Architectural constraints to attain 1 Exaflop/s for three scientific application classes

Abhinav Bhatele, Pritish Jetley, Hormozd Gahvari, Lukasz Wesolowski, William D. Gropp, Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
E-mail: {bhatele, pjetley2, gahvari, wesolwsk, wgropp, kale}@illinois.edu

***Abstract*— The first Teraflop/s computer, the ASCI Red, became operational in 1997, and it took more than 11 years for a Petaflop/s performance machine, the IBM Roadrunner, to appear on the Top500 list. Efforts have begun to study the hardware and software challenges for building an exascale machine. It is important to understand and meet these challenges in order to attain Exaflop/s performance. This paper presents a feasibility study of three important application classes to formulate the constraints that these classes will impose on the machine architecture for achieving a sustained performance of 1 Exaflop/s.**

The application classes being considered in this paper are – classical molecular dynamics, cosmological simulations and unstructured grid computations (finite element solvers). We analyze the problem sizes required for representative algorithms in each class to achieve 1 Exaflop/s and the hardware requirements in terms of the network and memory. Based on the analysis for achieving an Exaflop/s, we also discuss the performance of these algorithms for much smaller problem sizes.

***Keywords*-application scalability; exascale; performance analysis; molecular dynamics; cosmology; finite element methods**

I. Introduction

Parallel supercomputers have kept up the pace of peak performance improvement: The first peak Petaflop/s machine, Roadrunner, appeared on the Top500 [1] list in June 2008, and multiple systems beyond that performance level have been planned for near future. The community has set a goal of building an Exaflop/s machine by 2018. There are several hardware challenges to be overcome before we break the Exaflop/s barrier - power/energy costs, memory costs, communication and others. The continuous frequency increase that we enjoyed in the past has come to an end. In part due to this, it has been clear that a co-design approach, where machines are designed in conjunction with exascale applications will be needed to achieve the goal of an Exaflop/s by 2018 [2].

Assuming that we can overcome the hardware challenges and an Exaflop/s machine is built, scientists will have to modify/develop algorithms and applications that scale to exascale. To this end, we analyze three prevalent application classes that currently occupy a significant portion of compute cycles on various supercomputers (supported by INCITE and PRAC allocation awards) – classical molecular dynamics, cosmological simulations and unstructured mesh computations (finite element solvers).

Goals arising from the science involved suggest that the scientific communities using these applications will need exascale performance, so it is important to project the performance of these applications on an exascale machine.

These three application classes encompass some of the most common parallel data structures, including structured grids, unstructured grids and particles (N-body). Between the three chosen classes, a range of computational and communication patterns are covered which should provide insight into the scaling challenges we will face on the road to exascale. We consider weak scaling of these applications to the full size of the machine. At exascale, scientifically important objectives may also involve studying problems smaller than what weak-scaling suggests (i.e. 1000 times larger problems compared with those on petascale). Therefore, we also study performance issues for smaller problem instances.

The first class of applications chosen for the study are molecular dynamics (MD) applications that focus on the simulation of biomolecular systems. Several highly scaling MD codes are used today on supercomputers – NAMD [3], AMBER [4], Gromacs [5], Desmond [6] and Blue Matter [7]. MD simulations involve calculation of forces on a system of N atoms. We discuss different parallelization strategies for the force calculation and select the one with the lowest computation to communication ratio. For the purposes of this study, we consider only short-range calculations (also referred to as Lennard-Jones dynamics).

The second class of applications are cosmological simulations. These applications constitute another important category with a unique communication pattern. Gravitational solvers for the N-body problem use one of many different methods: direct sum, tree-methods, particle-mesh methods and hybrid codes. Some examples of cosmology codes are PkdGRAV [8], ChaNGa [9], Enzo [10] and FLASH [11]. We consider tree methods for solving the N-body problem for our analysis and set aside hydrodynamics for a later study.

Unstructured grid problems, the third class under consideration, arise frequently in science and engineering. Many problem domains have complex shapes that do not lend themselves well to a simple finite difference discretization. Setting the problem as an unstructured grid, which involves breaking the domain into triangles (in 2D) and tetrahedra (in 3D), allows for complicated domains to be discretized in a straightforward

manner. To solve these problems, finite element method (FEM) solvers are most commonly employed. A detailed treatment of the finite element method can be found in [12], but the basic principle is to represent the solution to the problem as a sum of basis functions over elements in a mesh, and this matches up well with the setup of an unstructured grid problem. There are many ways to apply finite element solvers, but they generally center around assembling and solving a sparse linear system, which can be done once or repeated several times depending on the problem being solved. This is the approach we consider in this study.

We first introduce the performance model used in the paper. Each application class is then analyzed for its computation and communication requirements for weak scaling. The analysis helps derive constraints on the hardware, and then the analysis is repeated for smaller problem instances. We also analyze peak memory requirements of each application at scale. A recent paper by Gahvari et al. [13] does a similar analysis studying the feasibility of 3D FFT and multigrid at exascale.

II. Machine Parameters and Assumptions

This section describes the methodology we use to model the computation and communication behavior of parallel algorithms. The amount of computation for each problem is described in terms of the number of calculations, which is a function of the problem size, N, and the number of processing cores, P_c. For each calculation, we estimate the number of floating point operations, n and multiply that by the time for computing a flop, t_c. Since the sequential performance often does not achieve the peak flop/s rating, we multiply the expression by an efficiency factor $1/\eta$, This gives the equation for computation time as,

$$T_{comp} = \frac{1}{\eta} \times f(N, P_c) \times n \times t_c \tag{II.1}$$

Communication on parallel machines can be described in terms of three parameters:

- Start-up time (t_s): This is the time required for handling of a message at the sender and receiver. It is often referred to as overhead and is incurred once per message.
- Per-hop time (t_h): This is the time spent at every switch/router on the network that the message goes through. It is multiplied by the number of hops or links, l, traversed by the message.
- Per-word time (t_w): If the bandwidth of each link on the network is B_w GB/s and the size of a word is 4 bytes, each word spends $t_w = 4/B_w$ time to traverse the link. This is referred to as the per-word transfer time.

Using these three parameters, we can express the time for sending a message on the network as,

$$t_s + l \times t_h + m \times t_w$$

where m is the size of the message in words. We assume that the exascale machine will use wormhole routing to send flits on the network (as is the case for most supercomputers today). This suggests that, in absence of contention and for messages of sufficiently large size, the second term in the equation above will be significantly smaller than the third term. Also, it should be possible to limit the number of links traversed to a few hops using an intelligent topology aware mapping [14]. So, for the analysis in this paper we ignore the second term in the equation. If an application sends $M = g(N, P_c)$ messages and each message is of size $h(N, P_c)$, the time for communication will be given by:

$$T_{comm} = M \times (t_s + h(N, P_c) \times t_w) \tag{II.2}$$

We want to make as few assumptions as possible about the architectural details of an Exaflop/s machine. However, we must fix a few parameters for our analysis. Most large supercomputers today have multiple cores per node and the number of cores on each node is expected to rise. Let us assume that our hypothetical machine will have 1 GHz processing cores and each node will contain 1024 such cores. The peak performance of the machine will be 10.74 Exaflop/s, requiring $P_c = 2^{30}$ 10 Gflop/s processing elements (number of nodes, $P_n = 2^{20}$). The compute time per floating point operation, $t_c = 0.1$ ns (assuming 10 flops per cycle).

Using the parameters and assumptions described above, we estimate the range of values for network latency and bandwidth and memory requirements for performing exascale simulations for the three application classes.

III. Molecular Dynamics

Molecular dynamics (MD) codes constitute an important class of parallel applications. We will focus on MD codes that are used for simulating the life of biomolecules to understand their structure and facilitate drug design. Over the years, a plethora of parallel codes have been written to simulate MD – NAMD [3], AMBER [4], Gromacs [15], Desmond [6] and Blue Matter [7] to name a few.

MD is a difficult problem to parallelize because of the small number of atoms and extremely small time scales (typically 1 to 2 femtoseconds) involved. Over the years, various parallelization techniques have been developed for scaling MD. Plimpton gives a detailed overview of different approaches to parallelizing MD in [16]. The traditional methods of parallelizing classical MD computations are atom decomposition and force decomposition. In atom decomposition, the atoms involved in the simulation are distributed among the processors, in no particular order and each processor is responsible for calculating forces for its atoms. In force decomposition, the force matrix for the atoms is distributed among the processors. If the number of atoms in the simulation is N and the number of processing cores is P_c, the communication to computation ratios for the two methods are:

$$C/C\ ratio_{atom} = \frac{N}{N/P_c} = P_c$$

$$C/C\ ratio_{force} = \frac{N/\sqrt{P_c}}{N/P_c} = \sqrt{P_c}$$

Both of these approaches are non-isoefficient and hence not used in modern, highly scaling MD codes. So, we focus on

the spatial decomposition method in this paper. In this method, the three-dimensional (3D) simulation box is spatially divided among the processors. Let us assume that the simulation box has dimensions $B_x \times B_y \times B_z$; then, each processor holds a cell of dimensions $B_x/\sqrt[3]{P_c} \times B_y/\sqrt[3]{P_c} \times B_z/\sqrt[3]{P_c}$ and is responsible for calculating forces for the atoms within its cell. For most MD simulations, we can safely assume that the density of atoms in any cell is roughly the same, which leads to approximately the same number of atoms per processor. For the spatial decomposition method, the communication to computation ratio is given by:

$$C/C\ ratio_{spatial} = \frac{N/P_c}{N/P_c} = 1$$

Modern methods of parallelizing MD, which are a hybrid between spatial and force decomposition [17] (also known by other names such as the midpoint method and the neutral territory method [18]) improve the communication to computation ratio as the cell size decreases, compared to the spatial decomposition method. However, their asymptotic complexities are similar to the spatial decomposition method and hence, we will not consider them separately.

To aid our complexity analysis, let us understand the parallel set-up of a "short-range" molecular dynamics simulation. The simulation time is broken down into a large number of small time steps (typically 1 fs each). At each time step, each processor calculates forces on the atoms that reside on it due to all other atoms within a certain distance, $r_c + margin$, where r_c is the cutoff radius and $margin$ accounts for atom movements between migration steps. To calculate the forces, each processor communicates with its neighbors in the 3D space to obtain the current positions of atoms within this radius. New positions and velocities are then calculated and updated, based on the force calculations within a time step. Based on the new positions, some atoms may move into a cell assigned to a different processor and they have to be migrated. Typically, migrations are not done every time step and to account for this, the size of each cell is chosen to be $r_c + margin$. Algorithm 1 shows the pseudocode for one time step of an MD simulation.

Algorithm 1 Computation in one time step of MD

```
Receive atoms from neighboring processors
for i = 1 to N_p do
    for j = 1 to N_i do
        if atoms are within cutoff radius, r_c then
            Compute forces on pairs of atoms
        end if
    end for
end for
Update atom positions and velocities
```

A. Weak Scaling

We begin with analyzing the weak scaling behavior of the spatial decomposition method. For this analysis, we need a lower bound on the number of atoms assigned to each core for maintaining good efficiency. Both Blue Matter [7] and NAMD [3] have demonstrated that for ratios of atoms to cores greater than 100, the non-bonded force calculation is the dominant contribution to the step time. And in this regime, the performance follows a "universal curve" irrespective of the molecular system, only depending on the number of atoms per core. This is achievable for short-range MD computations because the number of floating point operations per core is a linear function of the number of atoms. Assuming that our hypothetical system will have 100 atoms per core for achieving 10% of the peak which will be ≈ 1 Exaflop/s, total size of the molecular system would be $2^{30} \times 100 \approx 107$ billion atoms. Total number of floating point operations for a simulation system with N atoms is $33547 \times N$ (empirically obtained value for NAMD for a 12 Å cutoff). Considering that we want the flop/s to be greater than or equal to 1 Exaflop/s, dividing the total number of flops by the time for one time step gives:

$$\frac{flops}{T} > 10^{18} \qquad \text{(III.1)}$$
$$\frac{33547 \times N}{10^{18}} > T$$

Putting the value of $N = 2^{30} \times 100$,

$$T < 3.6 \times 10^{-3} \qquad \text{(III.2)}$$

This says that, to achieve 1 Exaflop/s performance for a 107 billion atom system running on 2^{30} cores, the time per step should be smaller than 3.6 ms. The time per step for each application class is the performance target to attain 1 Exaflop/s performance. Since all applications considered in the paper are iterative, the equations derived for T, T_{comm} and T_{comp} are for one time step.

Let us now estimate the amount of communication per node for this molecular system of 107 billion atoms. For a standard MD simulation, the size of each cell in the simulation box is 16 Å in each dimension (for a cutoff $r_c = 12$ Å and a $margin = 4$ Å) and the number of atoms in each cell is 400 (see Figure 1, extreme left). Since for the 100 billion atom system, we will have only 100 atoms on each core, this necessitates splitting each cell into half in two of the three dimensions (see Figure 1, center). In this mode, each cell communicates with approximately $5 \times 5 \times 3 = 75$ other cells to obtain the atoms necessary for calculating forces on its atoms. However, having multiple (1024) cores on each node implies that most of these messages are not sent on the network. If we assign a three-dimensional space containing $8 \times 8 \times 16 = 1024$ cells to a node, inter-node messages will be required only for cells on the surface. The number of messages will be $12 \times 10 \times 20 - 8 \times 8 \times 16 = 1376$ (two "ghost" layers of cells each in two dimensions and one layer of cells in the third dimension).

Based on the above derivations for communication and computation in an MD code for weak scaling, we can now use equations (II.1) and (II.2) to obtain the time for one time step of MD. In the case where there is no overlap of communication

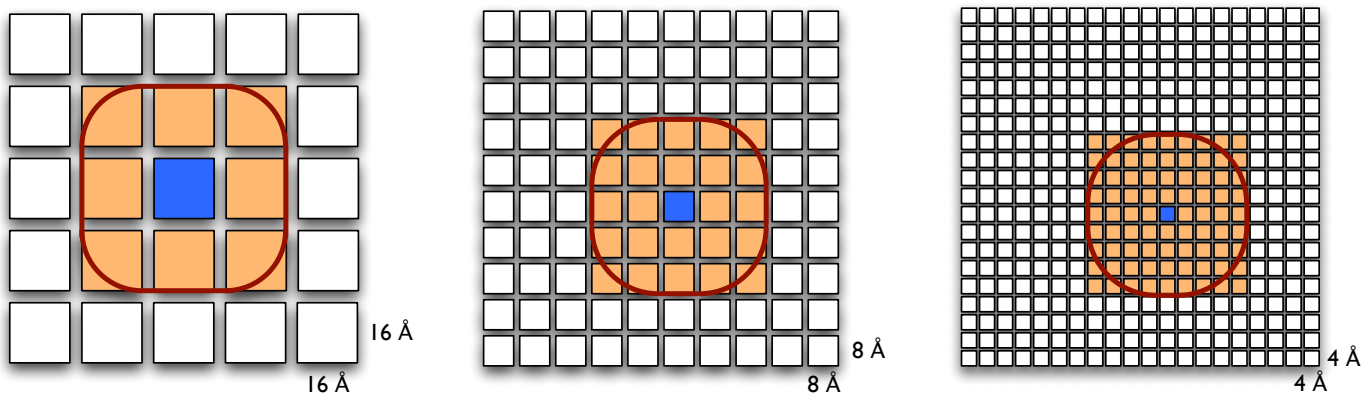

Fig. 1. A biomolecular simulation box (only two dimensions shown) split into cells of size $16 \times 16 \times 16$ Å (extreme left). Each processor holds one such cell containing approximately 400 atoms. When there are fewer atoms per processor (say 50), the three dimensions are further split to give cells of size $8 \times 8 \times 8$ Å (center). When there are around 6 atoms per processor, each dimension is reduced to one-fourth the original size (extreme right).

and computation, at every time step, each node sends positions and velocities of the atoms to its communicating neighbors and once it has received its incoming messages, calculates forces on its atoms. The expression for the time per step of an MD computation is:

$$T = \frac{1}{\eta} \times \frac{N}{P_c} \times 33547 \times t_c + 1376 \times \left(t_s + \frac{N}{P_c} 4t_w \right) \quad \text{(III.3)}$$

Substituting the expression for T from equation (III.3) in equation (III.2),

$$\frac{1}{\eta} \times \frac{N}{P_c} \times 33547 \times t_c + 1376 \times \left(t_s + \frac{N}{P_c} 4t_w \right) < 3.6 \times 10^{-3}$$

For the weak scaling analysis, putting in the values of ratio of atoms to processors, $N/P_c = 100$ and $t_c = 10^{-10}$ seconds,

$$\frac{1}{\eta} \times 33547 \times 10^{-8} + 1376 \times (t_s + 400t_w) < 3.6 \times 10^{-3}$$

$$1376 \times (t_s + 400t_w) < 3.6 \times 10^{-3} - \frac{1}{\eta} \times 3.35 \times 10^{-4}$$

$$t_s + 400t_w < 2.62 \times 10^{-6} - \frac{1}{\eta} \times 2.44 \times 10^{-7}$$

Figure 2 plots the values of t_s and t_w based on the equation above for different values of η. For the case of perfect efficiency, MD simulations do not put a considerable requirement on the per-processor communication bandwidth. However, it does require that the network latencies be small. If we look at the case of $\eta = 0.125$, the application would require a latency of below a microsecond and a per-processor communication bandwidth of 2 GB/s. It is also important to mention that our analysis assumes serialization of messages put on the network by a node arising from all of its 1024 cores. We expect that for future machines, multiple cores on a node will be able to inject messages on the network in parallel.

B. Memory requirements

MD codes have a relatively small memory footprint since the number of atoms on each core is small (between 5 to 400). However at the start of each time step, when atoms

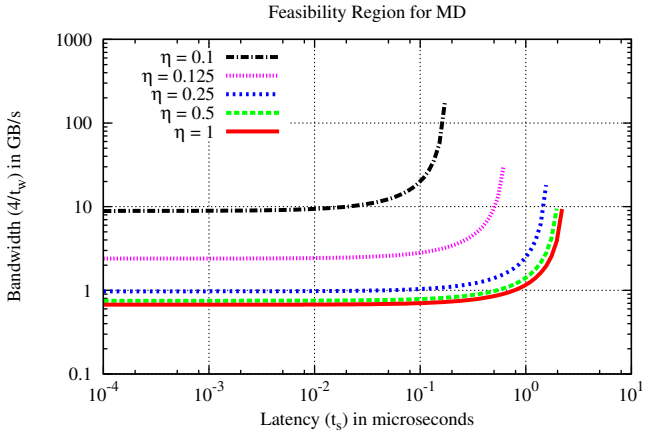

Fig. 2. Latency and bandwidth requirements for MD (weak scaling)

are received by the processing cores, the amount of memory needed increases. This is proportional to the total number of messages received by each core (75 for the case above). The size of each message is equal to N/P_c multiplied by the memory requirements for the atom data structure. The information about each atom sent in the message is the charge on the atom and its position. Hence the increase in memory consumption at the beginning of each time is equal to $75 \times (N/P_c) \times 32$ bytes = 0.23 MB. However, even this transient memory usage in MD simulations is not significant.

C. Smaller problem sizes

An important observation is that building a 107 billion atom molecular system and doing useful science with it, will be a challenge for biophysicists. Simulating such a large system to observe anything meaningful will require long simulations (milliseconds to seconds). The largest classical MD simulations done so far involve up to 3 million atoms, a five orders of magnitude difference. Hence, many scientists will still simulate systems smaller than 107 billion atoms and

it is important to analyze how MD codes will perform in this regime, which we can loosely call "strong scaling".

We will consider three cases for smaller problem sizes where the ratio of number of atoms to cores is 50, 20 and 5 respectively. Each of these cases will require the splitting of the basic cell of dimensions $16 \times 16 \times 16$ Å, containing 400 atoms, into a number of smaller cells:

- 50 atoms per core (50 billion atoms) – Dimensions of each cell will be $8 \times 8 \times 8$ Å.
- 20 atoms per core (20 billion atoms) – Dimensions of each cell will be $5.33 \times 5.33 \times 8$ Å.
- 5 atoms per core (5 billion atoms) – Dimensions of each cell will be $4 \times 4 \times 4$ Å (see Figure 1, extreme right).

Based on the total number of atoms in each of these smaller simulations, we can calculate the time per step for these cases (see Table I).

# Atoms	Atoms/core	Time (ms)
107 billion	100	3.602
53.6 billion	50	1.801
21.5 billion	20	0.720
5.4 billion	5	0.180

TABLE I
TIME PER STEP BOUNDS FOR MD SYSTEMS OF VARYING SIZES

Each case leads to different amounts of computation and communication and we can write equations for the smaller problems, similar to the weak scaling case:

$$T_{50} = \frac{1}{\eta} \times \frac{N_{50}}{P_c} \times 33547 \times t_c + 1856 \times \left(t_s + \frac{N_{50}}{P_c} 4t_w \right)$$
$$T_{20} = \frac{1}{\eta} \times \frac{N_{20}}{P_c} \times 33547 \times t_c + 2672 \times \left(t_s + \frac{N_{20}}{P_c} 4t_w \right)$$
$$T_{5} = \frac{1}{\eta} \times \frac{N_{5}}{P_c} \times 33547 \times t_c + 5120 \times \left(t_s + \frac{N_{5}}{P_c} 4t_w \right)$$

Putting the values of $N_{50}/P_c = 50$, $N_{20}/P_c = 20$, $N_5/P_c = 5$ and $t_c = 0.1$ ns,

$$t_s + 200t_w < 9.7 \times 10^{-7} - \frac{1}{\eta} \times 9.04 \times 10^{-8}$$
$$t_s + 80t_w < 2.69 \times 10^{-7} - \frac{1}{\eta} \times 2.51 \times 10^{-8}$$
$$t_s + 20t_w < 3.52 \times 10^{-8} - \frac{1}{\eta} \times 3.28 \times 10^{-9}$$

Using these equations, we plot the feasibility regions for 5 to 107 billion atoms in Figure 3. It is evident that smaller problem sizes put stronger constraints on the network. For example, doing a 5.4 billion atom simulation at exascale would require a latency in the range of 10 nanoseconds and a bandwidth in the range of 10 GB/s.

IV. COSMOLOGICAL SIMULATIONS

Cosmological simulations are used to understand the origin and evolution of stars, galaxies and the universe. The universe consists of two basic types of matter–baryonic matter composed of atoms and molecules and non-baryonic "dark" matter whose composition is unknown. Baryonic matter forms the part of the universe that astronomers can see directly and requires gas dynamics simulations. Dark matter, the dominant constituent of the universe for a significant portion of the time scales of interest, can be considered a collisionless fluid and can be simulated using N-body dynamics.

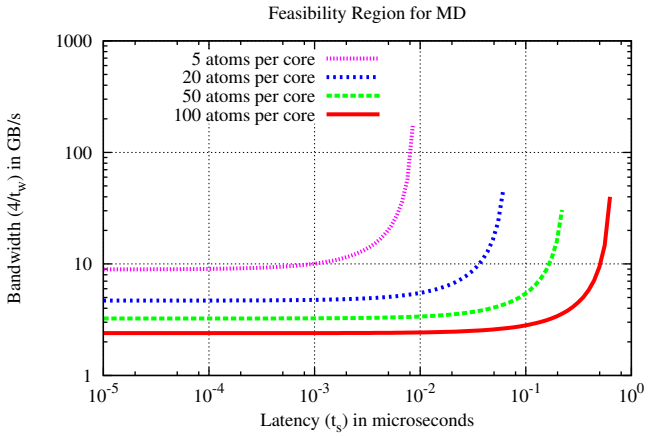

Fig. 3. Latency and bandwidth requirements for MD (smaller problem sizes)

For the purpose of discussion in this paper, we will concentrate on N-body simulations which are performed by codes known as *gravity solvers*. There are different approaches to solving the N-body problem: 1. Direct methods where all particle interactions are considered explicitly leading to $\mathcal{O}(N^2)$ computation, 2. Tree methods which involve a hierarchical multipole expansion reducing the complexity to $\mathcal{O}(N \lg N)$, and 3. Particle-mesh or "grid" methods where forces are calculated on a structured mesh. Examples of applications which use tree methods are PkdGRAV [8] and ChaNGa [9]. Examples of grid/AMR codes are Enzo [10] and FLASH [11]. We conduct our analysis in the context of the tree-based Barnes-Hut method [19], which gives an $\mathcal{O}(N \lg N)$ algorithm for simulating self-gravitating systems.

We begin by presenting an overview of N-body computations with the Barnes-Hut algorithm. First, particles are divided among cores through domain decomposition of the simulated universe, represented by a cube. We perform our analysis in the context of Oct decomposition, which entails the division of the simulation space into geometrically uniform subregions (or *cells*) in a recursive manner. This division places a tree-structure on the simulation space: the root of the tree represents the entire simulation space, which we assume to be a cube of length c. This cube is divided into eight cells of length $c/2$, each representing a child of the root cell. Each of these eight cells has eight children of its own, and so on. Particles are grouped into appropriately sized *buckets* of particles, which form the leaves of this tree. Each of the P_c cores holds a section of the space represented by a contiguous set of buckets. Therefore, the tree is distributed across cores.

Forces on particles are calculated by performing a traversal of the Barnes-Hut tree for each bucket. This procedure is carried out on a per-bucket (as opposed to a per-particle) basis to amortize the traversal cost over several proximal particles, while keeping the amount of extra work done because of clustering to a minimum. The traversal for a bucket b at depth d begins at the root of the tree. For each cell n that is encountered, an *acceptance criterion* is applied to decide whether or not n is sufficiently distant from b. The acceptance criterion is parameterized by the *opening angle*, θ_T, which is constant. Let D_n be the length of cell n, and $r_{b,n} = |\boldsymbol{r}_b - \boldsymbol{r}_n|$, the distance between b and n. If $D_n/r_{b,n} < \theta_T$, the force on b due to all the particles within cell n may be approximated by the multipole expansion of n, M_n. If not, each child c of n must be considered in turn by *expanding* the cell n. Note that expanding a cell may require communication, since the tree is distributed across processors. The per-bucket traversal procedure is outlined in Algorithm 2 below. The force computed for each particle is used to update its position and velocity. Subsequently, domain decomposition is performed and a new iteration of force evaluation begins.

Algorithm 2 $BarnesHut(n, b)$: Cell n, Bucket b

```
if n is a bucket then
    bucketForces(b, n)
else if D_n/|r_b − r_n| < θ_T then
    cellForces(b, M_n)
else
    for all c ∈ children(n) do
        BarnesHut(c, b)
    end for
end if
```

We estimate the resolution of exascale simulations by extrapolating from state-of-the art simulations at the Petaflop/s level. It has been observed that about 2^{13} particles per core are required to maintain a good scaling profile for ChaNGa [9] for current machines. Recall that the total number of interactions per time step for an N-body system using the Barnes-Hut technique is $\mathcal{O}(N \lg N)$. This suggests that in order to generate an equivalent amount of work per processor core as we scale, the number of particles required per core *decreases* slightly:

$$\frac{N' \lg N'}{P'_c} = \frac{N \lg N}{P_c}$$

where N' and N are the numbers of particles required at petascale (P'_c cores) and exascale (P_c cores). Given $P'_c = 2^{20}$, and particles per core at petascale = 2^{13}, we get $N' = 2^{33}$. Using $P_c = 2^{30}$, results in a total of $N = 6.2 \times 2^{40} \approx 6.82$ trillion particles, i.e. roughly 6350 particles per core at exascale.

A. Computation

Realistic simulations treat the input set of particles as part of larger structures through the application of periodic boundary conditions. When coupled with the uniformity of particle distribution, the assumption of periodic boundary conditions simplifies our analysis. In essence, each bucket of particles in the root cell, regardless of its position, interacts with exactly the same number of cells and particles as every other. This allows us to focus our analysis of the number of expansions to a single, arbitrary bucket. In order to calculate the number of cell expansions required to compute the gravitational forces on the particles of a bucket b, we consider each level of the tree in turn. Let the center of mass of b be situated at O. In general, a cell n at level d of the tree has an edge length of $c/2^d$. This cell will be expanded by b if the distance between O and the center of mass of n, written $r_{b,n}(d)$, is such that

$$r_{b,n}(d) \leq \frac{c}{2^d.\theta_T}$$

Therefore, bucket b expands all the cells at level d whose centers lie within the sphere of radius $r_T(d) = c/2^d.\theta_T$ described around O. Usually, $0.5 \leq \theta_T < 1$, so that $c/2^d \leq r_T(d) < c/2^{d-1}$. This means that we can calculate lower and upper bounds on the number of cells at level d that b expands by considering two spheres around O, of radius $r_0(d) = c/2^d$ and $2r_0(d) = c/2^{d-1}$, respectively. In particular, the centers of mass of 7 cells of length $c/2^d$ fall within the first sphere. Furthermore, 125 cells of the same edge length intersect with the second sphere. However, the centers of mass of only 33 of these 125 fall within the second sphere. By definition these are the only cells that are expanded. Therefore, bucket b expands between 7 and 33 cells at depth d, for every d. Therefore, the total number of calls to the acceptance criterion can be bounded above by:

$$33 \times \frac{N}{B} \times \lg \frac{N}{B} \times \frac{1}{\lg 8} = 11 \times \frac{N}{B} \times \lg \frac{N}{B}$$

A similar argument can be used to calculate the number of interactions performed. Consider a bucket b for which the Barnes-Hut tree is being traversed. Let $I_{bc}(d)$ be the number of bucket-to-cell interactions at depth d and $C_e(d)$, the number of cells expanded at depth d. Then, $I_{bc}(d) = 8 \times C_e(d-1) - C_e(d)$. Since $C_e(d) = 33$ for all d, we have $I_{bc}(d) = 7 \times 33$. With B particles per bucket and $\lg(N/B)/3$ levels in the tree, the total number of interactions per bucket equals $77 \times B \lg(N/B)$. For N/B buckets, this implies a total of $I_{\text{pc}} = 77 \times N \lg(N/B)$ particle-cell interactions. To this, we add the number of particle-particle interactions: each bucket expands 32 other buckets, resulting in $\approx 33 \times B \times B$ particle-particle interactions per bucket, and $I_{\text{pp}} = 33 \times BN$ particle-particle interactions in all. The total number of flops for each type of interaction is listed below:

$$312 \times 77 \times N \times \lg \frac{N}{B} + 38 \times 33 \times B \times N$$

The figures of 312 flops for each hexadecapole particle-cell interaction and 38 flops per particle-particle interaction are obtained from ChaNGa. In order to obtain a computational rate of at least 1 Exaflop/s, for $N = 6.2 \times 2^{40}$ and $B = 10$, using equation (III.1) we get,

$$\frac{24024 \times N \lg(N/B) + 1254 \times BN}{T} > 10^{18}$$

$$or \quad T < 6.52 \tag{IV.1}$$

This suggests that we need a time per step of 6.52 seconds to achieve 1 Exaflop/s performance.

B. Communication

Now, we estimate the volume of communication generated by the Barnes-Hut algorithm. Recall that particles are grouped into buckets of size B each. The even distribution of N particles among P_n processor nodes results in $N/(P_nB)$ buckets per processor node. Furthermore, given the even distribution of particles, each node receives an approximately cubic subdomain of edge length $a = c/\sqrt[3]{P_n}$. This is depicted as the striped area in Figure 4. Let n_b be the number of buckets along an edge of the cube. Then, $n_b^3 = N/(P_nB)$, so that $n_b = \sqrt[3]{N/(P_nB)}$. The processor cores that perform traversals for buckets within this volume request data in the form of cells and particles, both from remote nodes and local cores within the same node. However, buckets closer to the center of this cube request strictly a subset of the *remote* cells and particles requested by buckets closer to the faces. This observation is leveraged in production quality simulators by "caching" cells and particles fetched from remote sources, resulting in the reuse of remote data, and reducing the communication cost of the algorithm. Therefore, we attribute the aggregate remote communication generated by a processor node to the union of all cells and particles requested by the buckets along the faces of the cube. As shown in Figure 4, with a bound of $\theta_T = 0.5$, buckets along the faces of the cube expand a total of $12n_b^2 + 12 \times 3n_b + 8$ remote buckets of edge length $c\sqrt[3]{B/N}$. Therefore, each node requests the particles of $C^{\text{bkts}} = 12n_b^2 + 36n_b + 8$ buckets. The buckets along the faces also expand a total of $12(n_b/2)^2 + 36(n_b/2) + 8$ cells with edge length $2c\sqrt[3]{B/N}$, $12(n_b/4)^2 + 36(n_b/4) + 8$ cells with edge length $2^2c\sqrt[3]{B/N}$, etc. Therefore, the number of cells requested from remote nodes up to size $a = c/\sqrt[3]{P_n}$ is:

$$\begin{aligned} C_1^{\text{cell}} &= \sum_{i=0}^{\lg n_b} \left(12\left(\frac{n_b}{2^i}\right)^2 + 36\left(\frac{n_b}{2^i}\right) + 8\right) \\ &= 16n_b^2 + 72n_b + 8\lg n_b - 32 \quad \text{cells} \end{aligned}$$

For $i \leq \lg n_b$, the above reasoning is valid since there are multiple cells (or for $i = \lg n_b$, a single cell of edge length a) lining a processor node's subvolume. We must consider cells with edge length greater than a separately. Notice that there is an asymmetry of communication volume between processor nodes: two nodes may request slightly different numbers of higher-level cells depending on their positions within the simulated space. The greatest difference in the number of cells expanded occurs between the eight central processor nodes and the ones situated at the eight corners of the simulated universe. Even so, with $\theta_T = 0.5$, the number of cells expanded by the processor nodes in the corners equals 31, whereas 30 cells are expanded by the eight central processor nodes. We assume that each processor stores the root cell representing the entire simulation space. Therefore, we bound

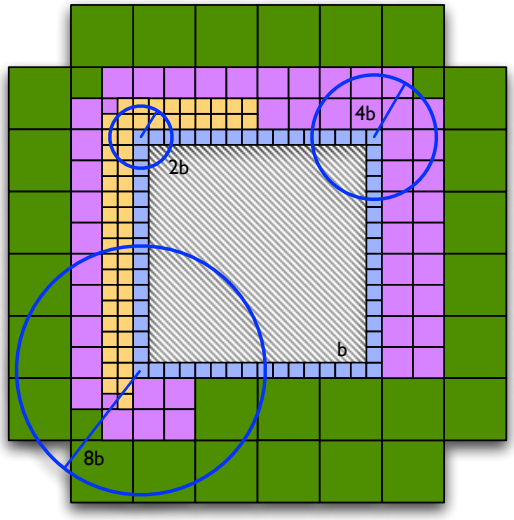

Fig. 4. Communication pattern of a single node at the bottom three depths in the Barnes-Hut tree. The striped region in the center represents the cubic subvolume of particles assigned to the node, and the immediate squares surrounding it represent the buckets along its faces. Progressively larger squares represent remote cells at different depths that are requested by the node for $\theta_T = 0.5$. Circles of radii $2b$, $4b$ and $8b$ described around the centers of corner buckets determine which cells are requested.

the amount of communication generated per processor by the expansion of higher-level cells as follows:

$$C_2^{\text{cell}} = 31\left(\frac{\lg P_n}{3} - 1\right) \quad \text{cells}$$

The expansion of each cell yields eight children. We assume that for each expanded cell, a single message is generated which contains all its children. This model may be extended so that whenever a cell is expanded by a processor node, it receives a subtree of depth m below that cell. The tradeoff to consider there is that between the number of messages (fewer for larger m) and the amount of network bandwidth wasted (more for larger m) because of requests for cells that are never needed by the traversal. We keep our analysis simple by setting $m = 1$. This results in about $(C_1^{\text{cell}} + C_2^{\text{cell}})/8$ messages containing eight cells each, and C^{bkts} messages to communicate B particles each.

We now use C_1^{cell}, C_2^{cell} and C^{bkts} to calculate constraints on t_s and t_w. In the following, we assume that the total number of flops are distributed evenly across processors (i.e. we assume perfect load balance). By setting $P_n = 2^{20}$, we get $n_b \approx 87$. Therefore, the total number of cell expansion messages equals $(C_1^{\text{cell}} + C_2^{\text{cell}})/8 \approx 15946$, and the total number of particle messages is $C^{\text{bkts}} = 93968$. We take the multipole moments of each cell to require 224 bytes (56 words) and each particle's coordinate information to be 40 bytes. This leads to 100 words for 10 particles in one bucket (these values are taken from ChaNGa). Assuming a network free of contention, this results in a communication time of

$$T_{comm} = 15946(t_s + 56t_w) + 93968(t_s + 100t_w)$$

Finally, we use the above expression for communication in equation (IV.1),

$$\frac{6.52 \times 10^{18}}{P_c} \times \frac{t_c}{\eta} + (1.1 \times 10^5 t_s + 1.03 \times 10^7 t_w) < 6.52$$

$$t_s + 93.62 t_w < 59.2 \left(1 - \frac{0.093}{\eta}\right) \times 10^{-6}$$

This equation is plotted in Figure 5. If we keep message latency constant, the bandwidth requirements increase as η decreases. It has been observed that an optimized version of ChaNGa delivers about 15% of the theoretical maximum performance on a single core. Therefore, a value of $\eta = 0.125$ is appropriate.

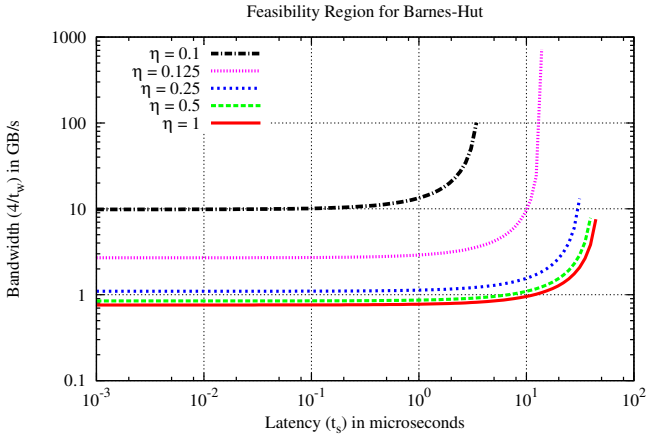

Fig. 5. Latency and bandwidth requirements for the Barnes-Hut simulation of 6.8 trillion uniformly distributed particles.

C. Memory requirements

In addition to its local cell and particle data, a processor node in our model must store particle and cell data requested from remote sources. This allows the caching and reuse of requested data, thereby reducing the amount of communication that must be performed. At the same time, this cache increases the transient memory requirements of each processor node. The local data comprises N/P_n particles and an octree of depth $\lg(N/P_n B)/3$. Furthermore, each processor node stores the path from the root cell to the cell that represents its subvolume. The length of this path is $\lg P_n/3$ nodes. Assuming the amount of data stored per local particle to be $S_p = 152$ bytes and that stored per cell to be $S_c = 224$ bytes, the amount of memory required for the local data is:

$$M_{local} = \left(8^{\frac{\lg(N/P_n B)}{3}} + \frac{\lg P_n}{3}\right) S_c + \frac{N}{P_n} S_p$$
$$= \left(\frac{N}{P_n B} + \frac{\lg P_n}{3}\right) S_c + \frac{N}{P_n} S_p$$

Substituting the values for the various variables, we get $M_{local} = 1.08$ GB per node. We estimate the amount of memory required to cache remote data by using the number of cells and buckets requested from remote sources. Recall that these values are given by $(C_1^{\text{cell}} + C_2^{\text{cell}})$ and C^{bkts}, respectively. Therefore, the total memory required per processor node to cache remote data is:

$$M_{remote} = (C_1^{\text{cell}} + C_2^{\text{cell}}) S_c + C^{\text{bkts}} B S'_p$$

where $S'_p = 40$ bytes is the amount of memory required per cached remote particle. Fixing the values of the various variables, we get $M_{remote} = 171$ MB per node.

D. Smaller problem sizes

Not all cosmological simulations conducted at exascale will use such large systems of particles. In particular, studies of isolated star clusters and planet disk formation require far fewer particles for faithful simulation. Such simulations on small-scale structures are fairly important in themselves. For this reason, we discuss the feasibility of conducting experiments of sizes significantly smaller than the large, 6.8 trillion particle simulation discussed previously. In particular, we analyze the constraints on machine characteristics as we scale down the problem size and attempt to maintain the same level of performance as seen with the large simulation.

We consider three particle systems of similar distribution characteristics to the 6.8 trillion particle data set introduced previously. The total number of particles for each of these data sets is given in Table II. The number of particles per core for each is also shown. In each case, the analysis for communication volume is roughly the same as outlined in Section IV-A and Section IV-B. each of the cases, we set η to a realistic value of 0.125, or $\approx 13\%$. The equations relating t_s and t_w for the three systems are listed below, in order:

$$t_s + 93.56 t_w < 8.94 \times 10^{-6}$$
$$t_s + 93.45 t_w < 5.16 \times 10^{-6}$$
$$t_s + 93.29 t_w < 2.92 \times 10^{-6}$$

# Particles	Particles/core	Time (s)
6.8 trillion	6350	6.52
1.7 trillion	1588	1.55
0.43 trillion	397	0.37
0.11 trillion	99	0.09

TABLE II
TIME PER STEP FOR BARNES-HUT SIMULATIONS OF DIFFERENT SIZES.

These constraints are depicted graphically in Figure 6. Notice that with fewer particles per core, keeping the overhead of transmission constant, we require more bandwidth to maintain a performance level of one Exaflop/s.

V. FINITE ELEMENT SOLVERS

Finite element solvers are the ones that are most commonly employed to solve unstructured grid problems, as their expression of the solution as a sum of basis functions over elements dovetails naturally with setup of an unstructured grid problem

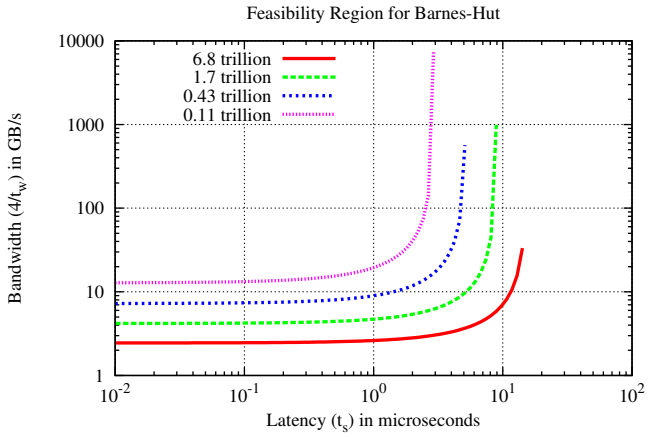

Fig. 6. Constraints on machine characteristics for Barnes-Hut simulations of different data sets. A value of $\eta = 0.125$ was used for each data set.

as a domain partitioned into a mesh of elements. A typical application of a finite element solver involves two phases to consider. There is an assembly phase, in which a linear system is put together, and a solve phase in which that system is solved. For a linear problem, there is just one assembly phase, and one or more solve phases – one for a time-independent problem, and one per time step for a time-dependent problem. For a nonlinear problem, for which the solution process is an iterative scheme comprised of the formation and solution of multiple linear systems, the process of assembly and solve for linear problems is repeated until convergence.

The problem setup in our analysis is based on the recent work [20], which strongly scales a finite element solver to near-petascale machines. Problem partitioning is by elements, so that each processor has complete information about the elements in its individual domain. Shared degrees of freedom, which occur wherever there are mesh points on a processor boundary, are stored redundantly. Figure 7 gives a simple example. Assuming a good partitioning of the problem among processors, the amount of shared degrees of freedom will just be the surface area of the individual processor domains, $\mathcal{O}\left(\sqrt{\frac{N}{P}}\right)$ for a 2D problem and $\mathcal{O}\left(\left(\frac{N}{P}\right)^{2/3}\right)$ for a 3D problem, assuming N global degrees of freedom. System assembly, which involves summing the contributions of each element into a global sparse matrix, can be accomplished with just nearest-neighbor communication of local values for shared degrees of freedom. As the matrix and vector entries themselves are integrals, the number of floating-point operations depends on the specific integration rule used.

There is a concern of scalability in terms of the amount of data transferred. Completely assembling the matrix entries requires the square of the surface area. However, this is not necessary. Completely assembling only the vector entries, which involves data transfer that is just linear in the surface area, is sufficient if a Krylov subspace method, which is based on matrix-vector multiplication, is used for the linear solve.

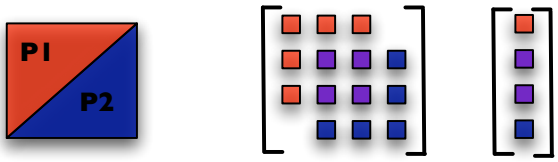

Fig. 7. Example of redundant storage of an unstructured mesh. The triangles belong to two different processors, P1 and P2, and each node represents a degree of freedom. Red entries are stored on P1, blue entries on P2, and purple entries on both processors.

If only the non-shared degrees of freedom are assembled, a nearest-neighbor exchange and summation of shared degrees of freedom after the product of this matrix with a completely assembled vector will give the same result as a product between a completely assembled matrix and completely assembled vector. With nearest-neighbor communication and a scalable amount of data being transferred, we turn our attention to the solve phase.

In the solve phase, a linear solver is used to solve the previously assembled linear system. Krylov subspace methods, which are based on matrix-vector multiplication, are a popular choice, and in fact the only choice when performing assembly as previously outlined. There are many different Krylov subspace methods [21], and the choice of method depends on the specific problem being solved. As our study is introductory, we examine here the simplest Krylov method, conjugate gradient (CG), which is the method of choice for problems that are symmetric and positive definite. Pseudocode is given in Algorithm 3.

Algorithm 3 CG(A,b,x_0,$rtol$)

$r_0 \leftarrow b - Ax_0$
$p_0 \leftarrow r_0$
$k \leftarrow 0$
while $||r_k||_2 \geq rtol$ **do**
  $\alpha_k \leftarrow \frac{r_k^T r_k}{p_k^T A p_k}$
  $x_{k+1} \leftarrow x_k + \alpha_k p_k$
  $r_{k+1} \leftarrow r_k - \alpha_k A p_k$
  $\beta_k \leftarrow \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k}$
  $p_{k+1} \leftarrow r_{k+1} + \beta_k p_k$
  $k \leftarrow k + 1$
end while
return x_k

The setup to CG requires one matrix-vector product (Ax_0), one vector subtraction ($b - Ax_0$), and one dot product ($r_0^T r_0$). The iteration loop requires one matrix-vector product (Ap_k), two vector additions ($x_k + \alpha_k p_k$ and $r_{k+1} + \beta_k p_k$), one vector subtraction ($r_k - \alpha_k A p_k$), and two dot products ($p_k^T A p_k$ and $r_{k+1}^T r_{k+1}$). We define the terms we will use to construct a performance model below:

- N – global number of degrees of freedom
- n_i – number of degrees of freedom stored on processor i

- $\tilde{n}_i$ – number of degrees of freedom stored on processor i that are shared with other processors
- s_i – average number of neighbors for degrees of freedom stored on processor i
- p_i – number of processor neighbors of processor i

On processor i, each matrix-vector multiply requires $2s_in_i$ flops, and p_i sends each consisting of $\tilde{n}_i$ floating-point values. The vector additions and subtractions require n_i flops in the setup and $2n_i$ flops in the loop, with no communication. The dot products, which are accomplished using allreduce operations, require $\frac{N}{P} + \lg P$ flops and $2 \times \lg P$ sends of one floating-point value for the allreduce and another $\tilde{n}_i$ flops coupled with p_i sends of $\tilde{n}_i$ floating-point values for the completion of the result.

A. Weak Scaling

We now analyze a simple weak-scaling scenario. We consider a problem solved on a 3D cubic mesh consisting of cubes each cut into five tetrahedra, as shown in Figure 8. We give each core 4K cubes, or 20K tetrahedra, to correspond to common elements per core counts for problems being solved today, with each processor's portion a $16 \times 16 \times 16$ cube. This results in $n_i = 17^3 = 4,913$ degrees of freedom stored on each processor, and $N = 16385^3 \approx 4.4$ trillion global degrees of freedom. Each processor would send messages to at most $p_i = 6$ neighbors during point-to-point communication, with a total of $\tilde{n}_i = 17^3 - 15^3 = 1538$ floating-point entries sent. The average number of neighboring degrees of freedom is $s_i = 18$ excluding points on the boundary of the global domain.

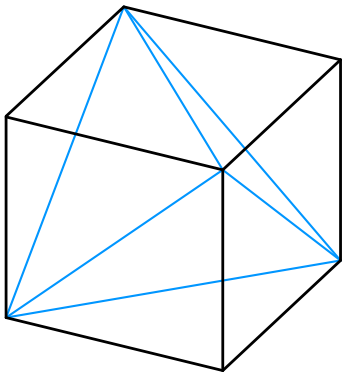

Fig. 8. Base unit of mesh, cube cut into five tetrahedra. Four tetrahedra surround the one in the center, with one of those four hidden behind the center tetrahedron.

Since there are 1024 cores per node on our hypothetical exascale machine, we for each node group the portions of all the cores on that node into an $8 \times 8 \times 16$ cube. This requires us to modify the numbers given above when writing down the performance model equation. The computation terms are unchanged; however, the number of sends becomes the surface area of the node, which is $8 \times 8 \times 16 - 6 \times 6 \times 14 = 520$. The number of elements sent is multiplied by this amount for point-to-point messages. The logarithmic terms in the allreduce time become logarithms of the number of nodes instead of the number of cores. Assuming a binary tree pattern for the allreduce, we get that the setup time for CG is

$$\begin{aligned} T_{\text{CG}}^{\text{setup}} = \frac{1}{\eta} \times \left((2s_i + 1)n_i + \frac{N}{P_c} + \lg P_n \right) t_c \\ + 2(520 + \lg P_n)t_s \\ + 2(520\tilde{n}_i + \lg P_n)t_w \end{aligned}$$

and the solve time is, per iteration

$$\begin{aligned} T_{\text{CG}}^{\text{iter}} = \frac{1}{\eta} \times \left((2s_i + 6)n_i + \frac{N}{P_c} + 2\lg P_n \right) t_c \\ + 2(520 + 2\lg P_n)t_s \\ + 2(520\tilde{n}_i + 2\lg P_n)t_w \end{aligned}$$

Allowing t_s and t_w to vary and fixing all other parameters, we plot the region

$$\frac{P_c(2s_i + 1)n_i + N + 2\lg P_n}{T_{\text{CG}}^{\text{iter}}} \geq 10^{18}$$

which simplifies to

$$1120t_s + 1599600t_w \leq 2.26 \times 10^{-4} - \frac{1}{\eta} \times 2.10 \times 10^{-5}$$

to see what machine parameters are required to achieve exascale performance for an iteration of CG. The plot, which is in Figure 9, shows a latency requirement on the order of tenths of microseconds and a node bandwidth requirement of tens of gigabytes per second.

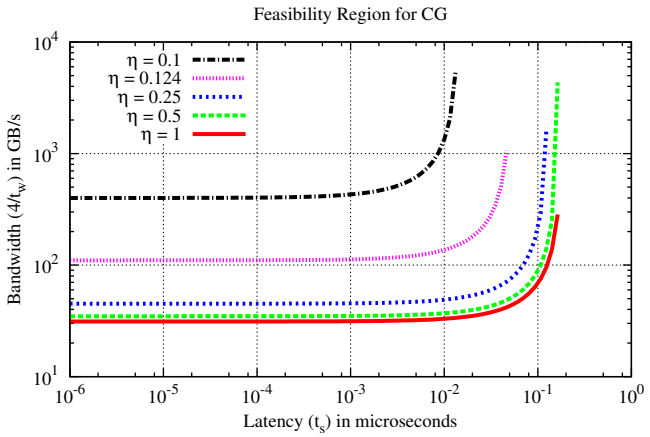

Fig. 9. Feasibility region for conjugate gradient iteration.

B. Memory Requirements

Finite element codes require storage of the mesh and the (sparse) linear system. This is not a scalability concern, as the storage of each scales with the number of degrees of freedom. The storage beyond this is not a concern either. The redundant storage in the scheme presented above is proportional to the surface area of the elements, and beyond the linear system, CG keeps only four additional vectors in memory. Even if another Krylov solver were used, the memory used by the solver is not

a scalability concern: CG-based methods keep only a handful of vectors in memory in addition to the linear system, and even the most general-purpose Krylov method, GMRES, in practice keeps a number of vectors in memory that is constant in the problem size due to its being restarted after a fixed number of iterations [21].

C. Smaller Problem Sizes

Applying the same analysis as for weak scaling, we can see what the feasibility region would be for smaller problem sizes. Figure 10 shows regions for five problem sizes, ranging from the weak scaling scenario discussed earlier to the smallest possible problem, with one cube per core. The successive feasibility regions are:

$$1120t_s + 1599600t_w \leq 2.26 \times 10^{-4} - \frac{1}{\eta} \times 2.10 \times 10^{-5}$$

$$1120t_s + 401520t_w \leq 3.34 \times 10^{-5} - \frac{1}{\eta} \times 3.12 \times 10^{-6}$$

$$1120t_s + 102000t_w \leq 5.70 \times 10^{-6} - \frac{1}{\eta} \times 5.35 \times 10^{-7}$$

$$1120t_s + 27120t_w \leq 1.23 \times 10^{-6} - \frac{1}{\eta} \times 1.18 \times 10^{-7}$$

$$1120t_s + 8400t_w \leq 3.62 \times 10^{-7} - \frac{1}{\eta} \times 3.77 \times 10^{-8}$$

Specific model parameters for each problem are shown in Table III. The latency and bandwidth requirements become increasingly restrictive with each reduction in size, to the point of needing sub-nanosecond latencies and node bandwidth on the order of hundreds of gigabytes per second.

Problem	Cubes/core	N	n_i	$\tilde{n}_i$
1	4096	4.40×10^{12}	4913	1538
2	512	5.50×10^{11}	729	386
3	64	6.88×10^{10}	125	98
4	8	8.60×10^{9}	27	26
5	1	1.08×10^{9}	8	8

TABLE III
PERFORMANCE MODEL PARAMETERS FOR DIFFERENT PROBLEM SIZES, RANGING FROM THE ORIGINAL ONE STUDIED IN THE WEAK SCALING STUDY TO ONE WITH ONE CUBE PER CORE. s_i IS NOT SHOWN BECAUSE IT REMAINS UNCHANGED.

D. Additional Issues

The analysis presented here has only scratched the surface when it comes to solving unstructured grid problems. There are a number of other factors that would be considered in a complete analysis. The partitioning of the grid naturally matters, as a poor partition would substantially degrade performance. The solver itself is a factor. Conjugate gradient is one of several Krylov subspace methods, and it is applicable only when the problem being solved is symmetric and positive definite. Other problems require different solvers that will also need to be studied.

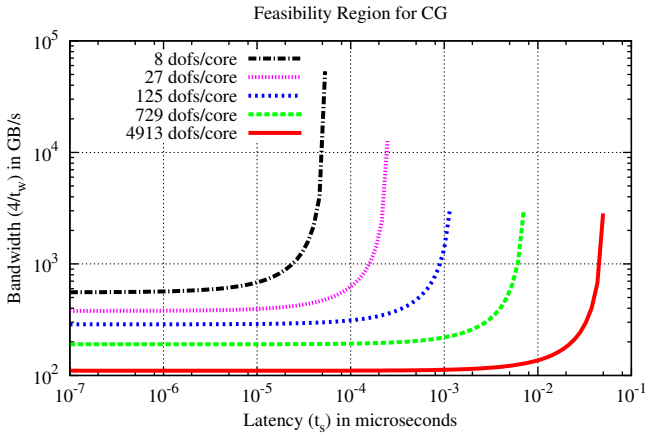

Fig. 10. Feasibility regions for conjugate gradient iteration for several problem sizes. For each successively smaller problem size, the number of cubes per core decreases by 2 in each dimension.

There is also the issue of preconditioning that we did not consider here. Preconditioning involves applying some kind of fast incomplete solve to the system during each iteration with the goal of speeding up convergence [21]. This means extra expense per iteration, and on a parallel computer care must be taken to choose a preconditioner that parallelizes well. Poor or no preconditioning results in a high iteration count that increases with the global problem size. In the case of conjugate gradient on a 3D problem, the number of iterations increases as $N^{1/3}$ [22], for a (non-ideal) overall asymptotic computation time of $\mathcal{O}(N^{4/3})$. [22] used multigrid as a preconditioner in CG to both dramatically reduce the iteration count and the rate of increase. The scalability of multigrid is itself a subject of much study [23]. Clearly, the issue of preconditioning is one of vital importance for ensuring scalability of unstructured grid problems.

VI. SUMMARY

This paper presented architectural constraints imposed by weak scaling and smaller problem sizes for several application classes to achieve 1 Exaflop/s performance on future machines. In general for all applications discussed, at a lower sequential efficiency than 1 ($\eta < 1$), the constraints on the network latency and bandwidth tighten. High latency and bandwidth requirements, especially for $\eta < 1$ and smaller problem sizes emphasize the importance of continuing research in developing communication-minimizing algorithms, as well as high-bandwidth network links and system infrastructure that minimizes the diameters of the interconnect networks.

For comparison between the application classes, Figure 11 presents the feasibility regions of the three classes for weak scaling (for $\eta = 1$). We can see that FEM puts the tightest constraints on both network latency and bandwidth. MD codes have the smallest bandwidth requirements because information of a small number of atoms is exchanged at every step. The modest communication requirements for MD and N-body

problems are due to the fact that each communicated value is used in a large number of floating point calculations (leading to a higher degree of reuse compared to FEM).

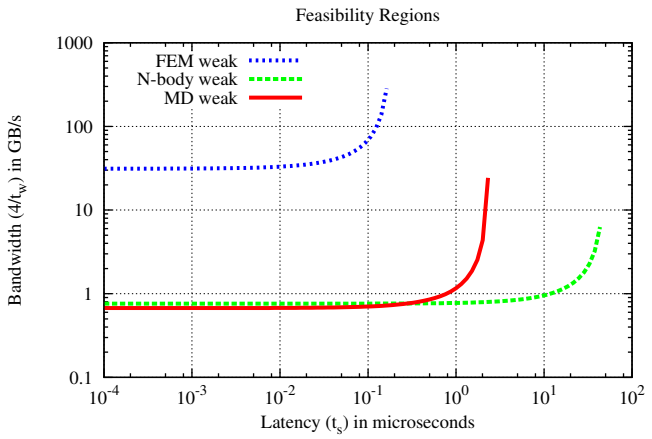

Fig. 11. Feasibility regions for molecular dynamics (MD), cosmology (N-body) and finite element solvers (FEM) for weak scaling to achieve 1 Exaflop/s

Smaller problem sizes for all the application classes lead to stronger constraints on the network latency and bandwidth. However, it is important to remember that the latency constraints can be relaxed to some extent since our analysis assumes serialization of messages originating from all cores on a node through the NIC or switch on the node. We expect that future machines will allow several cores on a node to inject messages on the network simultaneously. The memory requirements for all these application classes is not a scalability concern (although the requirements for Barnes-Hut are higher than the other two). This suggests that machines with low memory per core may be realistic in the future.

The analysis in this paper has made as few assumptions as possible. However, two assumptions which have simplified our analysis are absence of load imbalance and network contention. In a future study, we will analyze the impact of both factors on application performance. This is a preliminary study of the three application classes and we plan to do a more in-depth analysis of each application class which was impossible in this paper due to limited space.

ACKNOWLEDGMENTS

This work was supported in part by a DOE Grant DE-SC0001845 for HPC Colony II, NSF Grants ITR-HECURA-0833188 and SGER 0837719 and by the Institute for Advanced Computing Applications and Technologies (IACAT). The authors would like to thank Prof. Tom Quinn from the University of Washington for discussions about cosmological simulations.

REFERENCES

[1] "Top500 supercomputing sites," http://top500.org.

[2] L. V. Kale, "Early Application Development/Tuning and Application Characterization/Segmentation," *International Journal of High Performance Computing Applications*, vol. 23, no. 4, pp. 411–412, October 2009.

[3] A. Bhatele, S. Kumar, C. Mei, J. C. Phillips, G. Zheng, and L. V. Kale, "Overcoming scaling challenges in biomolecular simulations across multiple platforms," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, April 2008, pp. 1–12.

[4] P. K. Weiner and P. A. Kollman, "AMBER: Assisted model building with energy refinement. a general program for modeling molecules and their interactions," *Journal of Computational Chemistry*, vol. 2, p. 287, 1981.

[5] D. Van Der Spoel, E. Lindahl, B. Hess, G. Groenhof, A. E. Mark, and H. J. C. Berendsen, "Gromacs: Fast, flexible, and free," *Journal of Computational Chemistry*, vol. 26, pp. 1701–1718, December 2005.

[6] K. J. Bowers, E. Chow, H. Xu, R. O. Dror, M. P. Eastwood, B. A. Gregersen, J. L. Klepeis, I. Kolossvary, M. A. Moraes, F. D. Sacerdoti, J. K. Salmon, Y. Shan, and D. E. Shaw, "Scalable algorithms for molecular dynamics simulations on commodity clusters," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006.

[7] B. G. Fitch, A. Rayshubskiy, M. Eleftheriou, T. J. C. Ward, M. Giampapa, and M. C. Pitman, "Blue matter: Approaching the limits of concurrency for classical molecular dynamics," in *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM Press, 2006.

[8] M. D. Dikaiakos and J. Stadel, "A performance study of cosmological simulations on message-passing and shared-memory multiprocessors," in *Proceedings of the International Conference on Supercomputing - ICS'96*, Philadelphia, PA, December 1996, pp. 94–101.

[9] P. Jetley, F. Gioachin, C. Mendes, L. V. Kale, and T. R. Quinn, "Massively parallel cosmological simulations with ChaNGa," in *Proceedings of IEEE International Parallel and Distributed Processing Symposium 2008*, 2008, pp. 1–12.

[10] B. Oshea, G. Bryan, J. Bordner, M. Norman, T. Abel, R. Harkness, and A. Kritsuk, "Introducing enzo, an amr cosmology application," in *Adaptive Mesh Refinement - Theory and Applications*, ser. Lecture Notes in Computational Science and Engineering. Springer Berlin Heidelberg, 2005, vol. 41, pp. 341–349.

[11] G. Weirs, V. Dwarkadas, T. Plewa, C. Tomkins, and M. Marr-Lyon, "Validating the Flash code: vortex-dominated flows," in *Astrophysics and Space Science*. Springer, 2005, vol. 298, pp. 341–346.

[12] D. Braess, *Finite elements: Theory, fast solvers, and applications in solid mechanics*, 3rd ed. Cambridge University Press, 2007.

[13] H. Gahvari and W. Gropp, "An introductory exascale feasibility study for ffts and multigrid," in *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, April 2010, pp. 1–9.

[14] A. Bhatele, G. Gupta, L. V. Kale, and I.-H. Chung, "Automated Mapping of Regular Communication Graphs on Mesh Interconnects," in *Proceedings of International Conference on High Performance Computing (HiPC)*, 2010.

[15] B. Hess, C. Kutzner, D. van der Spoel, and E. Lindahl, "Gromacs 4: Algorithms for highly efficient, load-balanced, and scalable molecular simulation," *Journal of Chemical Theory and Computation*, vol. 4, no. 3, pp. 435–447, 2008.

[16] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, no. 1, pp. 1–19, 1995.

[17] L. Kalé, R. Skeel, M. Bhandarkar, R. Brunner, A. Gursoy, N. Krawetz, J. Phillips, A. Shinozaki, K. Varadarajan, and K. Schulten, "NAMD2: Greater scalability for parallel molecular dynamics," *Journal of Computational Physics*, vol. 151, pp. 283–312, 1999.

[18] M. Snir, "A note on n-body computations with cutoffs," *Theory of Computing Systems*, vol. 37, pp. 295–318, 2004.

[19] J. E. Barnes and P. Hut, "A hierarchical O(NlogN) force calculation algorithm," *Nature*, vol. 324, 1986.

[20] O. Sahni, M. Zhou, M. S. Shephard, and K. E. Jansen, "Scalable Implicit Finite Element Solver for Massively Parallel Processing with Demonstration to 160K cores," in *Supercomputing 2009*, Portland, OR, November 2009.

[21] Y. Saad, *Iterative Methods for Sparse Linear Systems*, 2nd ed. SIAM, 2003.

[22] S. F. Ashby and R. D. Falgout, "A Parallel Multigrid Preconditioned Conjugate Gradient Algorithm for Groundwater Flow Simulations," *Nuclear Science and Engineering*, vol. 124, pp. 145–159, 1996.

[23] E. Chow, R. D. Falgout, J. J. Hu, R. S. Tuminaro, and U. M. Yang, "A Survey of Parallelization Techniques for Multigrid Solvers," in *Parallel Processing for Scientific Computing*, M. A. Heroux, P. Raghavan, and H. D. Simon, Eds. SIAM, 2006, ch. 10.