. The protocol requires that every send operation requested by the sender process is matched by a recv operation in the receiver process, as illustrated in Fig. 9a.

MPI also supports explicit one-sided communication. In explicit one-sided communication, an MPI process can perform read/write operations on the address space of another process without the need of a matching call. MPI provides an explicit interface comprising put/get functions to update/read values on a process’s window, a uniquely identified partition in its private address space.

Fig. 9b illustrates how windows are used to communicate data between MPI processes. In this example, two operations are performed: process 0 executes a put operation to copy data from its private space to the window in process 1, which, in turn, executes a get operation to retrieve data from the window in process 2 to its own space.

One disadvantage of the two-sided and explicit one-sided communication mechanisms is that they do not not support having two processes access on the same memory space directly. In both cases, a data buffer is copied from the sender process into a buffer/window in the receiver’s space. Furthermore, a third copy of the message in an intermediate buffer might be needed when the send operation is posted before its recv counterpart. This becomes a source of unnecessary data duplication when communicating processes belong to the same physical address space, where buffers could instead be shared.

The cost of buffering has an impact on the per-message cost we have seen in Eq. 2. Filling data buffers increase the fixed overhead (\( s \)) associated with the message latency cost (\( L_\text{m} \)). The consequence for a larger \( s \) is that the performance of applications with fine-grained communication (i.e. requiring the communication of large number of small sized messages) are particularly affected.

MPI provides an extension for shared memory that allows processes to access memory allocations in the
same physical address space directly. This extension is, of course, limited to processes co-located in the same node and cannot be used as the only means of communication among all MPI processes across the network. Although this solves the problem of data duplication inside a node, it forces programmers to use two different interfaces for communicating among MPI processes regarding whether they are co-located or not. This is called the MPI+MPI model, which we analyze further in § V.

C. Temporal Decomposition of MPI Programs

Another determinant factor in the performance of large-scale MPI applications is how they deal with network latency. Most scientific applications authored using MPI follow the Bulk Synchronous Parallelism (BSP) execution model [70]. BSP provides a temporal model of how processes compute and communicate. Under this model, the behavior of an application is defined in supersteps. At each superstep, each process performs one, or a combination of two substeps: (1) compute, and (2) send/receive partial results. The data required for the next superstep will not be available until all processes finish their current superstep. This means that there is an implicit barrier at the end of each superstep where all processes synchronize.

The BSP model can be used to model many scientific application motifs. MPI provides the communication and synchronization capabilities to easily realize BSP applications. The problem is that, in the BSP model, communication and computation are performed in distinct phases, inhibiting any overlap between them.

Fig. 10 illustrates the plausible execution of an MPI process that performs computation and communication in two separate phases. The effective core usage (i.e. the time spent performing actual computation) is shown as solid blocks. The diagonal striped blocks represent communication operations. An upwards arrow indicates the start of a communication request, and downwards arrow represents its completion. We can see that a processor core remains unused while waiting for communication operations to complete.

Applications that execute computation and communication in separate stages suffer from the full cost of communication overhead. This represents an important liability for petascale and exascale systems where the latency of communicating a message can become a performance bottleneck.

A programmer can refactor a BSP program to overlap computation with communication by employing a split-phase technique [15]. This requires that the program is divided into smaller and independent sections that can execute/communicate concurrently. However, such transformation may require a significant amount of effort and make MPI applications difficult to implement.

D. Oversubscription Limitation in MPI

The granularity (G) of an SPMD application under MPI (i.e. how small are the parts of the workload distributed among processes) is proportional to both the problem size (N) and how many MPI processes (P) are used as per Eq. 3.

$$G = \frac{N}{P}$$  \hspace{1cm} (3)

Although the MPI specification provides very few restrictions on how an MPI library should be implemented, the most widely used implementations in supercomputers (e.g. MPICH [46], MVAPICH [47], Open MPI [55], Intel MPI [81]) instantiate each MPI process in the context of a separate OS process, as illustrated in Fig. 11.

Each OS process operates in its own local memory address space and executes autonomously from other processes. This simplifies the implementation of MPI libraries and works well for most applications. This approach, however, constrains the granularity of an MPI application to the number of available cores (c). An MPI program reaches its optimal performance, when the relation $P = c$ is satisfied since each process is matched to a single core, and all cores are used.

The problem of having the granularity of an MPI application fixed by the number of cores is that it does not allow performing efficient oversubscription – a necessary mechanism to hide the cost of communication. Experimental observation shows that using $P > c$ (oversubscription) in MPI libraries degrades application performance [30]. The are many causes for this:

(i) MPI processes will destructively compete for a core, constantly preempting each other from execution. This results in an increase in cache/TLB thrashing caused by repeated context switching.

(ii) Switching between MPI processes mapped to the same core carries the overhead of a kernel-level context switch.

(iii) MPI libraries achieve optimal performance when they employ busy waiting for the detection of new messages. This is the optimal strategy when only one MPI process per core is executed because it can instantly detect incoming messages. However, busy waiting produces destructive interference when cores are oversubscribed.

(iv) Barrier synchronization overhead increases with the amount of MPI processes.

Threaded implementations of MPI exist that solve the oversubscription limitation. These are presented in the next section.

V. ALTERNATIVE MPI-BASED MODELS

Several alternatives have been proposed to extend MPI’s functionality. In this section we present the two main alternatives models proposed in research.
A. Threaded MPI

The Threaded MPI model is motivated by the need to have MPI libraries that do not enforce the 1-to-1 MPI process to core mapping. These libraries enable efficient oversubscription by allowing the execution of multiple MPI processes in the context of a single OS process.

Fine-Grain MPI (FG-MPI) [36] instantiates MPI processes as user-level threads [35], functions that can be interrupted at any point of their execution while preserving their stack and processor state. User-level threads are managed and scheduled by a user-level runtime system. Other user-level thread-based MPI libraries, such as Threaded MPI (TMPI) [66] and AzequiaMPI [19] follow a similar approach as FG-MPI.

Preemption of user-level threads is enabled by calling a library-provided yield function that does not require the intervention of the kernel scheduler. When FG-MPI’s runtime system schedules a user-level thread for execution, it resumes from the point it had previously yielded.

By using user-level threads, FG-MPI enables the efficient execution of more than one MPI process per OS process. Fig. 12 shows how an example of how FG-MPI executes 2 MPI processes per core in the context of the same OS process.

The oversubscription factor ($V$) is a term used to define a number of MPI processes $P$ as an integer multiple of the number of cores $c$. For example, if $c = 16$, using $P = 16$ represents a conventional execution, while using $P = \{32, 48, 64\}$ represents $V = \{2, 3, 4\}$, respectively. The granularity of an SPMD application can then be reduced by a factor $V$.

$$G = N/(c \times V) \quad (4)$$

Realizing an oversubscribed execution is one ingredient to achieve communication/computation overlap. The idea is that processes waiting for a communication operation are inexpensively preempted from using the processor core and replaced with another that is ready to be executed [27], thereby keeping the core busy with useful computation.

Fig. 13a shows the execution timeline of an oversubscribed execution using $V=2$. We can see that the two MPI processes are allocated in the same core, each performing smaller bursts of effective computation than the ones shown in example in Fig. 10. The timeline also illustrates the overhead cost of context switching between processes 0 and 1 as diagonally striped blocks between task executions.

One risk associated with oversubscription is that the cost of context switching can overcome the benefits obtained by overlap. Fig. 13b shows the timeline of using $V=4$, where the increased oversubscription enables the processor core to be busy at all times. However, it also shows that the amount of work performed by each burst of computation is reduced, while the context switching overhead remains constant. This means that more of the core time is spent in switching overhead than actual computation. It is therefore important to perform a careful tuning of the oversubscription factor to find a sweet spot where the marginal gain in core usage equals the penalty of its associated overhead.

Another common problem with oversubscription in user-level thread-based MPI libraries is that global and static variables become shared among different tasks executing in the same process. In conventional MPI libraries, each process executes in a separate process-private memory space and has exclusive access of its global and static variables. However, when AMPI or FG-MPI populate a process with more than one task, global variables share the same virtual memory space, thus causing an incorrect behavior due to unintended data sharing.

A workaround for this problem is to perform a manual thread-wise privatization of global and static variables. The idea is to move all global variables into a structure or object that is instantiated at the beginning of the program and is passed as argument between subroutines. Some authors of AMPI have explored automated solutions [75]. However, these solutions are architecture-dependent and may not work in all cases.

B. Hybrid Model

The Hybrid (also called MPI+X) model has been proposed as a way to address the data duplication problem in MPI. This model involves a two-layer approach, where MPI is used to manage the process distribution across the network, and another interface is used to enable a shared memory execution inside each node or NUMA domain. By using shared memory, the impact of data motion is reduced since it eliminates redundant message copying and data duplication among MPI processes within the same physical address space.

The $X$ term in MPI+X can refer to any threading library or language extension. Threading libraries, such as OpenMP [56] or POSIX threads, are based on the execution of kernel-level threads. Contrary to user-level threads (that require library enabled yield/resume mechanisms), kernel-level threads are managed directly by the OS scheduler.

In the MPI+OpenMP [63] approach, MPI is used to create one process per node or NUMA domain, while
OpenMP ensures that all its threads are created within the same physical and virtual address spaces, thus providing a shared memory environment for programmers. The advantage of using kernel-level threads is that they can execute in parallel—scheduled across different cores—while sharing the same address space, as illustrated in Fig. [14]. This is not possible with user-level threads alone, since threads mapped to the same address space cannot execute simultaneously.

The locality of data in the MPI+OpenMP is given implicitly—that is, with an explicit allocation of shared variables in the code. Instead, all OpenMP threads instantiated within the same MPI process automatically share all of their global address space and pointers. Another option for the MPI+X approach, called MPI+MPI [28], works by handling communication across nodes through message passing or one-sided communication, while using MPI shared memory functions to manage shared memory access between processes in the same node.

MPI+MPI does not require a copy of a message into a receive buffer since data can be directly accessed by all processes in the same node/NUMA domain. This model enables shared memory without the need of threading (MPI processes execute as a single non-threaded OS process), by using inter-process shared memory mechanisms provided by the OS. However, since the address space is not shared (no threading), MPI requires programmers to explicitly define the locality of shared memory allocations. Memory allocations can only be shared between different MPI processes co-located in the same node but not across the network.

Despite the advantages of hybrid models, they have some drawbacks: (i) Programmers need to carefully define the interaction between shared and remote communication interfaces. (ii) As in any shared-memory program, synchronization mechanisms need to be implemented to prevent data races. (iii) Oversubscription is not efficient. In the case of MPI+OpenMP, switching between kernel-level threads involves higher overheads than switching between user-level threads. This penalizes performance in oversubscribed executions. (iv) MPI libraries are not thread safe. This means that kernel level threads may need to serialize their calls to MPI by using mechanisms for mutual exclusion.

C. Overview of MPI-Based Models

Despite the success of MPI in the scientific computing community, it has shown some difficulties in addressing some of the challenges of exascale. We have identified three main limitations:

(i) MPI is subjected to the data duplication problem. Data is duplicated because a message needs to be stored in the sender’s buffer, which is later deep copied into the receiver’s buffer. Furthermore, when a send operation is posted before a receive operation, even a third copy (intermediate buffer) is required, exacerbating the problem.

We have seen that Hybrid MPI models (MPI+X) are able to take advantage of shared memory. However, the main drawback of MPI+X models is that, by definition, they require two different interfaces for communication.

Programming an MPI+X application requires a careful insertion of synchronization mechanisms and a correct interaction between MPI and shared memory code. Developing such programs can require extensive efforts by the programmer and could be prone to bugs. (ii) The MPI specification does not prescribe any means for oversubscription. For flat (i.e., non-threaded) MPI libraries, this represents an important limitation at the exascale since oversubscription can hide some of the cost of communication. Threaded MPI libraries pose as an efficient solution to overcome the overhead of oversubscription in MPI. However, the limitation of Threaded MPI libraries is that they do not provide any means for solving the data duplication problem. In fact, in oversubscribed executions, data is further dispersed among MPI processes that cannot share memory, hindering data locality.

From our observations of the MPI model and its variants we can conclude that alternative parallel programming models targeted towards exascale computing should provide the following features: (a) Support for efficient oversubscription and shared memory, (b) A single interface for communication between processes inside and across nodes indistinctly.

In the next sections we introduce alternative models that have been proposed to overcome the limitations of MPI models, providing additional traits required by exascale computing. We start with the PGAS model, that provides a solution to the data duplication problem for applications with fine-grained communication.

VI. PARTITIONED GLOBAL ADDRESS SPACE MODEL

The Partitioned Global Address Space (PGAS) model [16] provides a framework for global memory that is meant to execute in multiple disjoint physical memory spaces. In a PGAS program, a distinction is made between variables private to a task, and those accessible by all tasks. We use the term task as a generic way to refer to the executing units by which a parallel application is divided. The task concept enforces no assumptions on the isolation of memory address spaces or the autonomy of execution.

Tasks can be implemented as a combination of user and kernel-level threads and managed by a runtime system to support oversubscription, shared memory, and/or data or execution dependencies.

A PGAS language hides the complexity of accessing memory globally across tasks. Global variables can be directly modified or read by any task via normal assignment operators and pointer accesses just like any other variable.

A global variable may be either physically located in the space of one task, or partitioned across the space of multiple tasks. The physical location of a partition (i.e., what node/NUMA domain contains it) is defined as the partition’s affinity. Affinity does not affect the correctness of a program, since partitions are equally accessible by all tasks. However, it plays an important
role in managing data locality. The optimal case is that in which shared partitions are located physically closest to the tasks accessing them.

Unified Parallel C (UPC) [12], is a PGAS extension to the C language. Similar tools have been developed for other languages as well, such as: Co-Array Fortran [53], and Titanium [74] for Java. In UPC, the allocation of shared spaces may be done statically, through non-initialized vector declarations (e.g. shared int array[SIZE];) or dynamically, through memory allocation functions (e.g. upc_global_alloc()).

Shared allocations in UPC can span tasks in linear or multi-dimensional arrays. Unless otherwise indicated by the user, all allocations in UPC are uniformly partitioned among tasks. For example, in a execution of 4 UPC tasks, a linear allocation of 100 bytes would result in 4 partitions of 25 bytes with 0, 1, 2, and 3 task affinities.

UPC implements implicit one-sided communication, where accesses to shared partitions are embedded into the language, instead of requiring explicit calls to get/put-like operations as in MPI. Accesses to remotely shared partitions are indistinguishable from private pointer accesses, except for the fact that the destination address refers to a remote location. This can be observed in Fig. [15].

Upon arriving at a read or assignment operation (=) on an element from a shared pointer, UPC performs the following operations: (1) dereferences the accessed element and obtains the offset within the shared allocation, (2) uses the offset to determine which partition it belongs to and its affinity, and (3) exchanges data with the task indicated by the affinity. Implicit one-sided operations in UPC solve the data duplication problem in MPI since UPC allows accessing remote partitions directly, without requiring a copy to/from a local buffer. Furthermore, empirical results have shown that the UPC approach requires less time per operation than the explicit one-sided communication primitives used in MPI – at the cost of adding synchronization mechanisms.

The problem with UPC’s approach is it can slow down the communication of large messages. UPC requires one independent operation per element when accessing a shared array. This makes the cost of communication linear in the number of bytes since each element access has to be resolved individually. The pathological case is shown in the code of Fig. [16] where every access to the shared pointer p represents a different operation. This problem is unavoidable since it is not possible to predict pointer-based accesses. As a consequence, implicit communication with UPC is only faster than explicit put/get functions in MPI for small sized messages, as shown in Fig. [17].

While implicit one-sided operations may be ideal for algorithms that rely on fine-grained communication (e.g. UPC is used in large-scale genome assembly algorithms [23]), it may become a significant overhead when communicating relatively large sets of data, as in Dense Linear (Matrix) Algebra algorithms. For this reason, UPC also provides functions for explicit one-sided communication (upc_memget, and upc_memput), similar to the ones provided by MPI.

As in MPI, the use of explicit operations in UPC requires a copy of the data from/to local buffers, therefore also incurring in the data duplication problem. Another issue with one-sided communication is that it requires synchronization mechanisms to prevent data races, just like in the Hybrid MPI model. UPC provides the upc_sync function for pair-wise synchronization between the sender and receiver tasks to verify that a message has been transferred. Semantically, upc_sync operates in a similar way as MPI_Wait.

The consequence of using explicit one-sided operations and pair-wise synchronization is that UPC programs can be very similar to MPI programs for algorithms that cannot benefit from fine-grained communication, as in the case of the Jacobi solver from Fig. [8]. The pseudocode of same solver programmed using UPC in Fig. [18] shows a very similar structure as the MPI version.

In the next section we analyze the APGAS model, an extension to the PGAS model that introduces the notion of task locality to enable the use of shared memory.
Asynchronous PGAS (APGAS) is an extension of the PGAS model that supports the explicit creation of tasks at runtime. Once created, the new task starts immediately and executes independently from its parent task. A parent task can then be set to wait for completion of its children tasks (and their descendants). Since the scheduling of tasks is based on the completion of other tasks, we say that these models have a task-dependency driven execution.

The main contribution in the APGAS model is that it allows programmers to define places, logical abstractions that provide the notion of task locality. Whereas affinity in UPC refers to where data is allocated, a place defines the physical resource where a task is executed. By specifying a place, the creation of tasks can be done locally (same place as parent task) or remotely (in a different place).

By defining their locality, tasks allocated to the same place are able to access variables from a shared address space without the need of an additional interface (as in Hybrid models). This locality-based sharing mechanism is automatically enabled by hardware, and does not suffer from the performance penalty for large messages we have seen from using software-based dereferencing as in UPC.

Our analysis of the PGAS model in the previous section showed that implicit one-sided communication can be inefficient when tasks are located in separate address spaces. However, by allowing the programmer to specify places, the APGAS model guarantees that data and tasks located in the same place will execute in the same physical address space without the need of software-based pointer access dereferencing, as in PGAS. To access variables among tasks in different places, programmers still need to specify the shared type modifier, just like in UPC. This shows that the PGAS model can be thought as a particular case of the APGAS model where all tasks execute in their own separate place.

**UPC++** is an APGAS communication library for C++ applications. UPC++ provides an interface that integrates the allocation and communication primitives in UPC with an interface for the creation of new tasks at runtime. UPC++ is based on ideas applied by APGAS-specific languages, such as X10.

In X10, tasks can be created by calling the async function. Async represents a Remote Method Invocation (RMI) – a request to execute a function in a remote location. By using async, the programmer specifies the place of the new task. The parent task can be set to wait for each of its child tasks (and their descendants) by using the finish function.

UPC++ adopts the async/finish semantics of X10, but also defines events. An event is a logical switch that is (partially) triggered upon the completion of a task. Events in UPC++ serve to create custom task dependency graphs by grouping the completion of one or more tasks as trigger conditions. By using the async_after() function, a new task is created but not executed until a certain event is satisfied.

Fig. 19 shows an example of the creation of a task dependency graph using events in UPC++. Tasks t1, t2, and t4 can start executing immediately. Event e1 is required for t3 to start executing, and will only be satisfied after tasks t1 and t2 are completed. The same applies to tasks t5 and t6 until event e2 is satisfied. Regarding locality, tasks (1,2), (3,4), and (5,6) are set to share the same place. We can see that two main aspects of this APGAS program have been explicitly defined: (i) the execution of tasks is given by the control-flow dependencies exposed by the defined events, and (ii) the locality of data is given by the defined places.

UPC++ has been proven to meet and even exceed the performance MPI at large scales, primarily in fine grained communication algorithms [77]. Fig. 20 shows the performance comparison between both models executing the LULESH solver using up to 32768 cores. LULESH is an example of an unstructured grid motif application that solves the Sedov Blast problem [29] in three spatial dimensions. LULESH can be used to benchmark performance in new hardware and programming tools. The experiment shows that UPC++ achieve a 10% speedup compared to MPI thanks to the use of shared memory among co-located tasks.

Charm++ is a programming framework based on APGAS principles. Charm++ extends the standard C++ syntax with structures based on the original Charm programming language. Charm++ provides an object-based extended API where tasks are defined as chares, C++ classes that can be defined by the programmer while inheriting a set of base chare methods. Chare classes have their own fields, constructors and entry methods. Tasks are created by instantiating objects derived from a chare class. The lifetime of a task is the same to that of a normal object, and communication is realized through calls to their entry methods.

Chares can be associated in arrays or groups that are allocated in the same processing element (PE).
processing element serves the same purpose as a place in UPC++. That is, all chares created in the same PE are guaranteed to execute in the same address space. When a chare is created, it executes its constructor method and remains inactive until one of its entry methods is called by another task. When tasks do not coexist in the same PE, Charm++ uses remote method invocations. Communication between tasks is then realized by two different mechanisms:

(i) When chares are co-located in the same PE, communication can be handled via shared memory. This is achieved by accessing the public fields (including arrays) of co-located tasks.

(ii) When chares do not share the same location, they perform explicit one-sided communication by sending pointers as arguments in RMIs. This prompts the runtime system to transfer data asynchronously from the pointer location. The called method is not executed until all data has been deep copied into the receiving task’s memory space. The called task then receives a local pointer to its copy of the data which can be accessed directly.

The disadvantage of shared memory communication in Charm++, when compared to UPC++, is that the programmer needs to handle communication between tasks using two different mechanisms whether or not they are co-located.

Fig. 21 shows an example of a program where a set of data set needs to be transferred to a child task. A new task is created in line 2 as a new object with initial arguments (args) and destination processing element (destPE). The value of destPE determines whether the child task will be allocated in the same space as its parent. The ckLocal method – inherited from the base chare class– returns a pointer to the object if the specified chare is co-located. In case it is co-located (line 7), the data array is shared among the two tasks and they can communicate through shared memory. In case the child task is remote, a NULL pointer is returned. This requires the transmission of the data pointer as an entry method argument of the ChareClass class (line 10), which incurs a deep copy of the data.

One of the distinctive features in Charm++ is that it supports the migration of chares among different PEs (and therefore, across the network) by deploying a packing and unpacking (PUP) framework. For a chare to be migratable, Charm++ only requires the programmer to define a put() virtual method that serializes/deserializes the contents of a task to/from a stream of bytes. Migratable chares is the main mechanism by which Charm++ enables load balancing and checkpoint/restart based fault tolerance.

In the next section we analyze a set of parallel programming models, called Dataflow models, that dynamically alter the execution order of an application based on the availability of data. We will see how this approach can be used to help hide the cost of network communication.

VIII. DATAFLOW MODELS

Dataflow processors were proposed to execute streams of instructions as directed acyclic graphs (DAG), where nodes represent instructions, and edges represent data dependencies between them. Instead of executing in the program’s order, instructions execute as soon as their required operands are satisfied, as long as no data hazards are detected. The earliest instances of such processors were the IBM System/360, implementing Tomasulo’s algorithm [68], and the CDC 6600, implementing the Scoreboard algorithm [67]. These ideas have paved the way for the out-of-order execution logic that have dramatically increased performance of processors.

The same principle has also been proposed for high-level parallel programming models [1]. A dataflow model defines the semantics of a program by declaring the elements that need to be calculated, and the dependencies (operands) required by each one of them.

In dataflow programming models, data dependencies are used to define the execution order of the application. The complexity of managing a data dependence-driven execution is, however, hidden from the programmer. For this reason, these models rely heavily in both translator/compiler and runtime system support. Compilers need to perform a static analysis of dependencies and embed out-of-order logic inside the code, while runtime systems are required to track the data the status of data dependencies (whether they are satisfied) and deciding which operation(s) can be executed next.

We identify four dataflow programming models that implement a range of different approaches: Concurrent Collections, Statement-level Dataflow, Region-level Dataflow, and Task-level Dataflow.

A. Concurrent Collections

The Concurrent Collections (CnC) model expresses the control-flow of a parallel program in terms of producer/consumer relations between data elements. A CnC program defines data collections where each element in a collection is assigned a tag or identifier. A programmer can define dependencies, where the value of a tag may require pre-calculating the value of other tags.

What makes CnC a particularly interesting programming model is the ability to compose a program through memoization – storing intermediate results for later reuse. By requesting the value of a tag, a CnC program calculates and stores the values of all the tags upon which it depends, and then performs a simple operation to calculate its value. This logic is applied recursively until the last dependencies are only external inputs (e.g. constants, command line arguments, files, user input). In this way, a CnC program obtains the value of a tag as series of simple data transformations starting from external inputs.

Fig. 22 represents a typical CnC code for calculating the $i^{th}$ number in the fibonacci sequence for any positive integer $i$. Left arrows indicate input dependencies. In this example, there are two inputs, $x$ and $y$, representing the previous two fibonacci numbers (in case of $i > 2$). For any $i$ required by the programmer, the CnC model will automatically resolve and memoize the range of previous fibonacci numbers. Right arrows indicate the result value,
and where it is stored. The result of each i is stored as tag fib: i.

CnC has proven to be an ideal model for developing linear algebra libraries. **DAGuE** \(^{10}\) is a programming language and runtime system based on the CnC model, that represents complex algebraic transformations, such as the Cholesky factorization (Dense Linear Algebra) as relations between matrix/vector operations.

The dependency graph of the Cholesky factorization generated by DAGuE is shown in Fig. 23. **DPOTRF**, **DTRSM**, **DSYRK** and **DGEMM**, represent simpler linear algebra operations upon which Cholesky is composed. The programmer only needs to define the operations between tags required to perform the factorization, while the DAGEu compiler/runtime system deals with the intricacies of fetching the dependencies for each element. As shown in the illustration, DAGEu automatically defines how data elements and operations are mapped across compute nodes, and how communication is automatically distinguished between local and remote.

**B. Statement-Level Dataflow**

The dataflow model can also be applied to procedural languages by having the statements of a program execute out-of-order, based on their dependencies.

**Swift/T** \(^{73}\) is a C-like language and compiler based on the statement-level dataflow model. Just like DAGEu, Swift/T requires no explicit definition of parallelism nor data locality. However Swift/T’s approach differs from DAGEu’s in that the former constructs the underlying DAG through a static analysis of program instructions, rather than relationship between data elements. During execution, its runtime system creates one task for each statement in the program and manages their data dependencies, while optimizing data and task locality.

Fig. 24 shows a simple example of a Swift/T program and the corresponding DAG that is generated by Swift/T primitive algebra operations upon which Cholesky is composed.

Fig. 22: A typical CnC fibonacci code. Adapted from \(^{71}\).

**Fig. 23:** Decomposition of the Cholesky factorization into simpler operations in DAGEu. Source: \(^{10}\).

in compilation time. The value of y will be calculated first (line 4) since it is the only one that has data no dependencies. Lines 3 and 5 are executed concurrently afterwards, since they only depend on the value of y. The value of z2 (line 6) is calculated once z1 and x are obtained. The last steps represent the calculation of the contents of the A/J array (line 7) based on the values of z1 and z2, and the parallel processing of all its elements (line 9).

One of the main advantages of both Swift/T and DAGEu is that they manage communication automatically and use shared memory when tasks are co-located in the same address space. However, the fact that they require the creation of large amounts of extremely fine-grained tasks and dependencies can potentially entail large overheads. Although their compilers and runtime systems are optimized to handle large amounts of elements, it is unclear how they deal with the following problems:

(i) The overhead required in managing fine-grained tasks (creation, allocation, dependency evaluation) may be on the same order of complexity as the very operations executed by each task.

(ii) As tasks execute smaller operations, they also produce smaller results. As a consequence, communication operations become extremely fine grained and susceptible to the cost of latency. Since latency will be the dominant cost of communication in exascale computers, fine grained communication may be punishing for performance.

(iii) The number of communication operations are correlated to the number of tasks, and therefore to the complexity of the algorithm.

**Fig. 24:** Example code of a Swift/T program and its corresponding DAG. Simplified for clarity.

**Fig. 25:** Pseudocode of the naive \(O(n^3)\) matrix multiplication algorithm kernel.

Nested loops represent pathological cases where performance drops quickly with increased problem sizes. Polynomial algorithms, such as the naive matrix multiplication shown in Fig. 25 require the creation of \(O(n^3)\) tasks (where \(n\) is the number of elements in a side of a square matrix). Published tests show that such algorithms fail to scale beyond \(O(10^6)\) cores, which is way below the order of magnitude involved in exascale computers \(^{73}\). This is illustrated in Fig. 26, where tasks completion per second drops after >1000 cores due to a nested loop in a Swift/T program.
The dataflow model can also be applied to a procedural language where entire regions of code are scheduled, instead of individual statements. Regions are defined as contiguous sections of the code that execute in-order and non-preemptively. Programs can have their code regions rearranged statically (on compilation time) or dynamically (on runtime) to execute out-of-order based on their data dependencies.

Bamboo is a source-to-source translator that reinterprets C/C++ MPI applications to execute as a dataflow program to realize communication/computation overlap. Bamboo performs a static analysis of Bamboo-specific annotations and MPI calls in the code and generates a data dependence graph. This graph is used to perform transformations in the source code that enable a data dependence-driven execution by generating code that is compatible with the Tarragon runtime system [14]. Tarragon provides support for thread-based oversubscription of Bamboo-translated code. For this reason, Bamboo also suffers from the global variable problem observed in the Threaded MPI model.

Bamboo extends the C++ syntax with \#pragma directives [5] used to define three types of regions in the code: (i) Overlap regions indicate what parts of the code will execute in-order and non-preemptively. Programs can have their code regions rearranged statically (on compilation time) or dynamically (on runtime) to execute out-of-order based on their data dependencies. (ii) Send/receive regions are used to enclose MPI send/receive operations, respectively. (iii) Compute regions are used to enclose computation that depends on the receive region and produces data for the send region.

Fig. 27 shows an example of Bamboo’s annotation syntax for the iterative solver presented in Fig. 8:}

```c
#pragma bamboo overlap
for (int step = 0; i < iterations; step++)
{
#pragma bamboo send
MPI_Isend(&Uprev[BoundaryCells] -> [up,down,left,right]);
#pragma bamboo receive
MPI_irecv(&U[GhostCells] <- [up,down,left,right]);
#pragma bamboo compute
{,
  for (int i = 0; i < N; i++)
    for (int j = 0; j < N; j++)
      U[i][j] = U[i-1][j] + U[i+1][j] +
               U[i][j-1] + U[i][j+1] - 4-U[i][j];
    swap(&U, &Uprev);
}
}
```

Blocking operations (e.g. MPI_Waitall) are postponed by Bamboo to prevent the entire process from being preempted while there are still regions ready to execute (ie. their particular data dependencies are satisfied).

**MPI/SMPSs** [42] is also a \#pragma directive-based interface for MPI C/C++ applications. The SMPSs model [58] was originally proposed as a framework to describe dependencies between tasks in OpenMP. MPI/SMPSs is an adaptation of the SMPSs interface to describe data dependencies across MPI processes.

Unlike Bamboo, where dependencies between regions are obtained from the static analysis of the code, SMPS requires on the programmer to explicitly name and define dependencies between regions. Although MPI/SMPSs gives more flexibility in defining more complex dependency graphs than the (receive→compute→send) pattern used by Bamboo, specifying dependencies manually could require an additional effort from programmers.

Both Bamboo and MPI/SMPSs automatically convert MPI applications into semantically-equivalent programs that execute based on data dependencies. Data dependence-driven execution has a significant impact in hiding the costs of network latency. We have seen in Fig. 13a that, although oversubscription enables overlapping computation and communication by switching tasks when one of them is blocked for the arrival of messages, each individual task still performs communication and computation in separate stages.

As a result of dataflow execution, a task can be scheduled as soon as at least one of its regions has satisfied its dependencies. The effect of this is a finer-grained execution of each task, realizing overlap without the additional context-switching overhead we had observed in oversubscription-only executions (Fig. 10).

Fig. 28 shows how a single task can overlap communication and computation in non-oversubscribed executions. Regions are represented as fine-dotted division lines within a task. When compared with Fig. 10, we see that this mechanism can potentially enable a better core usage, as compute regions still execute while communication operations run in the background.

It is important to note that overlap by oversubscription and data dependence-driven execution are not mutually exclusive. The effect of using both mechanisms can improve processor utilization than any of them separately. Fig. 28 shows how their combination represents can potentially improve over the oversubscription-only approach shown in Fig. 13a.

It has been shown in a variety of applications [51] [50] that the best performance is obtained by a combination of oversubscription and data dependence-driven execution.
they can essentially be reduced into two levels: several types of privilege levels, but for our analysis, that share the same access code). Regions are structures for grouping data objects (which). Region-level Dataflow model, that refers to sections of a circuit simulation algorithm in Legion. Each circle represents a certain circuit node (with voltage/current/charge values) and the connections represent wires connecting the nodes. To allow parallel execution, the circuit is divided into three partitions of similar number of nodes. We can then define 6 regions: 3 exclusive regions (cyan) that contain nodes modified by a single task, and 3 simultaneous regions (red), representing the boundaries between partitions that are modified by two or more tasks.

Once data regions are defined, the programmer needs to instantiate the tasks that will process them. This is done via the spawn function. This function has a similar semantics as the async function in UPC++ and X10, requiring a function to execute to be passed as argument. However, instead of requiring the task locality (as with places in UPC++) as an argument, Legion infers task locality dynamically from the data region(s) associated to new task. The scheduling of tasks is given by how dependencies between data regions and their privilege levels are structured.

The semantics of a Legion program is given only by defining data regions and instantiating tasks using the spawn function. However, to realize full performance, a programmer needs also to define mapping of regions to physical resources. An optimal mapping will assign neighboring data regions into the same resources to maximize the use of shared memory.

Legion gives a programmer an interface to define almost all the aspects of the execution of a program: data dependencies, task locality, and task-to-physical resource mapping. However, Legion requires that many of these aspects be explicitly defined by a programmer whereas tools like Bamboo or MPI/SMPss implement simpler and less complicated ways to support a dataflow execution.

### IX. DISCUSSION

#### A. Overview of Alternative Models

Table 1 shows how the programming tools we surveyed compare regarding their design paradigms. All of them provide one or more mechanisms to address the challenges brought by exascale computing.

Regarding the cost of *internal communication*, we have seen that defining the locality of data/tasks can help deal with the data duplication problem. Locality gives runtime systems the necessary information for mapping groups of data/tasks into the same physical resources, enabling the use of shared memory and removing the need for buffering, preventing unnecessary data motion within a compute node.

The tools surveyed also simplify the way communication is expressed, reducing the effort of programmers and eliminating the need of multiple interfaces. The PGAS model paved the way by integrating local and remote

![Fig. 29: Speed-Up of a 7-point stencil 3D Jacobi solver using Bamboo subject to different oversubscription factors. Source: [50].](image)

![Fig. 30: Nodes and wires in a circuit simulation program, split into three partitions. Source: [9].](image)
communication with an implicit one-sided communication interface. However, this type of communication proved to be less than efficient for messages larger than a few bytes. In APGAS models this is partially solved by providing the concept of task locality, guaranteeing that data and tasks running in the same places share the same address space. As a result, APGAS languages are able to integrate shared memory and distributed communication, just like in the Hybrid (MPI+X) model, but using just a single programming interface.

Some dataflow programming tools, such as DAGuE and Swift/T do not provide any interface for specifying locality, as this information does not play a role in defining the semantics of the program. Instead, these tools infer the best logical-to-physical resource mapping automatically during compilation or runtime. On the other hand, Legion requires programmers to use an interface for mapping data regions to physical resources manually. Although a manual mapping may be more efficient, mapping logical and physical resources by hand can be extremely complex and may not portable to other system configurations.

The contrast between manual/automatic mapping serves to illustrate that there is a trade-off between simplicity and efficiency regarding data locality. A possible alternative to get the best of both worlds could be to use compiler directives, as in Bamboo and the SMPSS model. Compiler directives may be a good alternative because they fit the same criteria that are required by locality specifications: (i) They are optional. Their inclusion or omission does not affect the semantics of a program. (ii) They are portable. They can be interpreted in different ways depending on the system. (iii) They can provide a simpler interface than that required by a manual mapping (Legion).

Regarding the cost of network communication, most tools support oversubscription by combining both user-level and kernel-level threads. We have shown how this mechanism is the basis for enabling communication and computation overlap. SPMD-based libraries, such as FG-MPI, Bamboo and UPC offer a simple way to achieve oversubscription by defining an oversubscription factor. That is, creating a fixed V number of tasks per core. It is less intuitive, however, how to achieve oversubscription in asynchronous models (e.g. APGAS, X10, DAGuE, Swift/T, and Legion), where tasks are created dynamically. For these models, either the programmer or the runtime system needs to make sure there is enough tasks at all times to maximize core usage and perform communication operations simultaneously.

Dataflow models also provide data dependency driven execution. This mechanism increases the overlap potential of an oversubscribed execution. In some cases, the dependency graph is automatically inferred from the code, and in other cases it needs to be specified manually.

While the task dependency-driven execution approach used in PGAS/APGAS tools could potentially enable communication/computation overlap, it is not a straightforward way of representing data dependencies. With task dependencies, the communication operations should be described by communication-only tasks upon which other compute-only tasks depend. This description of data dependencies is not as straightforward as in dataflow models, where communication operations are clearly identified as such.

### B. Towards an Ideal Model

We consider that each model provides a unique set of weaknesses and strengths. It is certainly not possible to build an ideal programming model since memory access, communication, and computation patterns vary widely among different applications. However, in this section we seek to define the design aspects we believe would be close to such ideal model:

(i) Follow the Single Program Multiple Data model. We consider that the SPMD is still a relevant way to decompose the problem domain of most scientific computing motifs. This involves creating a fixed number of tasks for the distribution of the workload, as opposed to asynchronous models where tasks are created dynamically on runtime. The simplicity of this approach would also make it easier for MPI programmers to transition to such ideal model.

(ii) Oversubscription = Locality. An ideal model should directly relate the oversubscription factor and the locality of tasks. This approach would give users the possibility of defining how many tasks are created per node as well as provide support for shared memory among tasks in the same node automatically.

(iii) Provide a single (implicit) communication protocol as in the PGAS/APGAS models. This would require no semantic difference between communicating among tasks in the same node or across nodes. The only difference between internal and network communication should be their performance.

To avoid the problem with large size messages, an ideal model should have heuristics to predict consecutive group pointer accesses as a single communication operation. Although this may not be generally possible, it would...
reduce the per-element access overhead we observed in PGAS models.

Additionally, an ideal model should provide a simple annotation-based syntax for programmers to distinguish between private and shared variables. For shared variables, this model should provide a way to indicate how data allocation will be distributed among tasks, as with affinity in UPC.

(iv) Provide an easy means for defining data dependencies. The experience of the Bamboo translator indicates that it is possible to have a simple annotation-based interface for defining data dependencies between regions of code to maximize communication/computation overlap.

X. CONCLUSIONS

In this study we have presented some of the challenges posed by exascale computing and the models that have been proposed to address them.

For many programmers, MPI is still a viable option since it has been able to catch up with some of the latest ideas and developments, especially in the use of shared memory. However, it is unclear whether the ad hoc addition of features to the MPI specification will be an option in the long run. As it stands now, it can be cumbersome even for expert programmers to fully understand and utilize the whole range of tools provided by MPI.

Alternative models have shown that it is possible to provide additional mechanisms such as oversubscription, task/data locality, shared memory, and data dependence-driven execution to alleviate the costs of communication while using simple programming interfaces.

Future research needs to be focused on integrating mechanisms to address the challenges we have described but also others that were out the scope of this study. Many-core devices continue to be a challenge due to their own interfaces and communication protocols, making it hard to conciliate with existing programming models. Other aspects we have not covered such as fault tolerance, load balancing, and power consumption will also become important challenges to address as we approach exascale computing.

ACKNOWLEDGEMENTS

I would like to thank my Ph.D advisor, Prof. Scott Baden, for his insightful comments and constructive criticism throughout the process of writing this study and Dr. Tan Nguyen (Lawrence Berkeley National Laboratory) for providing additional feedback.

REFERENCES


