Chapter 9

Crack Propagation Analysis with Automatic Load Balancing

Orion Sky Lawlor

Department of Computer Science, University of Alaska Fairbanks

M. Scot Breitenfeld and Philippe H. Geubelle

Department of Aerospace Engineering, University of Illinois at Urbana-Champaign

Gengbin Zheng

National Center for Supercomputing Applications, University of Illinois at Urbana-Champaign

9.1 Introduction

Researchers in the field of structural mechanics often turn to the parallel finite element method to model physical phenomena with finer detail, sophistication and accuracy. While parallel computing can provide large amounts of computational power, developing parallel software requires substantial effort to exploit this power efficiently.

A wide variety of applications involve explicit computations on unstructured grids. One of the key objectives of this work is to create a flexible framework to perform these types of simulations on parallel computing platforms. Parallel programming introduces several complications:

- Simply expressing a computation in parallel requires the use of either a specialized language such as HPF [134] or an additional library such as MPI.
- Parallel execution makes race conditions and nondeterministic execution possible. Some languages, such as HPF, have a simple lockstep control structure and are thus relatively immune to this problem. However, in other implementations, such as asynchronous MPI or threads, these issues are more common.
- Computation and communication must be overlapped to achieve optimal performance. However, few languages provide good support for this overlap, and even simple static schemes can be painfully difficult to implement in the general case, such as overlapping computation from one library with communication from a different library.
- Load imbalance can severely restrict performance, especially for dynamic applications. Automatic or application-independent load balancing capabilities are rare (Section 9.2.2).

Our approach to managing the complexity of parallel programming is based on a simple division of labor. In this approach, parallel programming specialists in computer science provide a simple but efficient *parallel framework* for the computation, while application specialists provide the numerics and physics. The parallel framework described hereafter abstracts away the details of its parallel implementation.

Since the parallel framework is application independent, it can be reused across multiple codes. This reuse amortizes the effort and time spent developing the framework and makes it feasible to invest in sophisticated capabilities such as adaptive computation and communication overlap and automatic measurement-based load balancing. Overall, this approach has proven quite effective, leveraging skills in both computer science and engineering to solve problems neither could solve independently.

9.1.1 ParFUM Framework

ParFUM [144] is a parallel framework for performing explicit computations on unstructured grids. The framework has been used for finite-element computations, solving partial differential equations, computational fluid dynamics and other problems.

The basic abstraction provided is very simple—the computational domain consists of an irregular mesh of nodes and elements. The elements are divided into partitions or chunks, normally using the graph partitioning library METIS [72], or ParMETIS [71]. These chunks reside in migratable AMPI virtual processors, thereby taking advantage of runtime optimizations including dynamic load balancing. The chunks of meshes and AMPI virtual processors are then distributed across the processors of the parallel machine. There is typically at least one chunk per processor, but usually many more. Mesh nodes can be either private, adjacent to the elements of a single partition, or shared, adjacent to the elements of different partitions.

A ParFUM application has two main subroutines: *init* and *driver*. The *init* subroutine executes only on processor 0 and is used to read the input mesh and physical data and register it with the framework. The framework then partitions the mesh into as many regions as requested, each partition being a virtual processor. It then executes the *driver* routine on each virtual processor. This routine computes the solution over the local partition of the mesh.

The solution loop for most applications involves a calculation in which each mesh node or element requires data from its neighboring entities. Thus entities on the boundary of a partition need data from entities on other partitions. ParFUM provides a flexible and scalable approach to meet an application's communication requirements. ParFUM adds local read-only copies of remote entities to the partition boundary. These read-only copies are referred to as *ghosts*. A single collective call to ParFUM allows the user to update all ghost mesh entities with data from the original copies on neighboring partitions. This lets application code have effortless access to data from neighboring entities on other partitions. Since the definition of "neighboring" can vary from one application to another, ParFUM provides a flexible mechanism for generating ghost layers. For example, an application might consider two tetrahedra that share a face as neighbors. In another application, tetrahedra that share edges might be considered neighbors. ParFUM users can specify the type of ghost layer required by defining the "neighboring" relationship in the *init* routine and adding multiple layers of ghosts according to the neighboring relationship for applications that require them. In addition, the definition of "neighboring" can vary for different layers. User-specified ghost layers are automatically added after partitioning the input mesh provided during the *init* routine. ParFUM also updates the connectivity and adjacency information of a partition's entities to reflect the additional layers of ghosts. Thus ParFUM satisfies the communication needs of a wide range of applications by allowing the user to add arbitrary ghost layers. After the communication for ghost layers, each local partition is nearly self-contained; a serial numerics routine can be run on the partition with only a minor modification to the boundary conditions.

With the above design, the ParFUM framework enables straightforward conversion of serial codes into parallel applications. For example, in an explicit structural dynamics computation, each iteration of the time loop has the following structure:

1. Compute element strains based on nodal displacements.

2. Compute element stresses based on element strains.
3. Compute nodal forces based on element stresses.
4. Apply external boundary conditions.
5. Update nodal accelerations, velocities and displacements based on Newtonian physics.

In a serial code, these operations apply over the entire mesh. However, since each operation is local, depending only on a node or element's immediate neighbors, we can partition the mesh and run the same code on each partition.

The only problem is ensuring that the boundary conditions of the different partitions match. We might choose to duplicate the nodes along the boundary and then sum up the nodal forces during step 3, which amounts to the simple change in step 4 to: Apply external *and internal* boundary conditions.

For existing codes that have already been parallelized with MPI, the conversion to ParFUM is even faster, thereby taking advantage of Charm++ features, such as dynamic load balancing.

9.1.2 Implementation of the ParFUM Framework

As shown in the software architecture diagram of Figure 9.1, our parallel FEM framework is written and parallelized using Adaptive MPI. Adaptive MPI (AMPI) [106, 105] is an MPI implementation and extension based on the Charm++ [117] programming model. Charm++ is a parallel C++ runtime system that embodies the concept of *processor virtualization* [122]. The idea behind processor virtualization is that the programmer decomposes the computation, without regard to the physical number of processors available, into a large number of logical work units and data units, which are encapsulated in *virtual processors* (VPs) [122]. The programmer leaves the assignment of VPs to physical processors to the runtime system, which incorporates intelligent optimization strategies and automatic runtime adaptation. These virtual processors themselves can be programmed using any programming paradigm. They can be organized as indexed collections of C++ objects that interact via asynchronous method invocations, as in Charm++ [147]. Alternatively, they can be MPI virtual processors implemented as user-level, lightweight threads (not to be confused with system-level threads or Pthreads) that interact with each other via messages, as in AMPI (illustrated in Figure 9.2).

This idea of processor virtualization brings significant benefits to both parallel programming productivity and parallel performance [123]. It empowers the runtime system to incorporate intelligent optimization strategies and automatic runtime adaptation. The following is a list of the benefits of processor virtualization.

Automatic load balancing: AMPI threads (the virtual processors) are decoupled from real processors. Therefore, they are location independent and

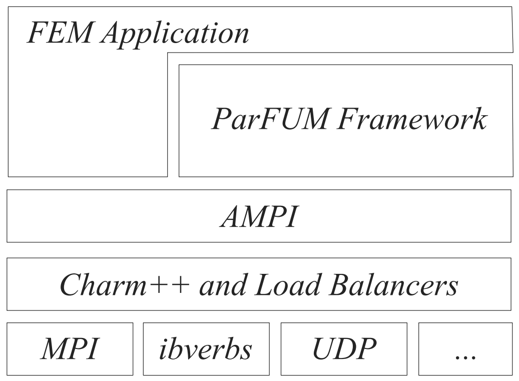


FIGURE 9.1: Software layer diagram for finite element codes in Charm++.

can migrate from processor to processor. Thread migration provides the basic mechanism for load balancing: if some of the physical processors become overloaded, the runtime system can migrate a few of their AMPI threads to underloaded physical processors. The AMPI runtime system provides transparent support of message forwarding after thread migration.

Adaptive overlapping of communication and computation: If one of the AMPI threads is blocked on a receive, another AMPI thread on the same physical processor can run. This largely eliminates the need for the programmer to manually specify some static computation/communication overlapping, as is often required in MPI.

Optimized communication library support: Besides the communication optimization inherited from Charm++, AMPI supports asynchronous or nonblocking interfaces to collective communication operations. This allows the overlapping between time-consuming collective operations and other useful computation.

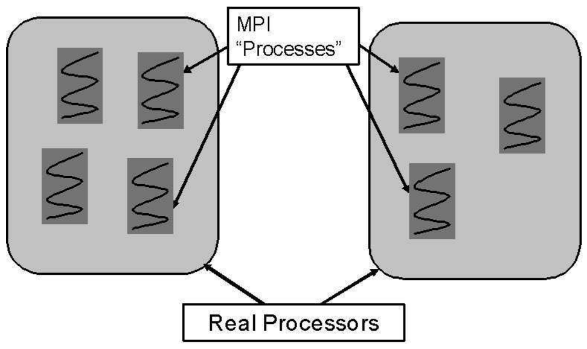


FIGURE 9.2: Implementation of AMPI virtual processors.

Better cache performance: A virtual processor handles a smaller set of data than a physical processor, so a virtual processor will have better memory locality. This blocking effect is the same method many *serial* cache optimizations employ, and AMPI programs get this benefit automatically.

Flexibility to run on an arbitrary number of processors: Since more than one VP can be executed on one physical processor, AMPI is capable of running MPI programs on any arbitrary number of processors. This feature proves to be useful in application development and debugging phases.

In many applications, we have demonstrated that the processor virtualization does not incur much cost in parallel performance [123], due to low scheduling overheads of user-level threads. In fact, it often improves cache performance significantly because of its blocking effect.

Charm++ and AMPI have been used as mature parallelization tools and runtime systems for a variety of real world applications for scalability [197, 140, 239, 80]. With these successes of improving parallel efficiency, several domain-specific frameworks on top of Charm++ and AMPI have been developed to further enhance programmer productivity while automating the parallelization process, which produces reusable libraries for parallel algorithms.

Additionally, Charm++ supports several useful features for monitoring running applications. Converse Client Server (CCS) [113, 47] provides a socket-based transport layer to get data in and out of any program, and NetFEM provides a higher level nodes-and-elements interface for remote online visualization, as shown in Figure 9.3.

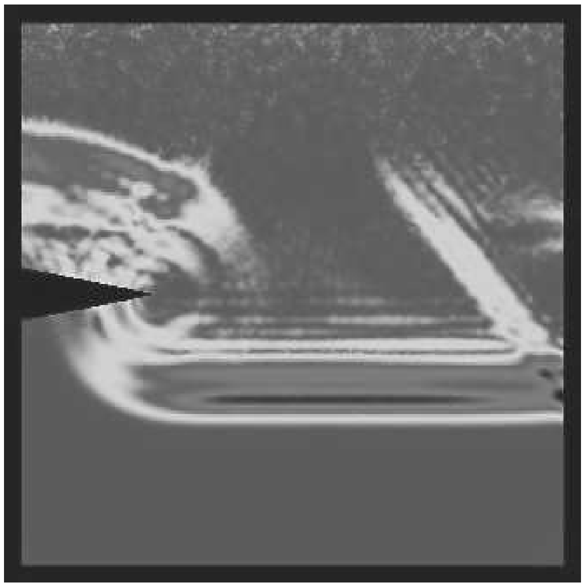


FIGURE 9.3: Diffraction off a crack, simulated in parallel and rendered using NetFEM directly from the running simulation **(see Color Plate 7)**.

9.2 Load Balancing Finite Element Codes in Charm++

Among the challenges associated with the parallelization of finite element codes, achieving load balance is key to scaling a dynamic application to a large number of processors. This is especially true for dynamic structural mechanics codes where simulations involve rapidly evolving geometry and physics, often resulting in a load imbalance between processors. As a result of this load imbalance, the application has to run at the speed of the slowest processor with deteriorated performance. The load imbalance problem has driven decades of research activities in load balancing techniques [30, 6, 251, 7].

Of interest in this work is dynamic load balancing, which attempts to solve the load balance problem at runtime according to the most up-to-date load information. This approach is a challenging software design issue and generally creates a burden for the application developers, who often must include the mechanism to inform the decision-making module concerning load balance the estimated CPU load and the communication structure. In addition, once load imbalance is detected and data migration is requested, a developer has to write complicated code for moving data across processors. The ideal load balancing framework should hide the details of load balancing so that the application developer can concentrate on modeling the physics of the problem.

In this section, we present an automatic load balancing method and its application to the three-dimensional finite element simulations of wave propagation and dynamic crack propagation events. The parallelization model used in this application is the processor virtualization supported by the migratable MPI threads. The application runs on a large number of MPI threads (that exceeds the actual physical number of processors), allowing to perform runtime load balancing by migrating MPI threads. The MPI runtime system automatically collects load information from the execution of the application. Based on this instrumented load data, the runtime module makes decisions on migrating MPI threads from heavily loaded processors to underloaded ones. This approach thus requires minimal efforts from the application developer.

9.2.1 Runtime Support for Thread Migration

In ParFUM applications, load balancing is achieved by migrating AMPI threads that host mesh partitions from overloaded processors to underloaded ones. When an AMPI thread migrates between processors, it must move all the associated data, including its stack and heap-allocated data. The Charm++ runtime supports both fully automated thread migration [253] and flexible user-controlled migration of data by additional helper functions.

In fully automatic mode, the AMPI runtime system automatically transfers a thread's stack and heap data which are allocated by a special memory allocator called an *isomalloc* [105], in a manner similar to that of $PM2$ [6]. It is portable on most platforms except for those where the `mmap` system call is unavailable. Isomalloc allocates data with a globally unique virtual address, reserving the same virtual space on all processors. With this mechanism, isomalloced data can be moved to a new processor without changing the address. This provides a clean way to move a thread's stack and heap data to a new machine automatically. In this case, migration is transparent to the user code.

Alternatively, users can write their own helper functions to pack and unpack heap data for migrating an AMPI thread. This is useful when application developers wish to have more control in reducing the data volume by using application specific knowledge and/or by packing only variables that are live at the time of migration. The PUP (Pack/UnPack) library [113] was written to simplify this process and reduce the amount of code the developers have to write. The developers only need to write a single PUP routine to traverse the data structure and this routine is used for both packing and unpacking.

9.2.2 Comparison to Prior Work

The goal of our work is a generic load balancing framework that optimizes the load balance of the irregular and highly dynamic applications with an application independent interface; therefore, we will focus our discussion in this section to those dynamic load balancing systems for parallel applications. In particular, we wish to distinguish our research using the following criteria:

- Supporting data migration. Migrating data has advantages over migrating "heavy-weight" processes, which adds complexity to the runtime system.
- General Purpose. Load balancing methods are designed to be application independent. They can be used for a wide variety of applications.
- Communication-aware load balancing. The framework takes communication into account explicitly, unlike implicit schemes which rely on domain-specific knowledge. Communication patterns, including multicast relationships and communication volume, are directly recorded into a load balancing database for load balancing algorithms.
- Automatic load measurement. The load balancing framework does not rely on the application developer to provide application load information, but measures computational costs at runtime.
- Adaptive to execution environment. Takes background load and non-migratable load into account.

Table 9.1 shows the comparison of the Charm++ load balancing framework to several other software systems that support dynamic load balancing. DRAMA [16] is designed specifically to support finite element applications. This specialization enables DRAMA to provide an "application independent"

System Name	Data Migration	General Purpose	Network Aware	Automatic Measurement	Adaptive
DRAMA	Yes	No	No	Yes	No
Zoltan	Yes	Yes	No	No	No
PREMA	Yes	No	No	No	No
Chombo	Yes	No	No	No	No
Charm++	Yes	Yes	Yes	Yes	Yes

TABLE 9.1: Software systems that support dynamic load balancing.

load balancing using its built-in cost functions for the category of applications. Zoltan [49, 50] does not make assumptions about applications' data and is designed to be a general purpose load balancing library. However, it relies on application developers to provide a cost function and communication graph. The system PREMA [8, 9] supports a very similar idea of migratable objects. However, its load balancing method primarily focuses on task scheduling problems as in noniterative applications. The Chombo [40] package has been developed by Lawrence Berkeley National Lab. It provides a set of tools including load balancing for implementing finite difference methods for the solution of partial differential equations on block-structured adaptively refined rectangular grids. It requires users to provide input for computational workload as a real number for each box (defined as a partition of mesh). Charm++ provides the most comprehensive features for load balancing. The measurement-based load balancing scheme enables automatic adaptation to application behavior. It is applicable to most scientific and engineering applications where computational load is persistent, even if it is dynamic. Charm++ load balancing is also capable of adapting to the change of background load [29] of the execution environment.

9.2.3 Automatic Load Balancing for FEM

Many modern explicit finite element applications are used to solve highly unsteady, dynamic, irregular problems. For example, an elasto-plastic solid mechanics simulator that we explore in Section 9.3 might use a different force calculation for highly stressed elements undergoing plastic deformation. Another simulation might use dynamic geometric mesh refinement to follow dynamic shocks [160]. In these applications, load balancing is required to achieve the desired high performance on large parallel machines.

ParFUM directly utilizes the load balancing framework in Charm++ and AMPI [251, 20]. The load balancing involves four distinct steps: (1) load evaluation; (2) load balancing initiation to determine when to start a new load balancing process; (3) load balancing decision making and (4) task and data migration.

The Charm++ load balancing framework adopts a unique measurement-based strategy for load evaluation. This scheme is based on runtime instrumentation, which is feasible due to the *principle of persistence* that can be found in most physical simulations: the communication patterns between objects as well as the computational load of each of them tend to persist over time, even in the case of dynamic applications. This implies that the recent past behavior of a system can be used as a good predictor of the near future. The load instrumentation is fully automatic at runtime. During the execution of a ParFUM application, the runtime measures the computation load for each object and records communication pattern into a load "database" on each processor. This approach provides an automatic load balancing solution that can adapt to application behavior while requiring minimal effort from the developers.

The runtime then assesses the load database periodically and determines if load imbalance is present. The load imbalance can be computed as:

$$\sigma = \frac{L_{max}}{L_{avg}} - 1, \tag{9.1}$$

where L_{max} is the maximum load across all processors, and L_{avg} is the average load of all the processors. Note that even when load imbalance occurs ($\sigma > 0$), it may not be profitable to start a new load balancing step due to the overhead of load balancing itself. In practice, a load imbalance threshold can be chosen based on a heuristic that the gain of the load balancing ($L_{max} - L_{avg}$) is at least greater than the estimated cost of the load balancing (C_{lb}). That is:

$$\sigma > \frac{C_{lb}}{L_{avg}}. \tag{9.2}$$

When load balancing is triggered, the load balancing decision module uses the load database to compute a new assignment of virtual processors to physical processors and informs the runtime to execute the migration decision.

9.2.4 Load Balancing Strategies

In the step that makes the load balancing decision, the Charm++ runtime assigns AMPI threads on physical processors, so as to minimize the maximum load (makespan) on the processors. This is known as the Makespan minimization problem, and the exact solution has been shown to be an NP-hard optimization problem [151]. However, many combinatorial algorithms have been developed that find a reasonably good approximate solution. Charm++ load balancing framework provides a spectrum of simple to sophisticated heuristic-based load balancing algorithms, some of which are described in more detail below:

- Greedy Strategy: This simple strategy organizes all the objects in decreasing order of their computation times. The algorithm repeatedly selects the heaviest un-assigned object and assigns it to the least loaded processor. This algorithm may lead to a large number of migrations. However, it works effectively in most cases.

- Refinement Strategy: The refinement strategy is an algorithm which improves the load balance by incrementally adjusting the existing object distribution, especially on highly loaded processors. The computational cost of this algorithm is low because only a subset of processors is examined. Furthermore, this algorithm results in only a few objects being migrated, which makes it suitable for fine-tuning the load balance.
- METIS-based Strategy: This strategy uses the METIS graph partitioning library [130] to partition the object-communication graph. The objective of this strategy is to find a reasonable load balance, while minimizing the communication among processors.

The Charm++ load balancing framework also allows a developer to implement his own load balancing strategies based on heuristics specific to the target application (such as in the NAMD [197] molecular simulation code).

Load balancing can be done in either centralized or distributed approach depending on how the load balancing decisions are made. In the centralized approach, one central processor makes the decisions globally. The load databases of all processors are collected to the central processor, which may incur high communication overhead and memory usage for the central processor. In the distributed approach, load balance decisions are made in a distributed fashion, where load data is only exchanged among neighboring processors. Due to the lack of the global information and aging of the load data, distributed load balancing tends to converge slowly to the good load balance discovered by the centralized approach. Therefore, we typically use a centralized or global load balancing strategy.

9.2.5 Agile Load Balancing

Applications with rapidly changing load require frequent load balancing, which demands rapid load balancing with minimal overhead. Normal load balancing strategies in Charm++ occur in *synchronous* mode, as shown in Figure 9.4. At load balancing time, the application on each processor stops after it finishes its designated iterations and hands control to the load balancing framework to make load balancing decisions. The application can only resume when the load balancing step finishes and all AMPI threads migrate to the destination processors. In practice, this "stop and go" load balancing scheme is simple to implement, and has one important advantage—thread migration happens under user control, so that a user can choose a convenient time for the thread migration, to minimize the implementation complexity and runtime data size of the migration. However, this scheme is not efficient due to the effect of the global barrier. It suffers from high overhead due to the fact that the load balancing process on the central processor has to wait

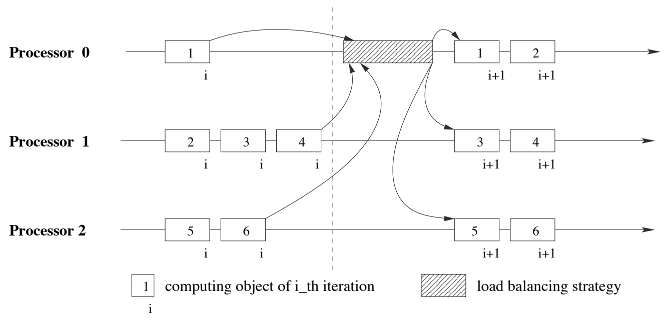


FIGURE 9.4: Traditional synchronous load balancing.

for the slowest processor to join load balancing, thus wasting CPU cycles on other processors. This motivated the development of an agile load balancing strategy that performs *asynchronous* load balancing which allows overlapping of load balancing time and normal computation.

The asynchronous load balancing scheme takes full advantage of Charm++'s intelligent runtime support for concurrent compositionality [123] that allows dynamic overlapping of the execution of different composed modules in time and space. In the asynchronous scheme, the load balancing process occurs concurrently, or in the background of normal computation. When it is time for load balancing, each processor sends its load database to the central processor and continues its normal computation without waiting for load balancing to start. When a migration decision is calculated at the background on the central processor, the AMPI threads are instructed to migrate to their new processors in the middle of their computation.

There are a few advantages of asynchronous load balancing over the synchronous scheme. First, eliminating the global barrier helps in reducing the idle time on faster processors which otherwise would have to wait for the slower processors to join the load balancing step. Second, it allows the overlapping of load balancing decision making time and computation in an application, which potentially could help improve the overall performance. Finally, each thread can have more flexible control on when to migrate to the designated processor. For example, a thread can choose to migrate when it is about to be idle, which potentially allows overlapping of the thread migration and computation of other threads.

Asynchronous load balancing, however, imposes a significant challenge to thread migration in the AMPI runtime system. AMPI threads may migrate *at any time*, whenever they receive the migration notification. In practice, it is not trivial for an AMPI thread to migrate at any time due to the complex runtime state involved, for example when a thread is suspended in the middle of pending receives. In order to support any-time migration of AMPI threads, we extended the AMPI runtime to be able to transfer a complete runtime state associated with the AMPI threads including the pending receive requests and buffered messages for future receives. With the help of isomalloc stack and heap, AMPI threads can be migrated to a new processor transparently at any time: a thread can actually be suspended on one processor, migrated and resumed on a different processor in a new address space. For AMPI threads with pending receives, incoming messages are redirected automatically to the destination processors by the runtime system.

In the next section, we present a simulation case study to demonstrate the effectiveness of our finite element framework and load balancing strategies.

9.3 Cohesive and Elasto-plastic Finite Element Model of Fracture

To simulate the spontaneous initiation and propagation of a crack in a discretized domain, an explicit cohesive-volumetric finite element (CVFE) scheme [246], [31], [74] is used. As its name indicates, the scheme relies on a combination of volumetric elements used to capture the constitutive response of the continuum medium, and cohesive interfacial elements to model the failure process taking place in the vicinity of the advancing crack front. The CVFE concept is illustrated in Figure 9.5, which presents two 4-node tetrahedral volumetric elements tied together by a 6-node cohesive element shown in its deformed configuration, as the adjacent nodes are initially superposed and the cohesive element has no volume.

In the present study, the mechanical response of the cohesive elements is described by the bilinear traction-separation law illustrated in Figure 9.6 for the case of tensile (Mode I) failure. After an initial stiffening (rising) phase, the

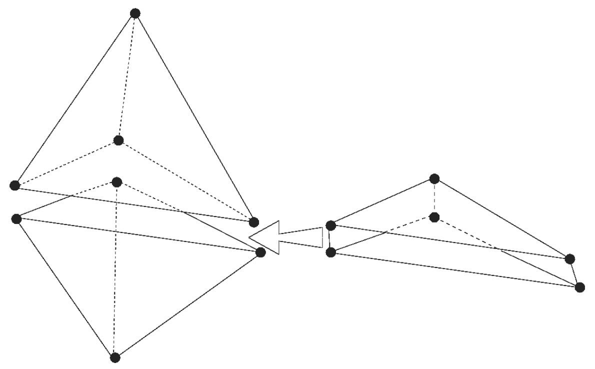


FIGURE 9.5: Two 4-node tetrahedral volumetric elements linked by a 6-node cohesive element.

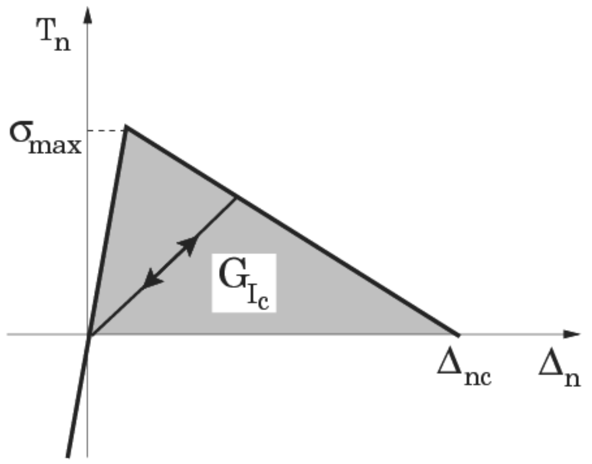


FIGURE 9.6: Bilinear traction-separation law for mode I failure modeling. The area under the curve corresponds to the mode I fracture toughness G_{I_c} of the material.

cohesive traction T_n reaches a maximum corresponding to the failure strength σ_{max} of the material, followed by a downward phase that represents the progressive failure of the material. Once the critical value Δ_{nc} of the displacement jump is reached, no more traction is exerted across the cohesive interface and a traction-free surface (i.e., a crack) is created in the discretized domain. The emphasis of the dynamic fracture study summarized hereafter is on the simulation of purely Mode I failure, although cohesive models have also been proposed for the simulation of mixed-mode fracture events. Also illustrated in Figure 9.6 is an unloading and reloading path followed by the cohesive traction during an unloading event taking place while the material fails.

The finite element formulation of the CVFE scheme is derived from the following form of the principle of virtual work:

$$\int_V (\rho \ddot{u}_i \, \delta u_i + S_{ij} \, \delta E_{ij}) \, dV = \int_{S_T} T_i^{ex} \, \delta u_i \, dS_T + \int_{S_c} T_i \, \delta \Delta_i \, dS_c, \tag{9.3}$$

where the left-hand side corresponds to the virtual work done by the inertial forces ($\rho \ddot{u}_i$) and the internal stresses (S_{ij}), and the right-hand side denotes the virtual work associated with the externally applied traction (T_i^{ex}) and cohesive traction (T_i) acting along their respective surfaces of application S_T and S_c. In Equation (9.3), ρ denotes the material density, u_i and E_{ij} are the displacement and strain fields, respectively, and Δ_i denotes the displacement jump across the cohesive surfaces. The implementation relies on an explicit time stepping scheme based on the central difference formulation [246]. A nonlinear kinematics description is used to capture the large deformation and rotation associated with the propagation of the crack. The strain measure used here is the Lagrangian strain tensor $\mathbf{E}$.

To complete the CVFE scheme, we need to model the constitutive response of the material, i.e., to describe the response of the volumetric elements. In the present study, we use an explicit elasto-visco-plastic update scheme nonlinear elasticity, which is compatible with the nonlinear kinematic description and relies on the multiplicative decomposition of the deformation gradient $\mathbf{F}$ into elastic and plastic parts as

$$\mathbf{F} = \mathbf{F}^e \mathbf{F}^p. \tag{9.4}$$

The update of the plastic component $\mathbf{F}^p$ of the deformation gradient at the $(n+1)^{th}$ time step is obtained by

$$\mathbf{F}^p_{n+1} = exp\left[\sum_A \frac{\Delta\gamma}{\sqrt{2}\tilde{\sigma}}\left(\sigma^A - \frac{I_1^\sigma}{3}\right)\mathbf{N}^A \otimes \mathbf{N}^A\right] \bullet \mathbf{F}^p_n, \tag{9.5}$$

where $\mathbf{N}^A$ (A=1, 2, 3) denote the Lagrangian axes defined in the initial configuration, $\Delta\gamma$ is the discretized plastic strain increment, I_1^σ is the first Cauchy stress invariant and $\tilde{\sigma} = \sqrt{(\sigma' : \sigma')/2}$ is the effective stress, with σ' denoting the Cauchy stress deviator whose spectral decomposition is

$$\sigma' = \sum_A \left(\sigma^A - \frac{I_1^\sigma}{3}\right)\mathbf{N}^A \otimes \mathbf{N}^A. \tag{9.6}$$

The plastic strain increment is given by $\Delta\gamma = \Delta t\, \dot{\gamma}$, where the plastic strain rate is described in this study by the classical Perzyna two-parameter model [196]

$$\dot{\gamma} = \eta \left(\frac{f(\sigma)}{\sigma_Y}\right)^n, \tag{9.7}$$

in which n and η are material constants, σ_Y is the current yield stress and $f(\sigma) = (\tilde{\sigma} - \sigma_Y)$ is the overstress. Strain hardening is captured by introducing a tangent modulus E_t relating the increment of the yield stress, $\Delta\sigma_Y$, to the plastic strain increment, $\Delta\gamma$. Finally, the linear relation

$$\mathbf{S} = \mathbf{L}\mathbf{E} \tag{9.8}$$

between the second Piola-Kirchhoff stresses $\mathbf{S}$ and the Lagrangian strains $\mathbf{E}$ is used to describe the elastic response. Assuming material isotropy, the stiffness tensor $\mathbf{L}$ is defined by the Young's modulus E and Poisson's ratio ν.

The main source of load imbalance comes from the very different computational costs associated with the elastic and visco-plastic constitutive updates. As long as the effective stress remains below a given level (chosen in this study as 80% of the yield stress), only the elastic relation (9.8) is computed. Once this threshold is reached for the first time, the visco-plastic update is performed, which typically represents a doubling in the computational cost. Consequently, as the crack propagates through the discretized domain, the load associated with each processor can be substantially heterogeneous due to the plastic zone around the crack tip, thus suggesting the need for a robust dynamic load balancing scheme as described in Section 9.2.

9.3.1 Case Study 1: Elasto-Plastic Wave Propagation

The first application is the quasi-one-dimensional elasto-plastic wave propagation problem depicted in Figure 9.7. It consists of a rectangular bar of length $L = 10$ m and cross-section $A = 1$ m^2. The bar is initially at rest and stress free. It is fixed at one end and subjected at the other end to an applied velocity V ramped linearly from 0 to 20 m/s over .16 ms and then held at a constant velocity of 20 m/s thereafter. The time step size is $3\,\mu$s and the total number of time steps is 1,100. The material properties are chosen as follows: yield stress $\sigma_Y = 480$ MPa, stiffness $E = 73$ GPa and $E_t = 7.3$ GPa, exponent $n = 0.5$, fluidity $\eta = 10^{-6}$/s, Poisson's ratio $\nu = .33$ and density $\rho = 2800$ kg/m^3.

The applied velocity generates a one-dimensional stress wave that propagates through the bar and reflects from the fixed end. At every wave reflection, the stress level in the bar increases as the end of the bar is continuously pulled at a velocity V. During the initial stage of the dynamic event, the material response is elastic as the first stress wave travels through the bar at the dilatational wave speed $c_d = 6215$ m/s with an amplitude

$$\sigma = \rho c_d V = 348\,\text{MPa} \; < \sigma_Y . \tag{9.9}$$

After one reflection of the wave from the fixed end, the stress level in the bar exceeds the yield stress of the material and the material becomes plastic. A snapshot of the location of the elasto-plastic stress wave is shown in Figure 9.7. The computational overload associated with the plastic update routine (approximately a factor of two increase compared to the elastic case) leads to a significant dynamic load imbalance while the bar transforms from elastic to plastic. As mentioned earlier, in these simulations, the plastic check and update subroutine is called upon when the equivalent stress level exceeds 80% of the yield stress.

The unstructured 800,000-element tetrahedral mesh that spans the bar is initially partitioned into chunks using METIS, and these chunks are then mapped to the processors. During the simulation, the processors advance in lockstep with frequent synchronizing communications required by exchanging of boundary conditions, which may lead to bad performance when load imbalance occurs.

The simulation was run on Tungsten Xeon Linux cluster at the National Center for Supercomputing Applications (NCSA). This cluster is based on Dell PowerEdge 1750 servers, each with two Intel Xeon 3.2 GHz processors, running Red Hat Linux and Myrinet interconnect network. The test ran on

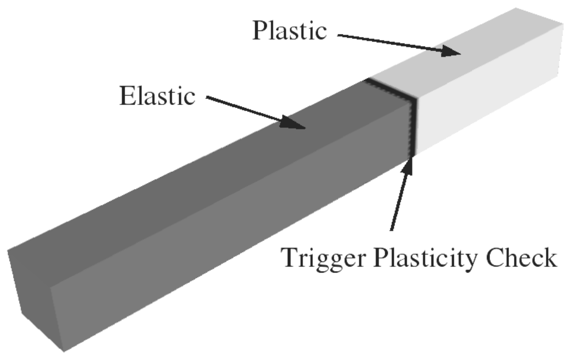


FIGURE 9.7: Location of the traveling elasto-plastic wave at time $c_d t/L = 1.3$.

32 processors with 160 AMPI virtual processors. Figure 9.8 shows the results without load balancing in a CPU utilization graph over a certain time interval. The figure was generated by *Projections* [125], a performance visualization and analysis tool associated with Charm++ that supplies application-level visual and analytical performance feedback. This utilization graph shows how the overall utilization changes as the wave propagates through the bar. The total runtime was 177 seconds for this run.

A separate interest, although not investigated further in this study, is the period of initial load imbalance (observed for the runtime before 48 s in

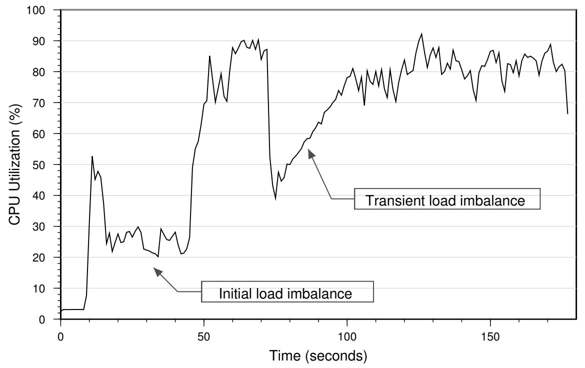


FIGURE 9.8: CPU utilization graph without load balancing (Tungsten Xeon).

Figure 9.8) caused by the quiet generation of subnormal numbers (floating-point numbers that are very close to zero) during the initial propagation of the elastic wave along the initially quiescent bar. This phenomenon is discussed by Lawlor et al. [145], who propose an approach to mitigate such performance effects caused by the inherent processor design. However, this study is only concerned with the load imbalance associated with the transformation of the bar from elastic to plastic (observed for the runtime between 72 s and 100 s in Figure 9.8).

As indicated earlier, the load imbalance in this problem is highly transient, as elements at the wave front change from an elastic to a plastic state. In Figure 9.9, the effects of the plasticity calculations are clearly noticeable in terms of execution time which linearly ramps from the condition of fully elastic to fully plastic resulting in a doubling of the execution time. This leads to a load imbalance, which is resolved by migrating chunks from heavily loaded processors to light ones while the bar goes into the plastic regime.

Even though we used a variety of methods and time frames, the problem was not considerably sped up by load balancing. The transition time was too fast for the load balancer to significantly speed up the simulation. Also the period of imbalance is a very small portion compared to the total runtime. Therefore, a performance improvement here necessitates that the overhead and delays associated with the invocation of the load balancer be minimal. Nevertheless, we managed to speed up the simulation by 7 seconds as shown in

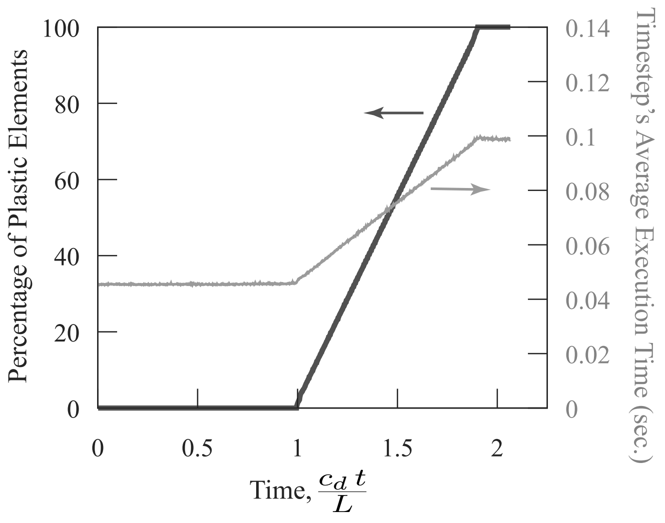


FIGURE 9.9: Evolution of the number of plastic elements.

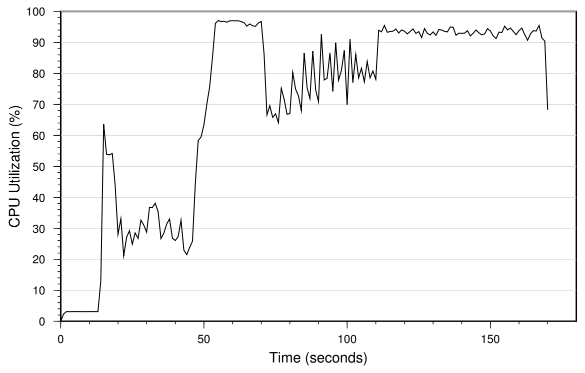


FIGURE 9.10: CPU utilization graph with synchronous load balancing (Tungsten Xeon).

Figure 9.10. The time required for completion reduces to 170 seconds, which yields a 4 percent overall improvement by the load balancing.

We repeated the same test on 32 processors of the SGI Altix (IA64) at NCSA with the same 160 AMPI virtual processors. Figure 9.11 shows the result without load balancing in the Projections utilization graph. The total execution time was 207 seconds and a more severe effect of subnormal numbers on this machine was observed in the first hundred seconds of execution time.

In the second run, we ran the same test with the greedy load balancing scheme described in the previous section. The result is shown in Figure 9.12(a) in the same utilization graph. The load balancing is invoked around time

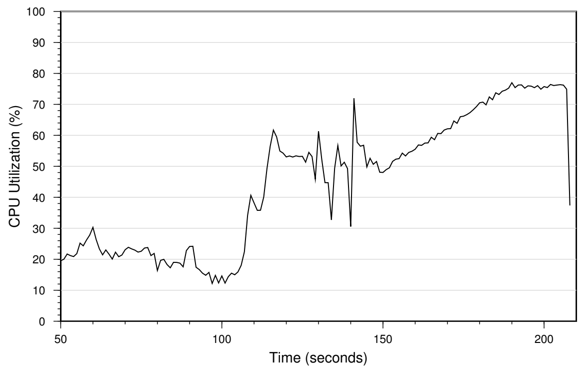
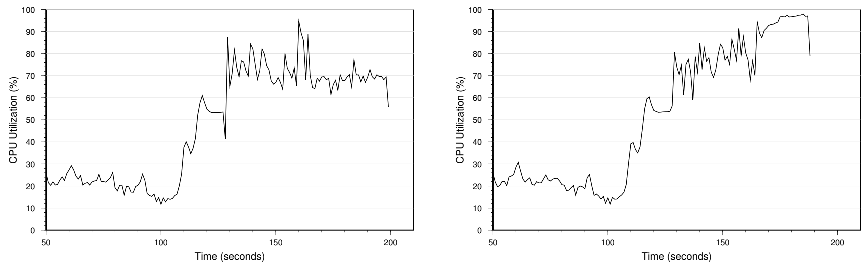


FIGURE 9.11: CPU utilization graph without load balancing (SGI Altix).



(a) Synchronous load balancing. (b) Asynchronous load balancing.

FIGURE 9.12: CPU utilization graph (SGI Altix).

interval 130 in the figure. After the load balancing, the CPU utilization is slightly improved and the total execution time is now around 198 seconds.

Finally, we ran the same test with the same greedy algorithm in an asynchronous load balancing scheme described in Section 9.2.5. The asynchronous load balancing scheme avoids the stall of an application for load balancing and overlaps the computation with the load balancing and migration. The result is shown in Figure 9.12(b) in a utilization graph. It can be seen that, after load balancing, the overall CPU utilization was further improved and the total execution time is 187 seconds, which is a 20 second improvement.

9.3.2 Case Study 2: Dynamic Fracture

The second application involves a single edge notched fracture specimen of width $W = 5$ m, height $H = 5$ m, thickness $T = 1$ m and initial crack length $a_0 = 1$ m, having a weakened plane starting at the crack tip and extending along the crack plane to the opposite edge of the specimen. The material properties used in this simulation are $\sigma_Y = 900$ MPa, $E = 210$ GPa, $E_t = 2.4$ GPa, $n = 0.5$, $\eta = 10^{-6}$/s, $\nu = .3$ and $\rho = 7850$ kg/m^3. The boundary conditions along the top and bottom surfaces of the specimen have a linearly ramped velocity of 0.0 to 1.0 m/s over 2.0 ms which is then held at a constant velocity of 1.0 m/s thereafter. The time step size is .47 μs and the total number of time steps is 1.25e5. A single layer of six-node cohesive elements is placed along the weakened interface, with the failure properties described by a critical crack opening displacement value $\Delta_{nc} = .8$ mm and a cohesive failure strength $\sigma_{max} = 95$ MPa. The mesh consists of 91,292 cohesive elements along the interface plane and 4,198,134 linear strain tetrahedral elements. As the stress wave emanating from the top and bottom edges of the specimen reaches the fracture plane, a region of high stress concentration is created around the initial crack tip. In that region, the equivalent stress exceeds the yield stress of the material leading to the creation of a plastic zone. As the stress level

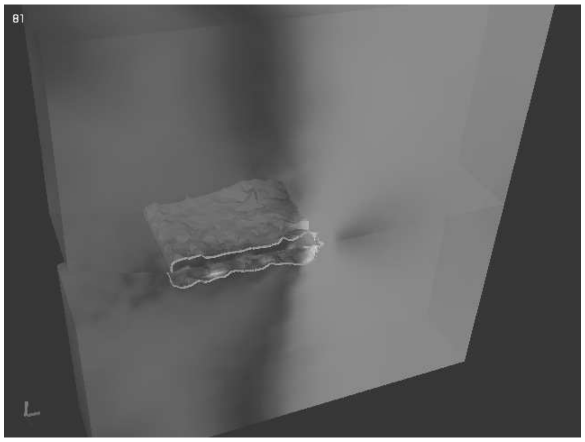


FIGURE 9.13: Snapshot of the plastic zone surrounding the propagating planar crack at time $c_d t/a_0 = 27$. The iso-surfaces denote the extent of the region where the elements have exceeded the yield stress of the material **(see Color Plate 7)**.

continues to build up in the vicinity of the crack front, the cohesive tractions along the fracture plane start to exceed the cohesive failure strength of the weakened plane and a crack starts to propagate rapidly along the fracture plane, surrounded by a plastic zone and leaving behind a plastic wake, as illustrated in Figure 9.13.

This simulation was run on the Turing cluster at the University of Illinois at Urbana-Champaign. The cluster consists of 640 dual Apple G5 nodes connected with Myrinet network. The simulation without load balancing took about 12 hours on 100 processors. The evolution of the average processor utilization is shown in the bottom curve of Figure 9.14. As apparent there, around time 10,000 seconds, the CPU utilization dropped from around 85% to only about 44%. This is due to the advent of the elastic elements transitioning into plastic elements around the crack tip, leading to the beginning of load imbalance. As shown in Figure 9.15, the number of plastic elements starts to increase dramatically as the crack starts to propagate along the interface. As more elastic elements turn plastic, the CPU utilization slowly increases and stays around 65% (lower curve in Figure 9.14). The load imbalance can also be easily observed in the CPU utilization graph over processors in Figure 9.16(a). While some of the processors have CPU utilization as high as about 90%, some processors only have about 50% of CPU utilization during the whole execution.

With the greedy load balancing strategy invoked every 500 time steps, the simulation finished in only about 9.5 hours, a saving of nearly 2.5 hours or 20% over the same simulation with no load balancing. This increase is caused by the

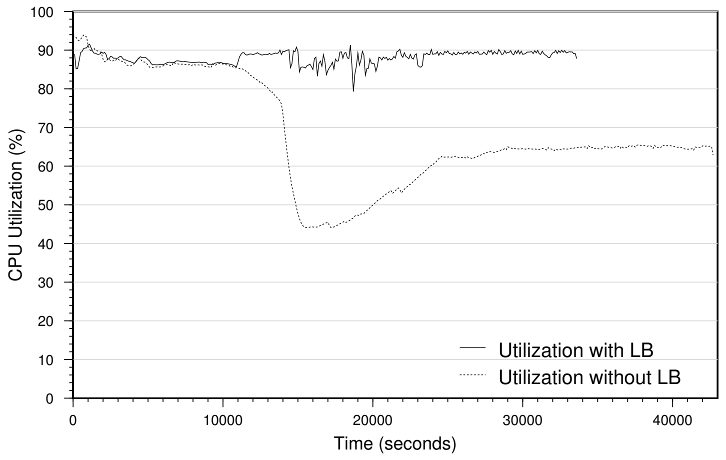


FIGURE 9.14: CPU utilization graph with and without load balancing for the fracture problem shown in Figure 9.13 (Turing Apple Cluster).

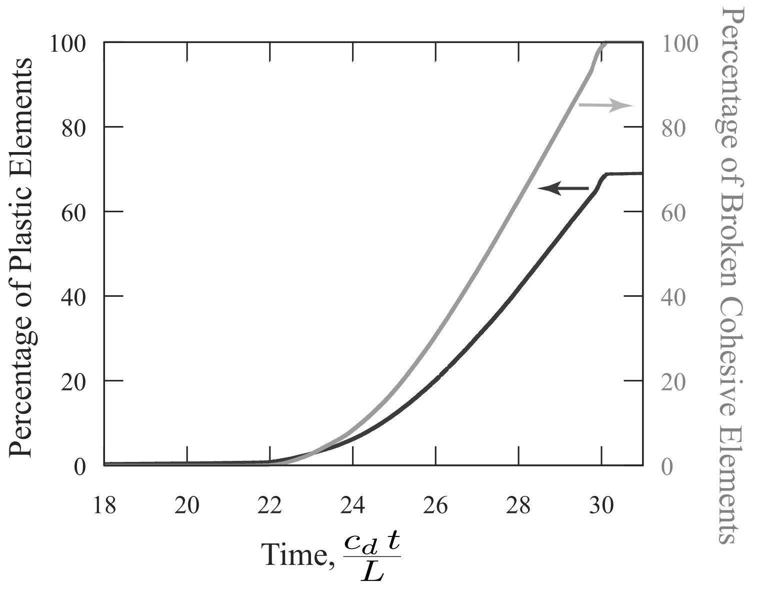
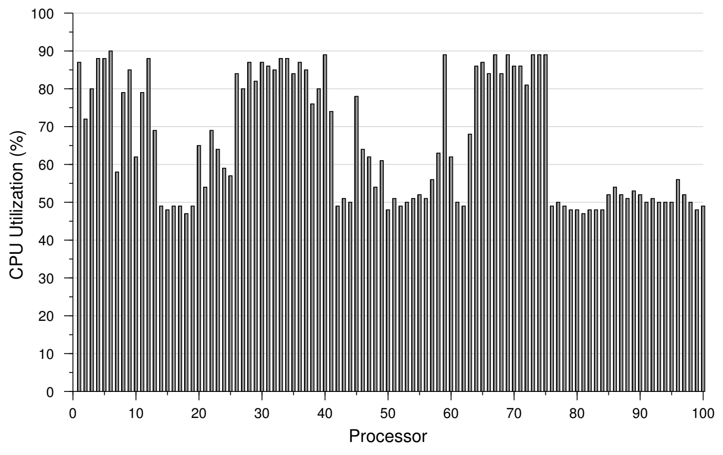
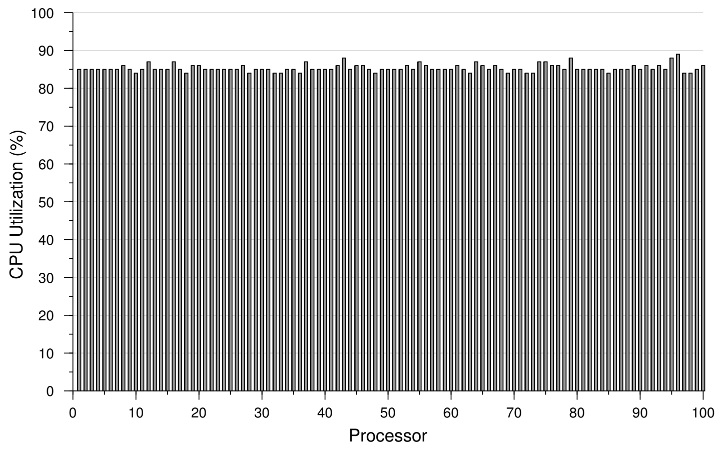


FIGURE 9.15: Evolution of the number of plastic and broken cohesive elements.



(a) Without load balancing.



(b) With load balancing.

FIGURE 9.16: CPU utilization across processors (Turing Apple Cluster).

overall increased processor utilization, which can be seen in the upper curve of Figure 9.14. The peaks correspond to the times when the load balancer is activated during the simulation. There is an immediate improvement in the utilization when the load balancer is invoked. Then the performance slowly deteriorates as more elements become plastic. The next invocation tries to balance the load again. Figure 9.16(b) further illustrates that load balance has been improved from Figure 9.16(a) in the view of the CPU utilization across processors. It can be seen that a CPU utilization of around 85% is achieved on all processors with negligible load variance.

9.4 Conclusions

Dynamic and adaptive parallel load balancing is indispensable for handling load imbalance that may arise during a parallel simulation due to mesh adaptation, material nonlinearity and other modern irregular dynamic simulation behavior. We have demonstrated the successful application of the Charm++ measurement-based dynamic load balancing concept to a crack propagation problem, modeled with a cohesive/volumetric finite element scheme. The performance of the application was improved by an agile load balancing strategy which is designed to handle transient load imbalance due to the rapidly propagating wave. The performance study of this application demonstrated the ability of the automatic load balancing to achieve sustained high computational efficiency.

Acknowledgments

This work was an adjunct project affiliated with the Center for Simulation of Advanced Rockets (CSAR) which was funded by the U.S. Department of Energy's Advanced Simulation and Computing program.