ABSTRACT
This paper addresses three questions related to the use of parallel computing in Multibody Dynamics (MBD) simulation. The “why parallel computing?” question is answered based on the argument that in the upcoming decade parallel computing represents the main source of speed improvement in MBD simulation. The answer to “when is it relevant?” is built around the observation that MBD software users are increasingly interested in multi-physics problems that cross disciplinary boundaries and lead to large sets of equations. The “how?” question is addressed by providing an overview of the state of the art in parallel computing. Emphasis is placed on parallelization approaches and support tools specific to MBD simulation. Three MBD applications are presented where parallel computing has been used to increase problem size and/or reduce time to solution. The paper concludes with a summary of best practices relevant when mapping MBD solutions onto parallel computing hardware.

1 Introduction and Purpose.
Multibody dynamics (MBD) is an area of Computational Dynamics concerned with the dynamics of mechanical systems consisting of collections of bodies experiencing large displacements and possibly large deformations. The three dimensional (3D) bodies are subject to external forces and their time evolution can be constrained through kinematic constraints that couple the dynamics of pairs of bodies. The bodies can interact mutually through friction and contact, or alternatively, in Fluid-Solid Interaction (FSI) problems, their dynamics interplays with the dynamics of a fluid through a two way coupling mechanism.

Historically, MBD problems have relatively been small. A full vehicle dynamics analysis or an engine dynamics simulation in a commercial code such as ADAMS [50] typically leads to fewer than 50,000 equations. By Finite Element Analysis (FEA) or Computational Fluid Dynamics (CFD) standards, these are small problems. While nonetheless complex, MBD simulation has mostly led to computational problems that by and large have not justified heavy reliance on parallel computing. In fact, compared to solution providers in the FEA and CFD markets, all major players in the MBD arena: ADAMS, RecurDyn, LMS Virtual.Lab, or Simpack, have, until recently, been slow in adopting parallel computing in the equation formulation and numerical solution stages. There are still numerous studies in the MBD literature regarding how parallelism can reduce time to solution, see for instance [2, 12, 18, 49]. Parallel computing has already been applied in our community to address domain specific problems, see [8, 22, 67]. Finally, software for general purpose parallel MBD is discussed in [31, 42, 55].

Two forces are gradually changing the MBD landscape in relation to the simulation of very large systems. First, users of these simulation tools are requesting support for modeling features that cross the boundary of traditional MBD into multi-physics problems. These problems might include a fluid component, a granular component, a terrain component, etc., to yield models characterized by large sets of equations that stand to benefit from parallel computing. Second, once the privilege of a select group of research institutions, hardware that supports effective parallel computing is now ubiquitous. Due to low costs, less than $4000 for a 60 core Intel Xeon Phi chip or $3000 for a 2600 scalar processor Nvidia Kepler graphics processing unit (GPU) card, there
is a manifest democratization of the access to parallel computing hardware.

Given these ongoing changes in the MBD simulation arena, the purpose of this paper is fourfold: (i) briefly present actual MBD problems whose modeling results in large sets of equations hence confirming the aforementioned trend; (ii) summarize the state of the art in parallel computing and comment on how it can be leveraged in MBD simulation; (iii) present several applications in which parallel computing was used to reduce the time to solution and/or increase the size of the problem investigated; and (iv) summarize good practices relevant when mapping MBD solutions onto parallel computing hardware.

The paper is structured around these four goals as follows. Section 2 summarizes how MBD problems in granular dynamics and fluid-solid interaction lead to large and challenging numerical problems that stand to benefit from parallel computing. Section 3 discusses parallel computing issues: Is parallel computing a fad? What parallel computing avenues should one adopt in the context of MBD modeling and simulation? and finally, What are the tools of the trade (libraries, debuggers, profilers, integrated development environments) that one can use to leverage parallel computing in MBD? Section 4 concentrates on issues that can influence the choice of (a) modeling formulation; (b) numerical methods; and (c) the parallel computing platform, three factors that combine to provide the foundation of a parallel MBD simulation tool. Section 5 describes three applications that relied on different parallel computing platforms to increase problem size and/or reduce time to solution. Section 6 rounds off the paper with closing remarks and several recommendations.

Note that the discussion herein was predicated on the assumption that the MBD problem size is large to the extent that the overhead associated with using high level parallel programming approaches is negligible. Approaches suitable for real-time simulation, when the interest is in characterizing on a tight and predefined schedule the dynamics of small problems, would revolve around programming techniques drawing on low level parallel libraries: e.g., pthreads. These libraries often work within the context of a dedicated and stripped down version of an operating system running on a multicore CPU chip. This discussion; i.e., parallel computing for real-time simulation, falls outside the scope of this contribution.

2 MBD: The Need for Parallel Computing.

MBD simulation plays an important role in the overall product life cycle by providing early insights in engineering designs through virtual prototyping. The complexity of the models in MBD simulation has evolved from component to subsystem to system. Recent advances in processor performance and memory speed/size, combined with the widespread availability of parallel computing platforms, provide renewed interest in what might be called many-body dynamics: understanding through direct numerical simulation the emergent behavior of large systems of 3D bodies interacting through friction, contact, and potentially long range electrostatic forces. Similar to Molecular Dynamics, the macroscale dynamics represents the emergent behavior of millions of 3D bodies whose motion is monitored in time. A list of applications that rely on many-body dynamics includes: packing of polyhedra for crystalline pattern formation studies; packing of compounds in pharmaceutical pills; drug delivery for powders and suspensions; ground vehicle mobility on granular terrain; additive manufacturing, in which layer upon layer of granular material is moved, shaped, and melted by a 3D printer; granular impact cratering of relevance in astrophysics, etc.

At its core, the many-body dynamics problem is posed like the typical MBD problem. The equations of motion are formulated herein using an absolute, or Cartesian, representation of the attitude of each rigid body in the system. The state of the system is represented by the generalized positions \( \mathbf{q} = [\mathbf{r}_1^T, \mathbf{r}_2^T, \ldots, \mathbf{r}_n^T, \bar{\omega}_1^T, \bar{\omega}_2^T] \in \mathbb{R}^{7n_b} \) and their time derivatives \( \dot{\mathbf{q}} \), where \( n_b \) is the number of bodies, \( \dot{\mathbf{r}}_j \in \mathbb{R}^3 \) is the absolute location of the center of mass of body \( j \) and the quaternions (Euler parameters) \( \epsilon_j \in \mathbb{R}^4 \) are used to represent body orientation.

Bilateral constraints representing kinematic joints (spherical, prismatic, revolute, etc.) are included as algebraic equations restricting the relative position of two rigid bodies. Each constraint \( i \in \mathcal{B} \in \mathcal{B} \), the set of bilateral constraints, is represented by the scalar algebraic equation \( \Psi_i(\mathbf{q}, t) = 0 \) and transmits reaction forces to the connected bodies by means of a multiplier \( \gamma_{i,b} \).

Assuming smoothness of the constraint manifold, \( \Psi_i(\mathbf{q}, t) \) can be differentiable to obtain the Jacobian \( \nabla_{\mathbf{q}} \Psi_i = \left[ \frac{\partial \Psi_i}{\partial \mathbf{q}} \right]^T \). In what follows we will also use the notation \( \nabla \Psi_i = L(\mathbf{q}) \cdot \nabla_{\mathbf{q}} \Psi_i \).

Contact and friction forces are included in the system through complementarity constraints. Given a large number of rigid bodies with different shapes, modern collision detection algorithms are able to efficiently find a set of contact points, that is, points where a gap function \( \Phi(\mathbf{q}) \) can be defined for each pair of near-enough geometry features. Where defined, such a gap function must satisfy the nonpenetration condition \( \Phi(\mathbf{q}) \geq 0 \) for all contact points. More precisely, when a contact \( i \) is active \( (\Phi_i(\mathbf{q}) = 0) \), a normal force and a tangential friction force act on each of the two bodies at the contact point; if the contact is not active \( (\Phi_i(\mathbf{q}) > 0) \), no contact or friction forces exist. This results in a mathematical description of the model known as a complementarity problem [75]. In what follows, \( A(\mathbf{q}(t)) \)
where \( \mu \) is the friction coefficient associated with contact \( i \).

The time evolution of the dynamical system is governed by the following differential problem with set-valued functions and complementarity constraints, which is equivalent to a differential variational inequality (DVI) [61]:

\[
\dot{q} = L(q)v
\]
\[
M\dot{v} = f(t, q, v) + \sum_{i \in B} \hat{\gamma}_{i,b} \nabla \Psi_i + \sum_{i \in A} (\hat{\gamma}_{i,n} D_{i,n} + \hat{\gamma}_{i,u} D_{i,u} + \hat{\gamma}_{i,w} D_{i,w})
\]
\[
i \in B : \Psi_i(q, t) = 0
\]
\[
i \in A(q(t)) : 0 \leq \hat{\gamma}_{i,n} \perp \Phi_i(q) \geq 0, \quad \text{and}
\]
\[
(\hat{\gamma}_{i,u}, \hat{\gamma}_{i,w}) = \arg \min_{\sqrt{\hat{\gamma}_{i,u}^2 + \hat{\gamma}_{i,w}^2} \leq \mu_i \hat{\gamma}_{i,n}} v^T (\hat{\gamma}_{i,u} D_{i,u} + \hat{\gamma}_{i,w} D_{i,w})
\]

where \( f(t, q, v) \) denotes the external (applied) generalized forces and the notation \( \perp \) is used to indicate orthogonality, i.e., \( \hat{\gamma}_{i,n} \Phi_i(q) = 0 \).

The tangent space generators \( D_i = [D_{i,n}, D_{i,u}, D_{i,w}] \in \mathbb{R}^{6n_i \times 3} \) are defined as

\[
D_i^T = [0 \cdots \hat{A}_i^T \hat{A}_A \hat{s}_{i,A} \cdots \hat{A}_i^T - \hat{A}_i^T \hat{A}_B \hat{s}_{i,B} \cdots 0]
\]

where \( \hat{A}_i = [n_i, u_i, w_i] \in \mathbb{R}^{3 \times 3} \) is the orientation matrix associated with contact \( i \), \( A_A = \hat{A}(\epsilon_A) \) and \( A_B = \hat{A}(\epsilon_B) \) are the rotation matrices of bodies \( A \) and \( B \), respectively, and the vectors \( \hat{s}_{i,A} \) and \( \hat{s}_{i,B} \) represent the contact point positions in body-relative coordinates as illustrated in Fig. 1. The mass matrix \( M \) is constant and diagonal due to the choice of body-fixed centroidal reference frames.

Under numerical discretization, see for instance [3, 27, 28, 76], the numerical solution calls at each time step for solving a quadratic optimization problem with conic constraints:

\[
\min q(\gamma) = \frac{1}{2} \gamma^T N \gamma + r^T \gamma
\]

subject to \( \sqrt{\gamma_{i,n}^2 + \gamma_{i,w}^2} \leq \mu_i \gamma_{i,n}, \quad \forall i \in A \),

where \( N \) and \( r \) are a constant symmetric positive semidefinite matrix and a constant vector that depend on system state at the beginning of the time step; \( \gamma = [\gamma_1^T, \gamma_2^T, \ldots, \gamma_n^T]^T \) and \( \gamma_i = [\gamma_{i,n}, \gamma_{i,u}, \gamma_{i,w}]^T = h [\gamma_{i,n}, \gamma_{i,u}, \gamma_{i,w}]^T \) is the triplet expressing the magnitude of the contact impulses for contact \( i \), with \( h \) denoting the integration step-size. This constrained optimization problem is solved by an iterative method or interior point method [27]. In either case, the dimension of the problem is large. As an example, the penetration study illustrated in Fig. 2 used 0.6 million bodies [27]. There are approximately two million contacts at any time step, each introducing a set of three unknowns \( \gamma_i \), resulting in an optimization problem of dimension around six million. Parallel computing is essential in solving this sub-problem and the collision detection that determines the set of contacts in the first place. For a typical integration step-size

\[
\text{FIGURE 1.} \quad \text{Contact } i \text{ between two bodies } A, B \in \{1, 2, \ldots, n_b\}
\]
FIGURE 2. Cross-section view of 3D normal contact forces in granular material (modeled using 0.6 million rigid bodies), during impact by spherical object.

$h = 10^{-4} s$, a number of 10,000 solutions of Eq. 4 and 10,000 collision detections are carried out for each second of simulation.

There are several alternative approaches to modeling frictional contact beyond the one presented above. The most widely used relies on a penalty/regularization approach: a small penetration is allowed between two bodies and the interface frictional-contact force is proportional to the amount of penetration and relative velocity of the bodies at the contact point. A recent review of literature on penalty methods is provided in [35]. While penalty methods do not call for the costly solution of the optimization problem in Eq. 4, they are hampered by much smaller integration time steps ($h \approx 10^{-6} s$). Like the DVI approach described above, penalty methods also have to deal with very large sets of differential equations resulting from the use of seven generalized positions and six generalized velocities for each 3D body in the system (almost 8 million equations for the problem in Fig. 2).

The Fluid-Solid Interaction (FSI) problem is very briefly discussed herein as another MBD application that leads to large sets of equations and whose solution requires complex numerical methods. The fluid flow can be represented in either an Eulerian or a Lagrangian framework. Provided that the interfacial forces are captured thoroughly, the Lagrangian framework is capable of tracking the domain deformation introduced by the motion of the solid phase at almost no extra cost. Smoothed Particle Hydrodynamics (SPH) [21, 40, 46], its modifications [15, 45], and variations [37] have been widely used for the simulation of the fluid domain in a Lagrangian framework. The main evolution equations of the fluid flow using SPH are expressed as

$$\frac{d\rho_a}{dt} = \rho_a \sum_b \frac{m_b}{\rho_b} (v_a - v_b) \cdot \nabla W_{ab}$$

$$\frac{dv_a}{dt} = - \sum_b m_b \left[ \left( \frac{p_b}{\rho_a^2} + \frac{p_a}{\rho_b^2} \right) \nabla W_{ab} - \frac{(\mu_a + \mu_b)r_{ab}}{\rho_a^2 ab (r_{ab}^2 + \varepsilon h_{ab}^2)} v_{ab} \right] ,$$

which are solved in conjunction with $dr_a/dt = v_a$ to update the fluid properties at each SPH marker $a$. Above, $\rho$, $\mu$, $v$, and $p$ are local fluid density, viscosity, velocity, and pressure, respectively, $m$ is the representative fluid mass assigned to the SPH marker, and $r_{ab}$ is the distance between two fluid markers $a$ and $b$. The kernel function $W_{ab} = W(r_{ab}, h)$ is used to smooth out the local fluid properties within a resolution length $h$. The fluid flow evolution equations are solved explicitly, where pressure is related to density via an appropriate state equation to maintain the compressibility below 1%. Accuracy and stability can be improved through an XSPH modification [45] and Shephard filtering [13].

FIGURE 3. Coupling of the fluid and solid phases. BCE and fluid markers are represented by black and white circles, respectively.

The fluid-solid coupling should satisfy the no-slip and impenetrability conditions on the surface of the solid obstacles. By attaching Boundary Condition Enforcing (BCE) markers on the surface of the solid objects (see Fig. 3), the local relative velocity between the two phases is enforced to be zero at the marker locations. The position and velocity of the BCE markers are updated according to the motion of the solid phase, resulting in the propagation of the solid motion to the fluid domain. In turn, the interaction forces evaluated at the BCE markers are used to calculate the total force and torque exerted by the fluid on the solid.
FSI applications lead to large problems owing to the number of SPH markers that need to be considered in the solution. The results reported in [64,65] drew on models with close to one million degrees of freedom and thus provide ample opportunities for parallel computing. For the specifics of a parallel implementation of the FSI solution, the interested reader is referred to [42].

3 Parallel Computing: the Long Term View, Alternatives, and Solution Development Support

For the purpose of this discussion, parallel computing refers to the process of enlisting multiple hardware assets (cores or processors) to simultaneously execute, in a fashion at least partially controlled by the application developer, either (a) multiple sequences of instructions of a program at the same time; or (b) the same sequence of instructions acting at the same time on multiple instances of data. Following Flynn’s taxonomy of parallel computer [20], the former covers multiple instruction multiple data (MIMD) architectures, while the latter refers to single instruction multiple data (SIMD) architectures. Herein, we concentrate on (i) distributed computing using the Message Passing Interface (MPI) [24], which falls into (a) above; (ii) general-purpose GPU computing (GPGPU), which is an example of (b) above; and (iii) symmetric multiprocessing (SMP) with OpenMP [59], which can fall under (a) or (b). Details about MPI, GPGPU, and OpenMP will be provided after discussing the grounds for a migration to parallel computing in CAE in general and MBD in particular. The section will conclude with a brief summary of productivity tools: compilers, debuggers, profilers, development environments, and libraries that support the three main parallel computing platform considered in this paper; i.e., GPGPU, OpenMP, and MPI.

3.1 Is Parallel Computing a Fad?

There were three factors that precipitated the rise of parallel computing in the mid 2000s. Also called the three walls to sequential computing, these factors marked a shift in the chip industry towards multi-core designs.

The memory wall is a manifestation of the fact that, for decades, memory speed increased at a pace significantly slower than that of data processing. For instance, from 1986 to 2000, the CPU speed improved at an annual rate of 55%, while the main memory access speed improved only at 10% annually [63]. It became apparent that making a faster and faster processor will only lead to scenarios where this fast processor would be idle waiting on data to be moved back and forth. Indeed, a memory request that leads to a cache miss and thus triggers a main memory (RAM) transfer typically requires hundreds of cycles. If the cache miss is combined with a translation look-aside buffer (TLB) miss, this cost can be orders of magnitude larger. Comparing this to the one cycle required nowadays to carry out an add-multiply operation, led to the assertion that computation is not about number crunching, which basically takes no time, but rather data transfer.

There are two aspects to memory speed: latency and bandwidth. Latency is essentially the amount of time it takes for a memory transaction to begin; once the transaction has begun, bandwidth is a measure of the rate at which data can be transferred from the provider of data to the consumer of data. Reducing latency is difficult; increasing bandwidth is relatively easy by adding more lanes in the memory bus, increasing the memory clock frequency, or using techniques that allow moving bits over several phases of the signal and/or wavelengths. In other words, there is little difference between moving 4 bytes or 512 bytes since the cost is not dictated by the bandwidth constraint as much as it is by the latency associated with the data movement request. As such, it became apparent that if data must be moved then it is advantageous to move it in big chunks. This favored parallel execution, especially in its SIMD flavor.

The power wall has manifested as a constraint placed on the clock frequency at which a chip could operate. It was postulated that the transistors used in computers "continue to function as voltage-controlled switches - while all key figures of merit such as layout density, operating speed, and energy efficiency improve, provided geometric dimensions, voltages, and doping concentrations are consistently scaled to maintain the same electric field” [14]. It thus makes sense to decrease both the typical feature size and voltage in a chip, since when these changes are combined with appropriate adjustments in doping concentrations the end product is superior: more transistors per unit area, which is essentially what fueled Moore’s law for the last four decades. However, there are physical limits that challenge this quest for smaller operation voltage and feature length. Presently, the voltage can only be marginally decreased – aiming for voltages below 0.3 V introduces risk because controlling the transistors becomes very hard in an electrically noisy environment. Decreasing the feature length is challenging too since the insulator that separates the gate from the rest of the MOS transistor becomes narrower as the feature length decreases. This translates into power leaks, which are important since power dissipation on a chip changes as follows: it is increased by the leaks, it increases quadratically with the voltage and linearly with the clock frequency at which the transistors are operated. As seen, lowering the voltage is not an option anymore and as such, as the feature length goes down, the frequency must be reduced or at best can stay constant. Otherwise, the amount of heat dissipated goes up, which can trigger a thermal runaway; i.e., a situation where an increase in temperature raises the resistance of the transistors leading to even more power being dissipated. The device either fails sooner or functions unreliably.

Finally, the instruction level parallelism (ILP) wall reflects the inability of parallel execution at the chip level to yield further substantial efficiency gains in sequential programming. In
other words, techniques that have benefited sequential computing in the past – instruction pipelining, superscalar execution, out-of-order execution, register renaming, branch prediction, and speculative execution – have fulfilled to a very large extent their potential. Interestingly, despite the use of the word parallelism, the above ILP techniques were all meant to speed up sequential computing. For instance, pipelining allows the simultaneous execution of several instructions at the same time; however, all these instructions belonged to one sequential stream of instructions, which now was executed faster since at each clock tick one instruction would be finished. The obvious mechanism of obtaining higher speedups by making the pipeline deeper is counterbalanced by the consequent rise in instruction dependency. Register forwarding, register renaming, and several other techniques eliminate some of these conflicts but pipeline stalls are inherent in deep pipelines. Superscalar solutions employ two parallel pipelines that nonetheless work on the same stream of instructions and support speculative execution, out-of-order execution, branch prediction, etc. All these tricks eventually reached a level where the complexity of managing them did not justify the return on investment in terms of power and/or die area costs.

In the early to mid 2000s it became apparent that the momentum behind the sequential computing in the server and consumer market was waning. Memory access latencies could not be hidden behind one stream of instructions and the gap between data processing and data movement was getting wider; the frequency couldn’t be increased any further without resorting to solutions that were more expensive than the market was accepting; and the ILP had for most part exhausted its set of tricks. Moore’s law remained one bright spot in the picture and provided a safe bet for continued performance improvement. Two of the leading chip manufacturers, Intel and IBM, had roadmaps in place that suggested that the feature length was bound to keep decreasing for close to two decades. For instance, Intel released Ivy Bridge in 2012, which is based on 22 nm technology. It will release a chip based on 14 nm technology next year and 10 nm technology in 2016, with 7 nm and 5 nm estimated for 2020 and 2022, respectively. This ongoing miniaturization trend sends a clear message that more and more cores will be available on a chip, making the parallel computing more and more prevalent.

To put things in perspective and understand the pace at which the technology advanced, less than one decade ago, a 1 TB distributed memory IBM Blue Gene/L with 1024 PowerPC cores used to deliver 5.7 TFlops. It came at a price of approximately $1.4 million, required a dedicated power source, and needed hardware and software support staff. Today, a powerful desktop with 1 TB of RAM, a motherboard with two Intel Xeon chips for 32 cores (64 virtual cores), and four Kepler GPU cards delivers more than 6 TFlops at a price of $70,000. It does not have steep power requirements, it does not require a system administrator to maintain it, can run Linux or Windows, and offers a comprehensive set of productivity tools. This trend towards more performance for lower costs is expected to continue in the same directions, namely more cores packaged on a chip and an increase in memory bandwidth allowing to hide memory latency with useful execution. In other words, the new parallel computing paradigm is anchored into technology realities.

### 3.2 Relevant Flavors of Parallel Computing

While the hardware transition to multi- and many-core chip layouts happened quickly, the software shift to using parallel computing languages and libraries to produce programs that leverage this hardware has been and continues to be arduous.

For fine grain parallelism that suits the SIMD model, GPU computing is the best match in terms of hardware and software support. The most recent GPU card from Nvidia belongs to the Kepler architecture. It has 250-320 GB/s bandwidth\(^1\) to feed about 2500 scalar processors, which can deliver up to 1.31 TFlops. This is approximately 25% of an IBM Blue Gene/L of one decade ago. The GPU’s weak spot is associated with its memory size. At 6-8 GB is in many cases sufficient but very large flexible body dynamics problems or fluid-solid interaction problems might run into memory space issues. These are usually resolved by shuttling data back and forth between the GPU and the host CPU. Yet this happens over a PCI Express bus that usually sustains 8 GB/s bandwidths and thus becomes a simulation bottleneck. In terms of software support, these GPUs are programmed using either OpenCL, which is an open standard [52], or CUDA [56]. The latter currently has wider adoption and counts on a more mature set of productivity tools.

In our experience, GPU computing turned out to be extremely suitable for simulations of granular material, as further discussed in section 5.3. Another application suitable for GPU computing is large scale flexible body dynamics where one has to handle tens to hundreds of thousands of elements. Such a problem was approached in the framework of Absolute Nodal Coordinate Formulation (ANCF) in [34]. Likewise, any analysis that involves a large set of data that needs to be processed is very likely amenable to GPU computing. For instance, FEA would fall into this category, since each node and/or element in the mesh should be visited in order to evaluate a stiffness matrix and, for implicit integration-based nonlinear FEA, to compute the ingredients used in assembling the numerical discretization Jacobian. Likewise, N-body problems, where one has to loop through all bodies to assess the influence of the neighboring bodies, provide a good setup for effective use of GPU computing.

GPU computing falters when dealing with problems with relatively small data sets or when the tasks performed are different for each member of the data set. Unfortunately, this turns out to be the case of classical MBD. Most systems include less than 500 rigid bodies. Even when one brings into the picture a floating frame of reference formulation for handling flexibility,

\(^1\)We use GB to denote gigabytes and Gb for gigabits. Eight Gb make one GB.
typically the size of the problem does not justify the use of GPU computing. Providing a size after which GPU makes sense depends on the type of GPU being considered. Today’s high end GPUs have 2,500 scalar processors or more. To get a good occupancy of the hardware, one would have to use in CUDA close to 30,000 threads to keep these 2,500 busy. In other words, one would need to deal with an MBD problem where there are at least 30,000 quantities that need to be computed. Moreover, since the GPU is a SIMD-type device, it is desirable to have these threads to work on very similar problems e.g., compute the stiffness matrix of 30,000 elements, or evaluate the normal force at 30,000 contact points, etc.

Parallel computing with OpenMP [59] is more forgiving in handling scenarios that deviate from the SIMD model. Unlike GPU computing, which for meaningful speedups requires problems that call upon the use of at least tens of thousands of parallel threads, OpenMP can yield good speed gains by using a relatively small number of threads. While GPU threads are lightweight and switching the hardware from the execution of one thread to a different thread incurs little overhead, OpenMP threads are heavy; starting or stopping a thread is costly. In fact, the operating system typically fires up a number of threads equal to the number of cores present on the system and manages these threads for the duration of the simulation by allocating jobs to them. The term system here refers to workstations that typically have between 4 and 64 cores that operate in a symmetric multiprocessor (SMP) framework. In this context, symmetric indicates that the main memory is shared and each of the identical cores has similar access to this memory albeit at slightly different latencies based on the physical proximity of a core to the memory bank accessed.

The OpenMP standard is supported very well by Intel and AMD chips. OpenMP can be presently used on a power workstation that might have 0.5 TB of RAM and 64 cores. Additional compute power can be accessed if the system has Intel Xeon Phi accelerators, each of which brings to the table more than 250 parallel threads. Unlike GPU computing, OpenMP (a) supports very well coarse grain parallelism, and (b) allows one to work in a more convenient fashion with very large data sets. Existing workstations routinely have hundreds of GB of main memory, which is two orders of magnitude more memory than one can access on the GPU. The latter can directly access the host main memory, but this comes at a stiff price as the transaction is carried out over the PCI express bus. The GPU wins when it comes to sheer memory bandwidth and flop rate. Number crunching is basically free as long as data is available; the problems crop up when cache misses require costly trips to main memory. GPU can do a better job here since (i) it operates at higher bandwidth (roughly 300 GB/s as opposed to roughly 50 GB/s for multicore CPUs); and (ii) it is better at hiding memory latency with useful work. The latter is accomplished by overcommitting the 2,500 scalar processors on the GPU to handle tens of thousands of threads to increases the chance that the scheduler will find a warp of threads that are ready for execution and thus avoid stalls of the scalar processors. This is very similar to the idea of hyper-threading except that while on the Intel architecture one can have two candidates for occupying the hardware, in GPU computing one warp scheduler has, on average, up to 16 choices from which it can select the next warp of threads that will be executed. This is because each stream multiprocessor on a GPU card can be assigned for execution up to 64 warps (groups of 32 threads) and it uses four warp schedulers.

In terms of source code modification, OpenMP provides a very low entry point into parallel computing. For instance, a for-loop that might evaluate the generalized forces acting on each body in the system can be parallelized by simply adding one line of code. Unfortunately, OpenMP does not scale well; i.e., it works on a powerful workstation but cannot be implemented for distributed memory architectures. Specifically, one cannot spread parallelism on two workstations solely using OpenMP.

Going beyond one workstation is typically achieved by using implementations of the Message Passing Interface (MPI) standard [24], such as MPICH2 [25] or Open MPI [23]. MPI will become indispensable when large amounts of memory are needed, such as for large granular dynamics or fluid-solid interaction problems. The MPI nodes are linked together through a fast interconnect for passing data back and forth. Typical bandwidths are of the order of 40 Gb/s; i.e., close to what one would get over a PCI connection. However, at 1 microsecond [53], the latency is significantly larger, which is likely to impact MBD simulation since dynamics simulation is communication bound. Specifically, at each time step, the nodes need to exchange data to advance the dynamics simulation. If, for instance, the step size in a penalty approach to handling frictional contact is $10^{-6}$ s, this means that each second of simulation requires 1,000,000 data exchanges. This places a significant bottleneck on the simulation essentially overshadowing any other aspect of the computation to the point where the execution time will be mostly spent in sending messages back and forth between the collaborating nodes and waiting for such messages to arrive.

### 3.3 Productivity Tools and Library Support

Today, a developer can write parallel software without writing any line of code that will execute directly on the GPU. There are several relatively mature libraries that provide a low entry point for GPU computing; the user only needs to make a call into such a library to obtain, for instance, the solution of a sparse linear system using an iterative method that will be executed on the GPU. CUSP [9] is an open source library of generic parallel algorithms for sparse matrix and graph computations. It provides several solvers for linear systems (conjugate-gradient, bi-conjugate gradient, biconjugate gradient stabilized, generalized minimum residual, multi-mass conjugate-gradient, and multi-
mass biconjugate gradient stabilized), preconditioners (algebraic multigrid based on smoothed aggregation, approximate inverse, diagonal), and several utilities related to manipulating dense and sparse matrices. Basic operations such as prefix scans, gather, scatter, sort, etc. are supported by the open source template library Thrust [30]. Operations that combine several arrays to produce an output array can be implemented by writing only a few lines of C++ code that are subsequently executed in an optimized fashion on the GPU.

As an alternative to using GPGPU libraries, a developer can implement functions (kernels) that are directly executed on the GPU. This is not a low entry point to GPGPU computing since writing efficient kernels requires a good understanding of the GPU’s deep memory hierarchy (registers, local memory, shared memory, constant memory, texture memory, global memory, cache levels) and how to set up an optimal kernel execution configuration, namely defining (i) the number of parallel threads in a block and (ii) the number of such blocks of threads launched in a grid. For instance, if a number of 0.5 million force computations are necessary, the user should specify how a number of at least 0.5 million threads will be launched to compute the forces in parallel. One can choose a large number of threads per block and a small number of blocks; alternatively, one can decide on a small number of threads but large number of blocks. These decisions are based on several factors, including the number of registers (per thread) used by the kernel and the amount of required shared memory (per block). Moreover, these execution configurations may have to change from GPU to GPU, based on hardware capabilities. The implementation effort for developing GPU kernels can be significantly reduced by available development tools. Nvidia’s Nsight development platform offers an integrated environment for building, debugging, profiling and tracing parallel applications using CUDA C/C++. Available both under Windows and Linux and integrated with Visual Studio and Eclipse, respectively, the Nsight debugger supports all common debugging tasks (setting breakpoints in CUDA source, viewing local variables and inspecting memory, navigate the current call stack, etc.) as well as switching the debugging context to any thread or warp of threads. The latter is critical for effective debugging of GPU parallel software which routinely involves launching tens of thousands of parallel threads. The analysis and profiling tools provided with Nsight help the developer understand overall performance and identify hot spots and bottlenecks. The user can see the combined CPU/GPU activity along a timeline or query performance information at a very fine level of granularity (e.g., level 1 and 2 cache hits, effective memory bandwidths and device utilization, register usage).

OpenMP enjoys a similar level of library and productivity tools support. The philosophy behind OpenMP, shared also by the language cilk and the Intel Threading Building Blocks TBB library is that the developer need only expose parallelism after which the runtime environment, through an internal scheduler, will manage the distribution of the work to parallel execution threads. Exposing parallelism can be done with compile-time directives (OpenMP), simple language extensions (cilk), or library calls (TBB). In terms of library support, the Intel Math Kernel Library (MKL) is a mature solution that leverages multi-core chips for fast math operations, including the solution of sparse, dense, and banded linear systems. Preconditioners are available, as are iterative solvers. Finally, debugging and profiling OpenMP in particular and any multi-core code in general is facilitated by a series of productivity tools released by Intel: Intel VTune Amplifier, Intel Inspector, and Intel Advisor. The tools can be used for debugging, code profiling, and code analysis.

Distributed (message passing) computing remains the privilege of a relatively small number of organizations. This is reflected in the relatively low number of options available for MPI distributed computing, of which two solutions stand out: OpenMPI [58] and MPICH2 [48]. The latter implements MPI-3, the most recent version of the standard, and is the origin of several other vendor implementations tuned to specific hardware; Intel, IBM, and Cray each have their own MPI derivative based on MPICH2.

Several open source libraries are available for large scale distributed simulation using MPI. LAMMPS [38, 66] provides functionality originally supporting molecular dynamics simulations. Over time, LAMMPS or derivatives have been used in peridynamics [62] or more recently in granular dynamics applications [39]. Unlike LAMMPS, which focuses on many-body dynamics, PETSc [7] is an infrastructure that leverages MPI parallel computing for scientific applications modeled with partial differential equations. It provides extensive support for linear algebra, such as algebraic multigrid, direct solvers, iterative solvers, and preconditioners. It offers explicit and implicit integrators, including an interface to the multistep implicit integrator CVODE from the Sundials suite [29].

4 Parallel Computing in MBD
4.1 Equation Formulation Aspects

Broadly speaking, multibody formulations based on the Newton-Euler equations can be categorized into Cartesian (or descriptor form) or recursive formulations. The Cartesian formulation [26,60,73,74] gained broader acceptance early on as it was used in commercial packages such as ADAMS and DADS. In such an approach, the equations of motion can be formed in a straightforward manner, as illustrated in Section 2, simply by writing the free body differential equations for all bodies in the system and append the algebraic (nonlinear) equations associated with each joint and constraint in the system.

A variety of recursive formulations have been proposed beginning with the early 1980s, first in the robotics community for chain manipulators and open-loop systems [4, 17] and later on extended to more general multibody systems, including systems
with closed-loop kinematic chains. Recursive formulations are inherently \( O(n) \) for the inverse dynamics problem, where \( n \) represents the number of degrees of freedom. For the more complicated forward dynamics problem, various formulations have been proposed which, based on how they deal with the factorization of the resulting generalized mass matrix, can be classified as \( O(n^3) \), \( O(n^2) \), or \( O(n) \) and arise successively as a “natural consequence of a progressive exploitation of the structure of the mass matrix” as noted by Jain in [33]. Jain also showed that, except for minor differences, all linear complexity recursive algorithms, i.e., those that result in explicit derivations of the inverse of the mass matrix [5, 6, 11, 17, 68–70], are essentially the same algorithm. The equations of motion, expressed in terms of a minimal set of generalized states, are formed implicitly through successive outward (base-to-tips) and inward (tips-to-base) sweeps of the underlying multibody tree to recursively propagate kinematics and articulated inertias, respectively.

More recent divide and conquer formulations [18, 19, 51] achieve \( O(n) \) complexity for serial implementation but, due to their underlying binary tree structure, have logarithmic complexity when implemented in parallel.

Our goal here is neither to provide an exhaustive review of previous work in this area nor to advocate one choice over another. Instead, we will briefly comment on how they relate to the use of parallel computing in MBD. The differences in terms of type, structure, and size of the resulting equations in the two formulations can be summarized as follows:

**Type** For any non-trivial multibody system, the Cartesian formulation always results in a set of differential algebraic equations (DAEs).

On the other hand, recursive formulations result in systems of ordinary differential equations (ODEs) for open-loop systems and minimal-size DAEs for closed-loop system (using for example the cut joint approach [78]).

**Structure** The Cartesian formulation does not take into account the structure (topology) of the MBD, at least not at the equation formulation phase, but results in a set of equations with a very well defined and, most importantly, simple structure. Addition/removal of bodies or constraints is a trivial task.

On the other hand, recursive formulations effectively impose certain computational paths (namely outward/inward recursive traversals of the topological tree) which result in much smaller but more complex, especially for \( O(n) \) formulations, system of equations. Addition or removal of bodies or constraints is a more complex task as it requires a re-evaluation of the system’s topology.

**Size** There is a significant disparity in the size of the resulting DAE or ODE systems for the two main classes of formulations considered here. For a system consisting of \( n_b \) bodies and having \( n \) degrees of freedom, a Cartesian formulation results in a DAE with \( 6n_b \) and \( 6n_b - n \) differential and algebraic equations, respectively. If the system does not contain any kinematic closed loops, a recursive formulation leads to a system of \( n \) ODEs. However, in the presence of closed loops, the \( m \) constraints introduced in the process of extracting the topological spanning tree lead to larger-size (namely \( 2n + m \)) DAE systems. Note that if the multibody system contains many loops, such that \( m = O(n) \), in addition to the increase in problem size, one must also re-evaluate the algorithmic complexity of recursive formulations since \( O(n) \) algorithms actually perform as \( O(n + nm^2) \).

From the above observations it becomes apparent that there is significant potential for fine grain parallelism within the Cartesian framework. The fashion in which the equations of motion are generated and assembled, particularly the fact that constraints induced by a joint only involve states associated with the bodies connected by that joint, exposes a large number of independent/decoupled calculations. Hence, large problems posed in the Cartesian framework fit well the SIMD model supported by GPGPU computing. On the other hand, the problem size and the computational flow imposed by recursive formulations favor task parallelism that mirrors the structure of the spanning tree. OpenMP represents a good parallel computing platform with the caveat that attention needs to be paid to load balancing, which is influenced by the spanning tree associated with the system: vastly different branch sizes hurt load balancing as they’ll require different levels of traversing effort.

### 4.2 Numerical Solution Aspects

The numerical solution of dynamics problems involving frictional contact in a DVI formulation (such as that in Eq. 2) require new numerical approaches as one moves towards very large-scale many-body problems involving hundreds of thousands to millions of bodies and contacts. Classical approaches based on posing and solving linear complementarity problems (LCP) obtained as a result of polyhedral approximations of the friction cone not only suffer from accuracy issues stemming from the induced anisotropies but also lead to an increase in problem size that quickly becomes prohibitive. These limitations can be avoided by transforming the original nonlinear complementarity problem into a cone complementarity problem (CCP) which can be shown [3, 76] to represent the first-order optimality conditions of the constrained optimization problem of Eq. 4. Usual approaches to solving this problem rely on fixed-point iterations (Jacobi or Gauss-Seidel) with projection on a convex set; however, since these methods require a large number of iterations when handling large systems of engineering relevance, the CCP solution, which can account for up to two thirds of the total solution time, leads to prohibitively long simulations. Moreover, some of these methods (e.g. Gauss-Seidel) are not amenable to parallel computing. Alternative solvers, such as gradient projected minimum residual (for frictionless problems), precondi-
tioned spectral projected gradient with fallback, and primal-dual interior point methods, have been shown to have improved convergence properties leading to significant speedups [27, 28].

The efficiency of MBD simulations is directly influenced by the size of the integration step-size. Large step sizes are desirable especially in many-body problems with frictional contact, regardless of the modeling approach (DVI or DEM), since each time step requires a costly collision detection stage that can involve millions of contact events. As discussed in §2, the existing integration schemes require typical step sizes of $h = 10^{-4}$ for a DVI approach and as low as $h = 10^{-6}$ for DEM. In our implementation, each step of the DVI uses half-implicit numerical integration that calls for the solution of a large optimization problem. It quickly becomes apparent that the promise of large step-sizes associated with fully implicit methods is very attractive, since in this case the step size is unrestricted by stability considerations. However, fully implicit integration schemes lead to nonlinear systems typically solved using Newton-like methods and hence they require (i) the ability to evaluate Jacobian information and (ii) efficient linear solvers. Note that linear solvers are important not only for implicit integration, but also for interior point methods used to solve the CCP associated with the problem in Eq. 4. Even though very sparse, the sheer size of the resulting matrices in problems such as granular dynamics or Lagrangian formulations for fluid-solid interaction (e.g., SPH) make direct methods for linear systems less attractive. The alternative is to use matrix-free iterative techniques, such as the Krylov subspace iterative methods, which do not require explicitly forming the system’s matrix but only matrix-vector products [34]. Unfortunately, for fast convergence these iterative methods still require the use of adequate preconditioners. While domain-specific preconditioners for MBD problems are still lacking, general-purpose preconditioners are available for all parallel computing platforms discussed herein: PETSc [7] and hypre [16] libraries for MPI computing, Intel’s MKL for OpenMP, and CUSP [9] for GPGPU.

4.3 Software Development Aspects

The parallel computing of mid 1990s relied extensively on posix threads [57]. A very low level approach to parallel computing, posix threads are currently used only when performance arguments trump the development and maintenance effort associated with this parallel programming approach. Today, alternative solutions provide a parallel programming entry point in which the technical effort is focused more on modeling and numerical solution aspects and less on the implementation constructs required by these programming platforms/models.

While it is clear that when measured by the amount of data that needs to be processed during simulation very large models will always fall back on a distributed computing model supported by the MPI standard, simulations that fit on one workstation can be run either with OpenMP or relying on GPGPU computing. Debating which of the two platforms is superior falls outside the scope of this paper. However, for task parallelism, where several independent and sizable tasks need to be performed, OpenMP has intrinsic support that will give it an edge. For instance, for a typical multibody dynamics problem with 300 bodies, 100 joints, and 200 forces, OpenMP parallelizes the evaluation of the Jacobian associated with implicit integration very naturally: one task will be associated with the computation of the contribution of the 300 bodies, another task will be associated with the evaluation of the contribution of the 100 joints, and yet a third task will handle the contribution of the 200 forces. Three OpenMP tasks can be launched, each one of them with an inner loop that can be parallelized, and native dynamic load balancing would distribute the load and keep all cores busy. Here GPGPU computing is less attractive since the problem and associated coarse grain parallelism does not fit the SIMD model.

For many-body dynamics though, such as granular dynamics simulation, mobility on discrete terrain, fluid-solid interaction using SPH, etc., where there are tens of thousands of data instances to be processed, GPGPU is very attractive. Collision detection, evaluating the force at each contact point, or advancing the state of the system through explicit integration are computations that map well on GPUs owing to the SIMD nature of the job at hand. It is this type of fine grain parallelism; i.e., very large number of identical operations that need to be performed on a large number of items, that is the forte of GPGPU computing.

There is a gray area with no clear answer to whether OpenMP or GPU computing has the upper hand. In this case we implemented a coding style that allowed us to experiment with either platform by setting an environment variable at compile time. To support this, the code will have to be modified in a very limited and localized fashion. Specifically, both for data structures and function definitions a different syntax is needed depending on whether GPU or CPU computing is used. As an example, consider the code in the following listings. The purpose of this code is to update the velocity of each body given the force acting on it. If GPU computing is used, this should be performed with a kernel with one thread per body. If CPU computing is used, the velocity update would be achieved by looping over all bodies in the system using OpenMP through the inclusion of a simple #pragma omp parallel for.

First, a function is written to perform the desired operation. In this case, the function should increment the velocity of a given body based on the mass, inertia, force, torque, and time step. Note that this function is defined with _host_ _device_, which means it is compiled to be executed on both the CPU (host) and GPU (device).
Listing 1. Function to update the velocity of a body

```c
__host__ __device__
void addForces(uint& i, real inv_mass, real3 forces, real3 vel) {
  vel[i] += forces[i] * inv_mass;
}
```

The above function, in which the value of the step size \( h \) has already been factored in the inverse of the mass matrix, should be called in different ways if CPU or GPU is used. If the GPU is used, a GPU kernel (shown in the listing below) should call the function. Note the inclusion of __global__, which signifies a kernel function, and the macro INIT_CHECK_THREAD_BOUNDED, which ensures that the thread IDs executing the kernel are appropriate.

Listing 2. GPU kernel to update the velocity of all bodies in the system

```c
__global__
void device_addForces(real mass, real3 forces, real3 vel) {
  INIT_CHECK_THREAD_BOUNDED(INDEX1D, num_obj, const);
  addForces(index, mass, forces, vel);
}
```

For parallel execution on the CPU, the code should loop over all bodies in the system using an OpenMP for-loop, invoking the single-body update function as in the following listing.

Listing 3. Host loop to update the velocity of all bodies in the system

```c
void host_addForces(real mass, real3 forces, real3 vel) {
  #pragma omp parallel for
  for (uint index = 0; index < num_obj; index++) {
    addForces(index, mass, forces, vel);
  }
}
```

Finally, a compile-time flag, ENABLE_GPU, is used to control which parallel computing model is used. If the flag is defined, the GPU function is called; otherwise, the host function is called.

Listing 4. Velocity update, selecting between CPU and GPU execution

```c
#if defined ENABLE_GPU
COPY_TO_CONST_MEM(num_obj);
device_addForces <<<BLOCKS(num_obj), THREADS>>>(
  device_mass_data,
  device_force_data,
  device_vel_data);
#else
host_addForces(
  device_mass_data.data(),
  device_force_data.data(),
  device_vel_data.data());
#endif
```

This strategy is used for all code sections in which parallel computing can be leveraged. A single function is written to perform the necessary operations, capable of executing on the CPU and GPU (Listing 1). One wrapper function is written for the GPU, where a kernel calls the function within each thread (Listing 2), while a second wrapper is written for the CPU, where the function is called from within an OpenMP for-loop (Listing 3). Finally, the compile-time flag is used to selectively compile the appropriate wrapper function in the code (Listing 4).

Note that primitive functions are handled by Thrust directly. For example, sorting a vector can be accomplished by calling `thrust::sort(myVec.begin(), myVec.end());` If `myVec` is a host vector, without any user intervention, Thrust will use OpenMP to sort it. If `myVec` is a device vector, Thrust will use the GPU.

5 Examples: GPGPU, OpenMP, and MPI Parallel Computing in MBD

Three applications are discussed next to showcase the use of the parallel computing techniques discussed above. GPU computing is used in conjunction with the dynamics of a tracked vehicle operating on granular terrain. An OpenMP-enabled granular dynamics simulation is used to identify pattern formation in granular material that is shaken at a critical frequency. Finally, MPI-enabled distributed computing is used to investigate the dynamics of a wheeled vehicle operating on discrete deformable terrain. These models were run in Chrono, an open source parallel simulation framework for computational dynamics [42,72].

5.1 GPU Computing Example: Light Tracked Vehicle Mobility Simulation

A detailed account of how GPU computing is used to solve multibody dynamics problems can be found in [54]. The key kernels of the GPU implementation are listed in the following pseudocode:

Parallel Kernels for Solution of Dynamics Problem

1. Parallel Collision Detection
2. (Body parallel) Force kernel
3. (Contact parallel) Contact pre-processing kernel
4. Inner Iteration Loop:
   (a) (Contact parallel) CCP contact kernel
   (b) (Bilateral-Constraint parallel) CCP contact kernel
   (c) (Body parallel) Body velocity update kernel
5. (Body parallel) Time integration kernel

Using a combination of joints and linear actuators, a tracked vehicle model was simulated navigating over deformable terrain made up of gravel-type granular material. The vehicle is modeled to represent a small, lightweight tracked vehicle much like an autonomous robot.

There are two tracks, each with 61 track shoes (see Fig. 4). Each track shoe is made up of two cylinders and three rectangular plates and has a mass of 0.34 kg. Each shoe is connected to its neighbors using one revolute joint on each side. Within each
track there are five rollers, each with a mass of 15 kg, one idler and one sprocket both with a mass of 15 kg. The chassis has a mass of 200 kg and moments of inertia were computed for all parts using a CAD package. The purpose of the rollers is to keep the tracks separated and support the weight of the vehicle as it moves forward. The idler keeps the track tensioned. It is usually modeled with a linear spring/actuator but for the purposes of demonstration it was attached, using a revolute joint, to the vehicle chassis. The sprocket is used to drive the vehicle and is attached, using another revolute joint, to the chassis. Torque is applied to drive the track, with each track driven independently of the other.

The track for the vehicle was created by first generating a ring of connected track shoes. This ring was dropped onto a sprocket, five rollers, and an idler which was connected to the chassis using a linear spring. The idler was pushed with 2000 N of force until the track was tensioned and the idler had stopped moving. This pre-tensioned track was then saved to a data file and loaded for the simulation of the complete vehicle.

Figure 5 shows the joint forces for one revolute joint when the tracked vehicle was simulated as it moved over a bed of 84,000 granular particles. The particles were modeled as large pieces of gravel with a radius of 0.075 m, and a density of 1900 kg/m³. A torque of 100 Nm was applied to both sets of tracks to move the vehicle. Unlike the case where the vehicle moves on a flat section of ground, the forces experienced by the revolute joints are much noisier. Individual grains slip and stick under the tracks as the vehicle moves causing large vibrations to travel through the shoes. These vibrations would be reduced when modeling a more compliant terrain material that can dissipate energy on contact.

5.2 OpenMP Example: Pattern Formation in Granular Dynamics

This simulation was used to study the pattern formation in granular material shaken vertically. These patterns emerge in layers of material that is driven only by floor vibrations and have been studied extensively both in 2D and in 3D [10, 41, 44, 77]. This phenomenon is important in a variety of fields, from studying the formation of crystalline structures to processing and handling different types of granular material [32]. Without friction these patterns cannot emerge [47]. The simulation is challenging due to the number of particles involved and the velocity and amplitude of the vibrations of the moving plate.

The simulation was performed on a Intel Xeon E5-2630 with 6 physical cores and 12 virtual cores. The simulation in-
cluded 60,000 rigid particles, was advanced with a time step of $5 \times 10^{-4}$ s, and was used to investigate the system dynamics for 30 s. An OpenMP parallel version of the Accelerated Projected Gradient Descent solver was used with 50 iterations performed at each time step [27]. The parameters for friction, packing ratio, and vibration amplitude and frequency used were as provided in [10]. A friction value of $\mu = 0.5$ was chosen. The particle size, gravity and the total number of particles were used to compute the non-dimensional frequency $f^*$ and amplitude $\Gamma$ of the driving plate. Figure 6 shows a simulation with parameters $\Gamma = 0.4$ and $f^* = 0.38$. The results obtained are in line with the experimental data, which confirms that this choice of parameters produces a hexagonal pattern in the granular material [10].

5.3 MPI Example: Vehicle Mobility

When relying on MPI parallel computing, Chrono divides a simulation domain into a number of sub-domains that can each be simulated in parallel on discrete compute nodes. The solver transparently sets up the necessary communication channels between participating compute nodes and carries out the simulation including collision detection in a distributed and parallel fashion [43].

The general algorithmic flow of the domain decomposition approach for DEM simulations is given in the following pseudocode, which specifies the operations performed by a given node simulating a single sub-domain. All nodes operate independently, in parallel, subject to communication and synchronization barriers.

<table>
<thead>
<tr>
<th>PSEUDOCODE DEM MPI()</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1) Identify boundaries of sub-domain $S_t$</td>
</tr>
<tr>
<td>(2) Initialize communication pathways to each sub-domain $S_j$</td>
</tr>
<tr>
<td>(3) Initialize data structures</td>
</tr>
<tr>
<td>(4) for $t = 0$ to $t_{end}$</td>
</tr>
<tr>
<td>(5) Perform collision detection</td>
</tr>
<tr>
<td>(6) Compute collision forces</td>
</tr>
<tr>
<td>(7) Barrier: Intermediate communication of net forces on shared objects</td>
</tr>
<tr>
<td>(8) Compute net forces</td>
</tr>
<tr>
<td>(9) Perform integration to update state</td>
</tr>
<tr>
<td>(10) Barrier: Final synchronization</td>
</tr>
<tr>
<td>(11) Update set of bodies in sub-domain</td>
</tr>
<tr>
<td>(12) $t \leftarrow t + h$</td>
</tr>
<tr>
<td>(13) endfor</td>
</tr>
</tbody>
</table>

The example considered represents a replica of the Mars rover Opportunity. Details on the rover model, including inertia and geometry properties, can be found in [27]. The rover chassis was 100 kg; the mass of each wheel was 3 kg. The wheels were represented as cylinders with radius 10 cm and width 12.5 cm. A set of 30 grousers distributed evenly around the circumference of each wheel extend across the wheel’s width. Each grouser has a thickness of 0.5 cm and extends away from the wheel by 0.5 cm. The terrain was composed of 2,016,000 rigid spherical bodies with radius 5 mm and mass 4.5 g. The elements in the terrain were initialized in random positions above the simulation domain and allowed to settle under gravity. The simulation domain was rectangular, with length of 3 m in the driving direction and width of 1.5 m. All six wheels of the rover were driven with constant angular velocity of $\pi$ rad/s, or 30 rpm. The integration time step was $10^{-5}$ s and 6 s of dynamics were simulated, requiring about 298 hours of computation time. The DEM stiffness and damping parameters had the values $k_n = k_t = 2 \times 10^5$, $\gamma_n = 750$, and $\gamma_t = 0$. The simulation domain was partitioned into 64 sub-domains by dividing both the length and the width evenly into 8 sub-domains in each direction. The resulting sub-domains were rectangular, with length 0.375 m and width 0.1875 m. The depth direction was not divided.

A snapshot from the simulation can be seen in Fig. 7. The terrain particles are colored by sub-domain and shared bodies are colored white. Further, the wheels of the rover are checkered with the color of the master sub-domain. In Fig. 7 the wheels are pink, which indicates that the pink sub-domain has the master copy of the entire rover vehicle. Figure 8 shows the “footprint” of the rover. The terrain particles are colored uniformly, hiding the sub-domain division, and the rover and its wheels are not shown. The amount of simulation time is still prohibitively large for this problem, which aims at understanding how different vehicle topologies influence the dynamics of the rover. By orders of magnitude, the simulation bottleneck is associated with the data.
communication required by an exchange protocol that handled the cross-boundary motion and interaction of bodies.

6 Closing Remarks

Once the privilege of national research labs or supercomputing centers, parallel computing is now ubiquitous. In fact, it is virtually impossible today to purchase a workstation or laptop that doesn’t have at least four cores that can support parallel computing. These two observations frame the answer to the why in the title of this contribution: sequential computing promises only modest gains in efficiency, insufficient for fueling the evolution of the MBD field in the next decade. The continuous drive in MBD towards multi-physics problems yielding ever larger systems of equations provides the backdrop for the when in the title of the paper. Fluid-structure interaction, granular dynamics, and vehicle mobility on deformable discrete terrain represent example applications that will stand to benefit from the interplay between MBD and parallel computing. Finally, using as a justification several applications in which we have relied on different parallel computing platforms, we discussed aspects related to the modeling, numerical solution, and software implementation aspects of MBD simulation; i.e., we addressed the how aspect alluded to in the title of the contribution. Several comments made at various points in this paper are summarized as follows:

Parallel computing platform. Large models involving an amount of data that exceeds the memory available on a workstation require a distributed computing model facilitated by MPI parallelism. As discussed in §5.3, MPI incurs high communications cost that presently can limit duration or size of the problem being analyzed. This is particularly the case if the simulation proceeds at small step-sizes and data communication dwarfs any other component of the numerical solution. If the problem is small enough – tens to hundreds of GB on the CPU and up to 12 GB on the GPU – the approach described in §4.3 allows one to experiment with OpenMP or GPU computing and make a choice on a case-by-case basis. GPGPU computing is attractive for fine level granularity when the computational task fits the SIMD model. Problems characterized by task parallelism are ideally suited for OpenMP or related approaches (cilk, TBB).

Equation formulation. Recursive formulations do not have a level of granularity that fits well the SIMD model but are well suited for task-level parallelization. The finer degree of granularity associated with the Cartesian formulation makes it ideal for GPGPU computing.

Numerical solution. Minimizing time to solution may require adoption of numerical solution techniques different from those used for sequential calculations. This might require embracing strategies that are manifestly suboptimal in a sequential computing framework but which expose a high degree of parallelism and come with low synchronization demands. Moreover, effective parallel computing requires a shift in how we think about writing software. Rather than organizing data structures for the convenience of the programmer by reflecting the domain specific organization of a problem (bodies, joints, forces, etc.), the data structures should be organized to reflect the hardware constraints of the underlying architecture, which most often prefers data to be organized in long contiguous arrays that are suitable for parallel processing and ideal for vectorization. The latter approach though renders the software hard to design, debug, and maintain as it loses the connection to the domain problem that it solves.

We conclude with a few recommendations that coalesced during several years of experience with using parallel computing in MBD simulation.

1. The most important observation is that number crunching is basically free while data movement/memory transactions are one to two orders of magnitude more expensive. For memory transactions the most important aspect is memory access patterns. For GPGPU computing uncoalesced memory accesses reduce bandwidth, while for OpenMP violating spatial and temporal locality in data accesses increase the likelihood of cache misses. The recommendation is at the software design stage to organize model data in structures that are poised to minimize the amount of traffic to main memory or inter-core communication (see the “Numerical solution” remark above).

2. If available, use domain libraries: the solution is already there, bug-free and optimized. Reinventing the wheel may be useful for learning purposes but is unlikely to lead to performance improvements.

3. Debugging parallel code is challenging. Although involving
a steep learning curve, using debuggers is the only way to access program state information that cannot be replicated by rudimentary debugging techniques (such as using print statements). NVIDIA’s cuda-gdb for GPU computing, Intel’s Parallel Studio Suite for OpenMP computing, and Intel’s Cluster Studio for MPI computing represent debugging solutions widely used in the developer community.

4. Use dedicated profiling tools as the only way to conclusively find the hot spots and bottlenecks in a parallel program and then direct the optimization effort where it matter most. In this context, the recommendation is to always keep in mind Amdahl’s law [1] and parallelize the portions of the code that are the actual computational bottlenecks. Finally, recall that premature optimization is the root of all evil [36].

Acknowledgements

The hardware used to produce the results reported in this paper was procured through the Army Research Office instrumentation grant W911NF-11-1-0327. We thank NVIDIA for sponsoring our research programs in the area of high performance computing. We are grateful to Alex Dirr, Prateek Gupta, and Arindam Sinha for their feedback on a draft of this document.

REFERENCES


[22] F. González, A. Luaces, U. Lufrís, and M. González. Non-


