Engineering With Computers manuscript No.
(will be inserted by the editor)

Parallel Adaptive Simulations of Dynamic Fracture Events

Sandhya Mangala[1], Terry Wilmarth[3], Sayantan Chakravorty[2], Nilesh Choudhury[2], Laxmikant V. Kalé[2] and Philippe H. Geubelle[1]

[1] *Department of Aerospace Engineering*
[2] *Department of Computer Science*
[3] *Center for Simulation of Advanced Rockets*
University of Illinois at Urbana-Champaign
{mangala, wilmarth, schkrvrt, nchoudh2, kale, geubelle}@uiuc.edu

The date of receipt and acceptance will be inserted by the editor

Abstract Finite element simulations of dynamic fracture problems usually require very fine discretizations in the vicinity of the propagating stress waves and advancing crack fronts, while coarser meshes can be used in the remainder of the domain. This need for a constantly evolving discretization poses several challenges, especially when the simulation is performed on a parallel computing platform. To address this issue, we present a parallel computational framework developed specifically for unstructured meshes. This framework allows dynamic adaptive refinement and coarsening of finite element meshes and also performs load balancing between processors. We demonstrate the capability of this framework, called ParFUM, using two-dimensional structural dynamic problems involving the propagation of elastodynamic waves and the spontaneous initiation and propagation of cracks through a domain discretized with triangular finite elements.

1 Introduction

Explicit finite element schemes are often the method of choice to model dynamic fracture events. Examples include the virtual internal bond method [1,2], the cohesive finite element scheme [3–5] and dynamic damage modeling [6]. All these methods involve a set of computational issues associated with a rapidly changing geometry, high stress concentrations and singularities due to wave fronts and propagating cracks, the need for large domains to avoid interactions of waves emanating from the boundary, and a requirement for a very fine mesh around the crack tip to capture the failure process accurately. Since the use of a very fine grid in the entire computational domain is often prohibitively expensive, one needs to adapt the mesh constantly to account for the rapidly moving wave fronts and dynamically propagating crack tips.

In rapidly evolving problems such as dynamic fracture events, mesh modification poses a set of challenges with regard to the choice of the mesh adaptivity criterion and parallel implementation. The first step in mesh adaptivity is to identify automatically the critical regions of the mesh to be adapted using indicators based on mesh quality measures, error indicators or sharp gradients in the solution. Two classes of mesh adaptivity indicators, one based on an interpolation error [7] and the other based on *a posteriori* error estimates [8] have been proposed. The key idea of adaptivity criteria is to distribute the error indicator measure equally over the entire mesh. Wherever this measure is high, the mesh is refined to capture the spatial variation of the quantity underlying the error indicator more accurately; where the measure is low, the mesh is coarsened. By this process, an optimal mesh discretization is attained. In this work, we adopt the method based on the equi-distribution of the variation of the velocity field [7,9,10].

The second and undoubtedly more challenging issue is associated with the parallel implementation of the finite element solver. Parallel computing not only reduces wall-clock time but also allows us to solve very large problems which otherwise could not be simulated on a single processor. However, parallel programming presents additional issues of communication and data organization across multiple processors. These issues become particularly more complex in problems involving frequent adaptive remeshing as the nature of the discretization and its associated data structures change repeatedly over time. This in turn affects the per-processor computational load.

A parallel framework has been developed to facilitate the parallel implementation of large unstructured finite element or finite volume simulations. ParFUM [11] is a Parallel Framework for Unstructured Meshes developed in Charm++ [12]. It enables the relatively straightforward parallelization of existing serial applications and provides a simple environment in which to develop par-

allel unstructured mesh codes with minimal knowledge of parallel programming. A primary advantage to the use of ParFUM is the ability to perform automatic dynamic load balancing with minimal effort on the part of the programmer.

This article summarizes a recent study of 2D adaptive parallel finite element simulations of wave propagation and dynamic fracture problems implemented with ParFUM. We begin in Section 2 with a description of ParFUM and the mesh adaptivity strategy used by this framework. We then discuss the adaptivity criterion adopted in this work to refine and coarsen the mesh in Section 3. In Sections 4 and 5 we present two sets of elastodynamic and dynamic fracture applications and discuss the associated precision and computational efficiency issues. Finally, in Section 6, we present performance studies with the dynamic fracture application, highlighting the use of load balancing in ParFUM.

2 Adaptive Mesh Modification with ParFUM

ParFUM embodies a unique approach to parallelization of unstructured meshes and the applications that make use of them that leverages the support of the Charm++ run-time system and its associated features. Other support frameworks exist for unstructured meshing. One such notable approach is the SIERRA [13] framework from Sandia National Laboratories. SIERRA supports a large feature set and is designed to execute large-scale simulation applications on the latest supercomputers, but is not currently publicly available. On a related note, the TSTT Project [14] aims to provide interoperability amongst simulation tools by providing standardized interfaces.

2.1 The Charm++ Run-time system

ParFUM, as mentioned earlier, is based on the Charm++ run-time system and AMPI [15]. Therefore ParFUM inherits the many capabilities provided by the Charm++ run-time system such as processor virtualization, adaptive overlap of communication and computation, dynamic load balancing and portability to a multitude of parallel and distributed computing platforms.

Processor virtualization [16] is at the heart of the design of the Charm++ run-time system. Virtualization involves the decomposition of a computation into a large number of interacting entities or virtual processors (VPs) with little concern for the actual number of physical processors. As shown in Figure 1, the Charm++ run-time system maps VPs to physical processors and allows the VPs to interact via asynchronous method invocations. Charm++ users develop applications without concern for the creation of good mappings based on problem size and the number of physical processors. Charm++ and AMPI provide flexible strategies for setting up initial mappings, along with reasonable default mappings. Further, the Charm++ run-time system can modify the mapping of VPs to physical processors automatically at execution time by migrating VPs between physical processors. This allows the run-time system to perform measurement-based dynamic load balancing. Charm++ supports a variety of load balancing strategies suitable for different load patterns. Load balancing is discussed in further detail in Section 6.2. Processor virtualization also allows for the adaptive overlap of computation and communication. When one VP is waiting for a communication to take place, another VP on the same processor can perform computations. This reduces the amount of time that a processor would spend idle awaiting communication. Virtualization benefits are further illustrated in Section 6.1.

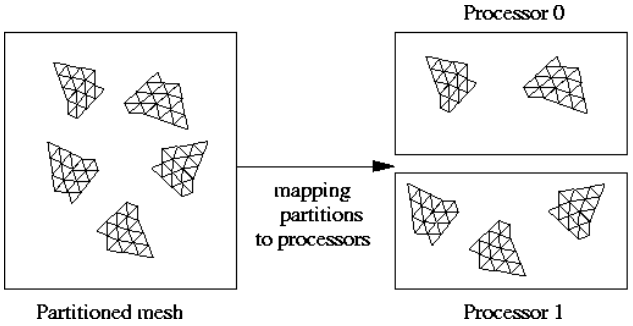

Fig. 1 The Charm++ run-time system maps virtual processors to physical processors. In ParFUM, each virtual processor is associated with one partition of the mesh.

Adaptive MPI (AMPI) [15,17] is an implementation of the Message Passing Interface (MPI) in Charm++. An MPI process consists of a user-level thread bound to a Charm++ VP. AMPI allows existing MPI applications to make use of Charm++'s capabilities, such as load balancing and adaptive overlap, with minimal modification. The base ParFUM implementation is in AMPI, allowing for rapid porting of existing MPI simulations to ParFUM.

2.2 ParFUM

ParFUM provides parallel support for unstructured mesh applications. As such, it supports parallel mesh partitioning, access to remote entities or *ghosts* along partition boundaries, updates of ghost data and reductions on shared partition boundary nodes, topological adjacencies and mesh adaptivity.

A ParFUM application has two subroutines: *init* and *driver*. The *init* subroutine executes only on VP 0 and reads the input mesh and data and registers it with the framework. The framework then partitions the mesh into the requested number of mesh regions, using a memory efficient parallel partitioner based on ParMetis [18]. Alternatively, a pre-partitioned mesh may be read in par-

allel in the *driver* routine. Each partition belongs to a single CHARM++ virtual processor and vice versa. The *driver* routine is executed on each VP and computes the solution over the partition for that virtual processor.

A typical solution loop performs calculations over each node or element and requires data from the neighboring nodes or elements. Nodes and elements are collectively referred to as *entities*. Thus entities on a partition boundary may need data from entities on other partitions. ParFUM provides functionality for adding local read-only copies of remote entities to the partition boundary. These read-only entities are called *ghosts*. A single collective call to ParFUM updates all ghost entities with data from the original entities on neighboring partitions. As the definition of "neighboring" varies 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 nodes 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. The definition of "neighboring" can vary for the different layers. User-specified ghost layers are automatically added during the partitioning of the input mesh. ParFUM also updates the connectivity and adjacency information of a partition's entities to reflect the additional layers of ghosts.

ParFUM further simplifies application development by providing the user with topological information. From the input mesh connectivity, ParFUM derives node-to-element, node-to-node and element-to-element adjacencies. These adjacencies are calculated only if the user requests them, avoiding the overhead of calculating and storing this data when it is not required.

ParFUM has additional capabilities such as load balancing and mesh adaptivity. Load balancing involves moving VPs (mesh partitions) from one physical processor to another. User data for a partition must be packed, transported and unpacked, a process that is automated in ParFUM. The user writes a single data structure traversal routine which is used for both packing and unpacking the data to be migrated. We discuss the load balancing of ParFUM applications in greater detail in Section 6.2.

2.3 Mesh Adaptivity in ParFUM

Mesh modification is accomplished in ParFUM in a manner that differs from other approaches. At the lowest level of detail, the framework provides low-level mesh modification primitives for *edge bisect*, *edge flip* and *edge contract* operations. These operations are self-contained parallel primitives that leave the mesh in a consistent state, updating all adjacencies and ghost layers as required. The primitives lock mesh entities so that multiple primitives can simultaneously operate on adjacent areas of the mesh. Higher-level meshing algorithms can be developed using these primitives with limited knowledge of parallel programming. ParFUM provides several such higher-level refinement and coarsening operations that are capable of modifying the mesh to the user's sizing specification. These operations include quality measures to improve element quality during refinement and coarsening. The current methods involve selecting elements to refine or coarsen based on edge lengths and element quality. Refinement incorporates the longest edge bisection strategies [19] of Rivara while coarsening involves shortest edge contractions. Other parallel approaches [20] to mesh adaptivity also incorporate strategies similar to Rivara's. In longest edge bisection, the longest edge of an element is identified for bisection. If the neighboring element along the longest edge considers that edge to be its longest edge as well, a point is inserted on the edge, typically at the midpoint, and new elements are created by bisecting the two adjacent elements at that point. We have two approaches to longest edge bisection algorithms that modify regions of the mesh. The first approach involves sorting the elements by the lengths of their longest edges and applying the primitive edge-bisect operation in that order. In parallel, this is implemented by sorting over the elements of each separate mesh partition. This approach is the primary approach used in the applications discussed subsequently. ParFUM also implements a second approach to longest edge bisection. In this approach, the longest edge bisection propagates through the mesh when the neighboring element does not share its longest edge with the initiating element. A combination of edge-flip and edge-bisect operations propagate through the mesh until an edge satisfying that relationship is reached.

In addition to leaving the mesh consistent after a primitive operation, ParFUM provides solution transfer capabilities for these operations. It provides some basic techniques for linear interpolation or volume-weighted solution transfer, but makes it possible for the user to override these approaches with specific techniques.

The simulation of dynamic fracture problems based on the finite element formulation described in Sections 4 and 5 constitutes an excellent testbed for ParFUM's parallel mesh adaptivity, solution transfer and load balancing capabilities.

3 Finite element formulation and mesh adaptivity criterion

The finite element formulation used in this work starts from the principle of virtual work over the deformable solid Ω:

$$\int_{\Omega} \mathbf{S} : \delta\mathbf{E}\, d\,\Omega + \int_{\Omega} \rho_o \ddot{\mathbf{u}}.\delta\mathbf{u}\, d\,\Omega = \int_{\Gamma_{ex}} \mathbf{T}_{ex}.\delta\mathbf{u}\, d\,\Gamma, \quad (1)$$

where $\mathbf{S}$ and $\mathbf{E}$ respectively denote second Piola-Kirchoff stress and the Lagrangian strain tensors, $\mathbf{u}$ is the displacement vector field, a superposed dot denotes differentiation with time, $\mathbf{T}_{ex}$ is the traction applied along the boundary Γ_{ex}, and ρ_o is the material density. The corresponding semi-discrete finite element formulation is

$$\mathbf{M}\,\mathbf{a} = -\mathbf{R}^{\mathbf{in}} + \mathbf{R}^{\mathbf{ex}}, \tag{2}$$

where $\mathbf{M}$ is the lumped mass matrix, $\mathbf{a}$ is the nodal acceleration vector, $\mathbf{R}^{\mathbf{in}}$ and $\mathbf{R}^{\mathbf{ex}}$ respectively denote the internal and external force vectors. The numerical scheme is completed by a second-order explicit (central difference) timestepping scheme. For more details, see [5].

To reduce the incidence of the numerical oscillations inherent in explicit time integration scheme, we adopt the artificial viscous damping method proposed by Lew *et al.* [21], which introduces a viscous second Piola-Kirchoff stress tensor $\mathbf{S}^{\nu}$ given by

$$\mathbf{S}^{\nu} = J\mathbf{F}^{-1}\boldsymbol{\sigma}^{\nu}\mathbf{F}^{-T}, \tag{3}$$

where J is the Jacobian of deformation, $\mathbf{F}$ is the deformation gradient tensor and

$$\boldsymbol{\sigma}^{\nu} = 2\,\Delta\eta\; dev(sym\dot{\mathbf{F}}\mathbf{F}^{-1}). \tag{4}$$

Here *sym* and *dev* respectively denote the symmetric and deviatoric components of a tensor and $\Delta\eta$ is the artificial viscosity coefficient defined by

$$\Delta\eta = \begin{cases} max(0, -\frac{3}{4}h\rho(c_1\Delta v - c_2 a)) & \Delta v < 0, \\ 0 & \Delta v \geq 0, \end{cases} \tag{5}$$

where h is the measure of the element size defined by $h = (Jd!|K|)^{1/d}$, d is the dimension of the space, $|K|$ is the volume of the element in its reference configuration, c_1 and c_2 are artificial viscosity coefficients, a is the characteristic sound speed of the material and is equal to the dilatational wave speed and $\rho = \rho_0/J$ is the mass density per unit deformed volume. Δv is the velocity jump across the element given by

$$\Delta v = h\frac{\partial log J}{\partial t}, \tag{6}$$

where t is time. The coefficients of artificial viscosity c_1 and c_2 are calibrated depending on the strength of the shock. Benson [22] discussed the procedures for determining these coefficients. It was proposed in [21] that the value of c_2 is typically in the range of 0.1 to 1 and the value of c_1 may be expected to remain close to 1 for strong shocks.

As mentioned earlier, we adopt in this work a mesh adaptivity criterion based on the scheme proposed by Diaz *et al.* [7], who used the error associated with the interpolation of the true solution using discrete finite element functions to identify the critical regions of the mesh. They also showed that, by distributing the interpolation error equally over the given mesh, the finite element solution can be made more accurate. An extension

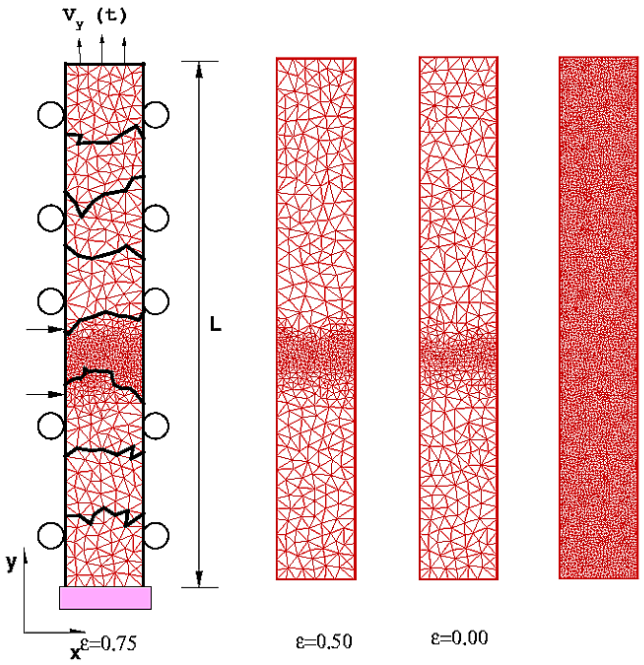

Fig. 2 Adaptive simulation of 1D wave propagation performed on 8 processors for three values of the cut-off parameter ϵ and for $\Delta t_{adaptive}/t_{ramp} = 0.25$. The adapted mesh is shown at time $t = 0.627L/C_d$. The left figure also shows the boundary conditions and the inter-processor boundary. The two horizontal arrows on the left denote the location of the leading and trailing edges of the wave front. The right-most figure shows the static fine mesh used for comparison in the present study.

of this method was proposed by Ortiz and Quigley [9], in which the variation of the velocity field was used as the error indicator in identifying the critical regions. The authors also proved that the equi-distribution of variation strategy minimizes the interpolation error. This approach was also used by Camacho and Ortiz [10] who defined a refinement indicator $\mathbf{I}_m$ based on the velocity gradient tensor, $\nabla\mathbf{v}$:

$$\mathbf{I}_m = ||\nabla\mathbf{v}||_m, \tag{7}$$

where $||\ ||_m$ indicates the $\mathbf{L}_2$-norm over the element m. The total interpolation error over the entire domain is then

$$\mathbf{I} = \sum_{m=1}^{n_{el}} \mathbf{I}_m, \tag{8}$$

where n_{el} is the total number of elements in the domain. As mentioned earlier, minimizing $\mathbf{I}$ is achieved by equidistributing $\mathbf{I}_m$ equally over the entire domain. In other words, introducing the average error measure

$$\mathbf{I}_{aver} = \frac{1}{n_{el}}\sum_{m=1}^{n_{el}} \mathbf{I}_m = \frac{\mathbf{I}}{n_{el}}, \tag{9}$$

we define the global normalized indicator on an element m as

$$\beta_m = \frac{\mathbf{I}_m}{\mathbf{I}_{aver}}. \tag{10}$$

Any element for which $\beta_m > 1$ is refined. Those with $\beta_m < 1$ are coarsened. Based on our experience, we have however adopted a more conservative approach to adaptive mesh modification, for which the refinement/coarsening limit for β_m is set at $1 - \epsilon$, where ϵ is a user-specified cut-off parameter ($0 \leq \epsilon < 1$). As shown in the next section, the choice of ϵ affects both the precision and computational cost of the numerical solution.

4 1D wave propagation

As a first assessment of the proposed parallel mesh adaptivity scheme, we study in this section the simple 1D wave propagation problem shown in Figure 2. The time-dependent y-velocity $V_y(t)$ applied along the top edge of the rectangular domain increases linearly with time t to its maximum value V_0 for $0 \leq t \leq t_{ramp}$ and is then kept constant, leading to the propagation of a planar dilatational stress wave traveling at a speed

$$C_d = \sqrt{\frac{E(1-\nu)}{\rho_o(1+\nu)(1-2\nu)}}, \tag{11}$$

where E, ν and ρ_o denote the material stiffness, Poisson's ratio and density, respectively.

The problem constitutes a good testbed for the mesh adaptivity scheme since it involves the rapid propagation of a relatively narrow region with high stress gradient (requiring a fine discretization) surrounded by regions of uniform stress distribution (where a coarse mesh can be used). In a traditional, non-adaptive approach, the accuracy of the solution could only be improved by increasing the density of the entire mesh, resulting in a large number of nodal degrees of freedom. In a dynamic adaptive setup, the critical region associated with the wave front is identified and the mesh can be selectively adapted. In adaptive simulations involving refinement and coarsening operations, the number of degrees of freedom remains lower than that of a uniformly fine mesh while achieving similar accuracy. The effect of the level and frequency of mesh adaptivity on the trade-off between accuracy and computational cost is the primary focus of this study.

In the numerical simulations presented in this section, the normalized velocity gradient β defined in the previous section is used to identify the critical regions. The variable parameters are the cut-off parameter ϵ and the frequency of mesh modification, i.e., the time interval $\Delta t_{adaptive}$ at which refinement and coarsening operations are conducted. The timestep size Δt is always kept below the value required by CFL stability condition [23]. The ramp time t_{ramp} is chosen as $C_d t_{ramp}/L = 0.125$,

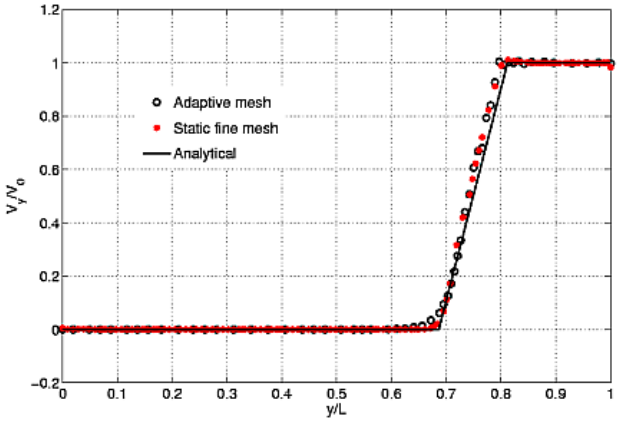

Fig. 3 Vertical velocity distribution along $x = 0$ at $C_d t/L = 0.3135$ obtained analytically (solid curve) and numerically (symbols) for the 1D wave problem shown in Figure 2. The mesh adaptivity parameters are $\epsilon = 0.75$ and $\Delta t_{adaptive}/t_{ramp} = 0.25$

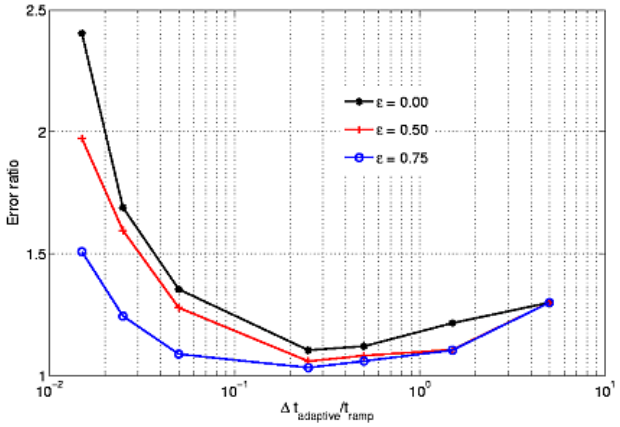

Fig. 4 Comparison between the numerical error on the y-velocity field associated with the adaptive and static simulations for the 1D wave propagation problem shown in Figure 2: effect of the mesh adaptivity parameters ϵ and $\Delta t_{adaptive}/t_{ramp}$. The error defined by (12) is computed at $C_d t/L = 0.314$.

where L denotes the domain length. To allow for a direct comparison between adaptive and non-adaptive simulations, the minimum element size used in the refinement studies is chosen equal to the average element size of the fine uniform mesh, while the maximum element size used in the coarsening process is set to the average element size of the initial coarse mesh.

Figure 2 shows adaptive meshes obtained for three values of ϵ ($\epsilon = 0, 0.5, 0.75$) at $C_d t/L = 0.627$. The left-most figure also illustrates the inter-processor boundary although no attempt was made to achieve dynamic load balancing in this case. The 8-processor run was simply conducted here to illustrate the ability of ParFUM to refine and coarsen the mesh adaptively across processor boundaries. The frequency of mesh modification was chosen as $\Delta t_{adaptive}/t_{ramp} = 0.25$. As apparent

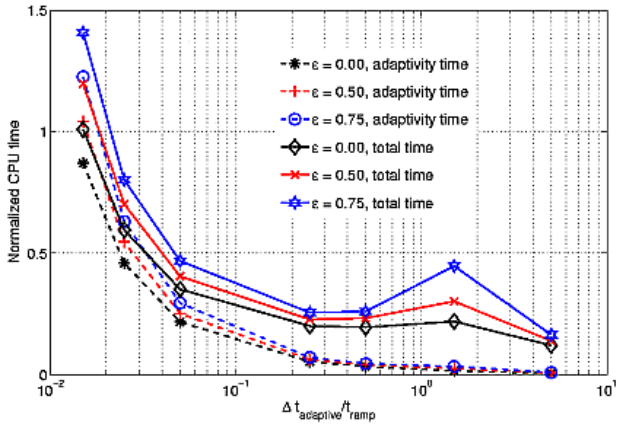

Fig. 5 Effect of the mesh adaptivity parameters ϵ and $\Delta t_{adaptive}/t_{ramp}$ on the computational cost of the adaptive simulations for the total time (solid curves) and the CPU time dedicated to the mesh refinement and coarsening operations (dashed curves) for the 1D wave propagation problem shown in Figure 2. The CPU times are normalized by the total CPU time associated with the fine static mesh simulation.

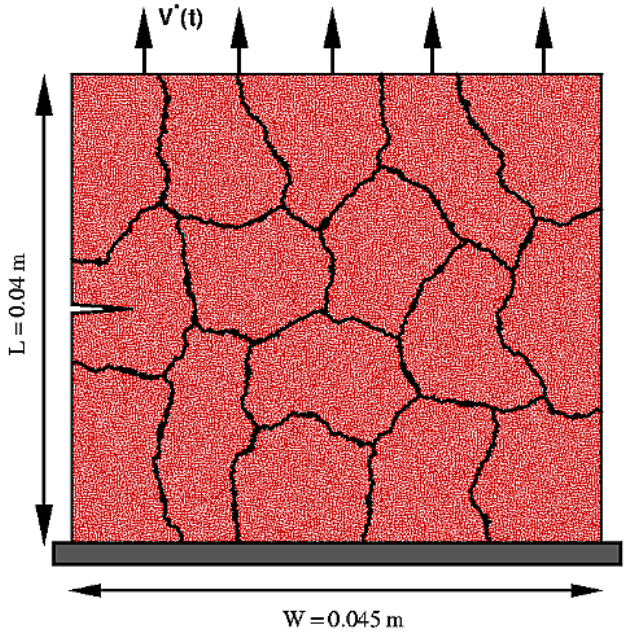

Fig. 6 Initial mesh, boundary conditions and inter-processor boundaries for the parallel adaptive finite element simulations of a dynamic fracture event. The initial crack length $a_0 = 0.005m$.

in Figure 2, a smaller value of the cut-off parameter ϵ leads to a sharper mesh refinement in the vicinity of the wave front, whose leading and trailing edges are indicated by horizontal arrows. The ability of the adaptive solution to capture the traveling wave front is illustrated in Figure 3, which presents a snapshot of the vertical velocity component along the edge $x = 0$ at $C_d t/L = 0.314$ obtained analytically (solid curve) and numerically with the adaptive (open symbols) and fine static (closed symbols) meshes. For this particular adaptive simulation, the parameters for mesh adaptivity were chosen as $\Delta t_{adaptive}/t_{ramp} = 0.25$ and $\epsilon = 0.75$. Overall, the y-velocity profile associated with the adaptive mesh is in close agreement with the numerical solution associated with the fine static mesh and with the analytical solution. However, the adaptive finite element does not fully capture the steep rise in the wave front due to the viscous smoothing taking place during mesh modification.

To quantify the effect of the mesh adaptivity parameters on the accuracy of the adaptive finite element solutions, we introduce the following time-dependent error measure in the y-velocity field:

$$Error = \frac{\int_0^L |(v_{y(adaptive)} - v_{y(analytical)})|\, dy}{\int_0^L |(v_{y(fine)} - v_{y(analytical)})|\, dy}, \qquad (12)$$

where $v_{y(adaptive)}$, $v_{y(fine)}$ and $v_{y(analytical)}$ respectively denote the y-velocity distribution corresponding to the adaptive mesh, fine static mesh and analytical solutions along the edge $x = 0$. The combined effect of the frequency in mesh modification (presented in terms of $\Delta t_{adaptive}/t_{ramp}$) and the cut-off parameter ϵ is shown in Figure 4 at time $C_d t/L = 0.314$. As apparent there, a larger value of ϵ leads to a smaller error ratio, as a larger refinement zone is introduced in the vicinity of the traveling wave front. Another key observation derived from Figure 4 is that there appears to be an optimum value of the mesh modification frequency for which the relative error is minimized for all values of ϵ. The more frequently the mesh repair process is introduced, the better the fine adaptive mesh is able to track the moving shock front. However, below the optimum value frequency of mesh modification, $\Delta t_{adaptive}/t_{ramp} \simeq 0.25$ in this case, the precision of the adaptive numerical solution starts to degrade due to the additional viscous damping introduced to smooth out the numerical oscillations associated with the rapidly changing mesh. This spurious numerical damping had been alluded to in the discussion pertaining to Figure 3.

Figure 5 addresses the issue of computational cost and presents the effect of the mesh adaptivity parameters on the CPU time associated with the entire simulation (solid curves) and with the mesh modification operations (dashed curves). The CPU times have been normalized by the total time associated with the fine static mesh simulation performed on the right-most mesh in Figure 2. As expected, the computational cost increases with the frequency of mesh modification (i.e., as $\Delta t_{adaptive}/t_{ramp}$ decreases) due to the increased cost of mesh refinement and coarsening. However, a deviation from this trend is observed for very low frequencies of mesh modification, where the quantity of elements in the adapted mesh and the corresponding total computa-

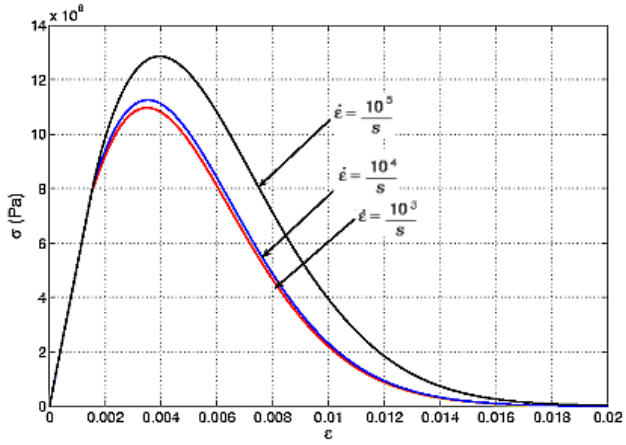

Fig. 7 Stress-strain curve under uniaxial state of strain obtained for $p_1 = 10$, $p_2 = 0.8$, $Y_{in} = 6000J/m^3$ and for various strain rates $\dot{\epsilon}$.

tional cost increase due to the infrequent mesh modifications. The results presented in Figures 4 and 5 indicate that, for $\epsilon = 0.75$ and $\Delta t_{adaptive}/t_{ramp} = 0.25$, we can achieve a precision comparable to that obtained with the static fine mesh (with a relative error ratio of about 1.03) for about one fifth of the total computational cost. The efficiency of the adaptive scheme is expected to increase further for larger problems, such as the dynamic failure problem in the next section.

5 Dynamic failure

5.1 Problem description

To assess the ability of the parallel adaptive scheme to capture dynamic fracture problems, we now turn our attention to the structural problem shown in Figure 6. An initially quiescent pre-notched compact tension specimen, with length $L = 0.04m$, width $W = 0.045m$ and an initial crack length $a_0 = 5mm$, is subjected to an imposed vertical velocity $V^*(t)$ applied along its upper edge. The imposed velocity history is similar to that used in the previous section: V^* increases linearly with time up to a value $V_0 = 2.5m/s$ attained at $C_d t_{ramp}/L = 0.01$, and is then kept constant. The lower boundary is held fixed, while the left and right edges and the crack faces are traction free. These loading conditions lead to the downward propagation of a 1D elastic wave which is diffracted by the crack and creates a region of high stress concentrations in the vicinity of the crack tip. The amplitude of the stress wave ($\sigma_0 = \rho_o C_d V_0$) is chosen such that the material initially behaves in an elastic fashion, with its response defined by the stiffness $E = 3.45GPa$, Poisson's ratio $\nu = 0.35$ and density $\rho_o = 1190kg/m^3$. The analysis being conducted under plane strain conditions, the associated dilatational wave speed C_d, defined by (11), is equal to $2090.4m/s$.

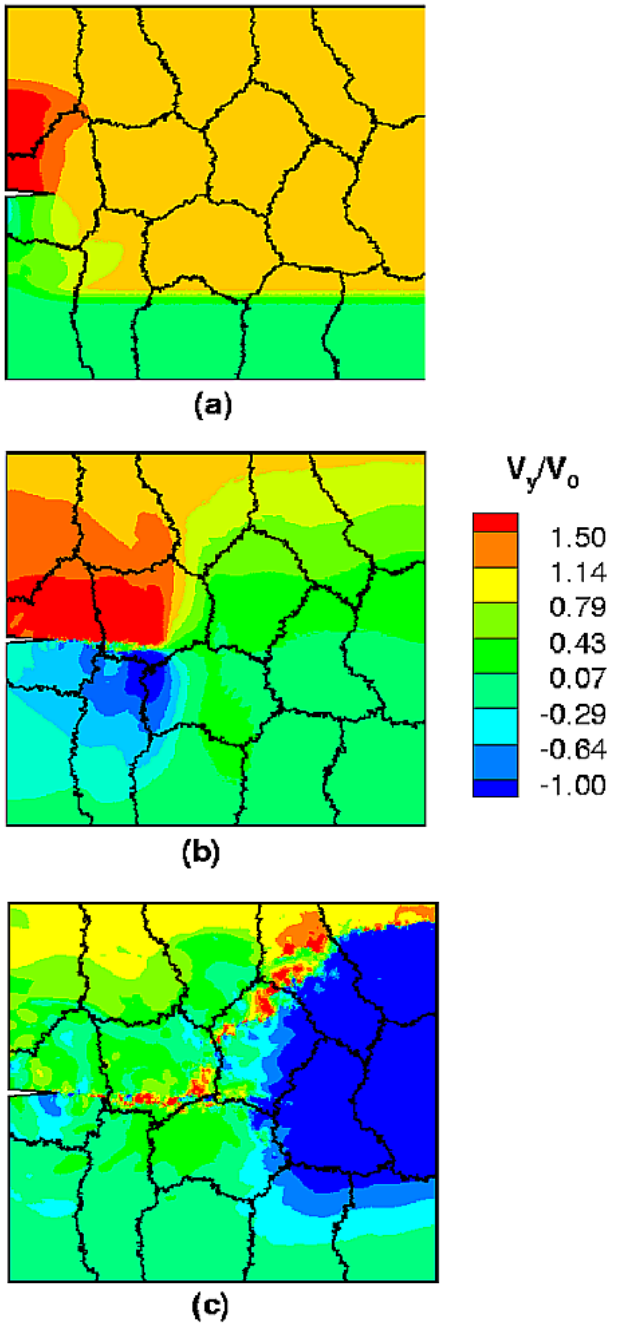

Fig. 8 Normalized vertical velocity V_y/V_0 distribution at times (a) $t = 0.783L/C_d$, (b) $t = 2.613L/C_d$ and (c) $t = 4.180L/C_d$.

To capture the initiation and dynamic growth of the crack, we use in the present study a simple isotropic rate-dependent damage model inspired from Ju [24], which is based on the following expression of the free energy function Ψ:

$$\Psi(\boldsymbol{\epsilon}^e, \omega) \equiv (1 - \omega)\Psi^0(\boldsymbol{\epsilon}^e), \tag{13}$$

where ω denotes the scalar damage parameter, and Ψ^0 is the undamaged free energy, a function of the elastic

strain tensor $\boldsymbol{\epsilon}^e$. The Clausius-Duhem inequality leads to the following expression of the stress:

$$\boldsymbol{\sigma} = \frac{\partial \Psi(\boldsymbol{\epsilon}^e, \omega)}{\partial \boldsymbol{\epsilon}^e} = (1-\omega)\frac{\partial \Psi^0}{\partial \boldsymbol{\epsilon}^e}, \tag{14}$$

where, for a linearly elastic solid,

$$\Psi^0 = \frac{1}{2}\boldsymbol{\epsilon}^e : \mathbb{C} : \boldsymbol{\epsilon}^e. \tag{15}$$

In equation (15), $\mathbb{C}$ is the fourth-order elastic stiffness tensor. Denoting $\xi \equiv \Psi(\boldsymbol{\epsilon}^e, \omega)$ for convenience, we introduce the damage criteria

$$g(\xi, \varkappa^t) = G(\xi) - \varkappa^t \leqslant 0, t \in \Re_+, \tag{16}$$

where the damage function $G(\xi)$ is given by the Weibull distribution

$$G(\xi) = 1 - exp\left[-\left(\frac{\xi - Y_{in}}{p_1 Y_{in}}\right)^{p_2}\right], \tag{17}$$

and $\varkappa^t$ denotes the damage threshold at time t. In (17), the initial threshold Y_{in} and the quantities p_1 and p_2 are material parameters. The damage model is completed by the Kuhn-Tucker relations that define consistent loading and unloading conditions as:

$$\dot{\varkappa^t} \geq 0, \quad g(\xi; \varkappa^t) \leq 0, \quad \dot{\varkappa^t} g(\xi, \varkappa^t) = 0. \tag{18}$$

Figure 7 presents the associated uni-axial stress-strain curve obtained with $p_1 = 10$, $p_2 = 0.8$ and $Y_{in} = 6000 J/m^3$ and for three values of the strain rate $\dot{\epsilon}$. As apparent there, after an undamaged linearly elastic response, the material exhibits a strongly nonlinear softening behavior, which is rate dependent for strain rates exceeding $10^4/s$. This softening response leads to a strain localization which in this work is attributed to the propagation of a crack.

5.2 Adaptive simulation

In the numerical simulation of the fracture event, we fix the cut-off parameter ϵ introduced in Section 3 at 0.25, and the frequency of mesh modification $\Delta t_{adaptive}/t_{ramp} = 10$, which corresponds to a mesh modification operation every 2000 timesteps. The initial mesh, shown in Figure 6, is composed of 50,000 3-node elements and is partitioned among 16 processors, each with 10 virtual processors. The initial size of each virtual processor is thus approximately 300 elements.

Snapshots of the y-velocity distribution are presented in Figure 8. In Figure 8(a), for which $t = 0.783L/C_d$, the downward traveling plane wave is clearly visible, together with the diffracted wave emanating from the crack tip. The strain localization originating from the crack tip is seen in Figure 8(b), for which $t = 2.613L/C_d$. This leads to a sharp velocity gradient across the crack resulting in positive and negative velocities above and below the crack, respectively. The last figure (Figure 8(c) obtained at $t = 4.180L/C_d$) shows the crack branching towards the upper edge of the specimen, with the sharp discontinuity in the velocity field now in the vicinity of the upper right corner of the domain.

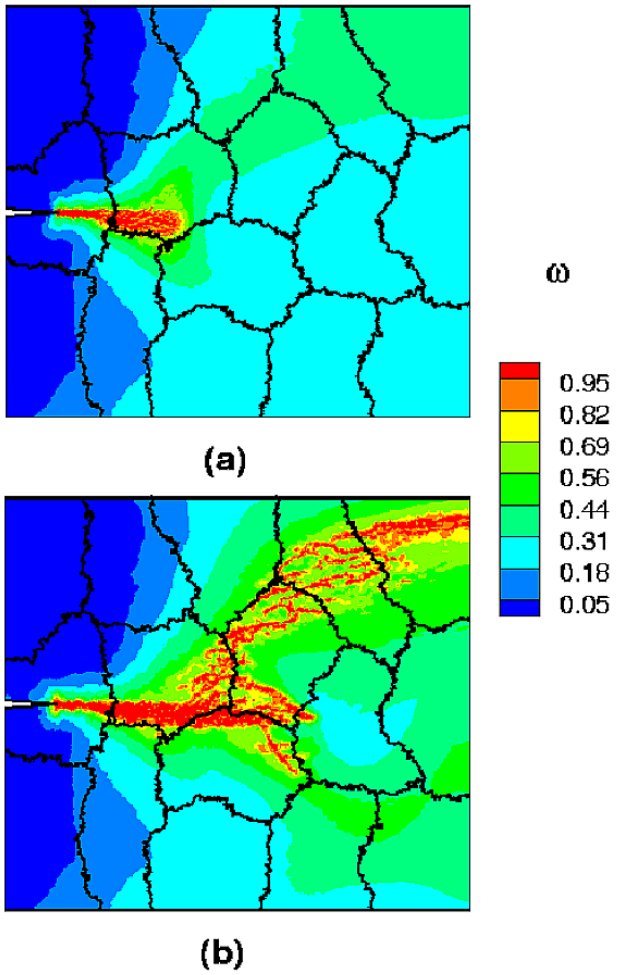

Fig. 9 Damage parameter ω distribution for the dynamic fracture problem shown in Figure 6 at (a) $t = 2.613L/C_d$ and (b) $t = 4.180L/C_d$.

The initial straight propagation and subsequent branching of the crack are further illustrated in Figure 9, which shows the evolution of the distribution of the damage parameter ω introduced in (13). In Figure 9(a), which corresponds to $t = 2.613L/C_d$, we observe the widening of the damage zone as the crack propagates faster ahead of the initial notch tip [6]. Figure 9(b), for which $t = 4.180L/C_d$, clearly shows the branching of the crack towards the upper right-hand corner of the domain, which leads to the slow-down and eventual arrest of the straight branch of the crack.

To capture this complex failure process accurately and efficiently, the finite element mesh is adaptively refined and coarsened, as illustrated in Figure 10, which presents snapshots of the dynamically adapted mesh at times $C_d t/L =$ (a) 0.783, (b) 2.613 and (c) 4.180. The

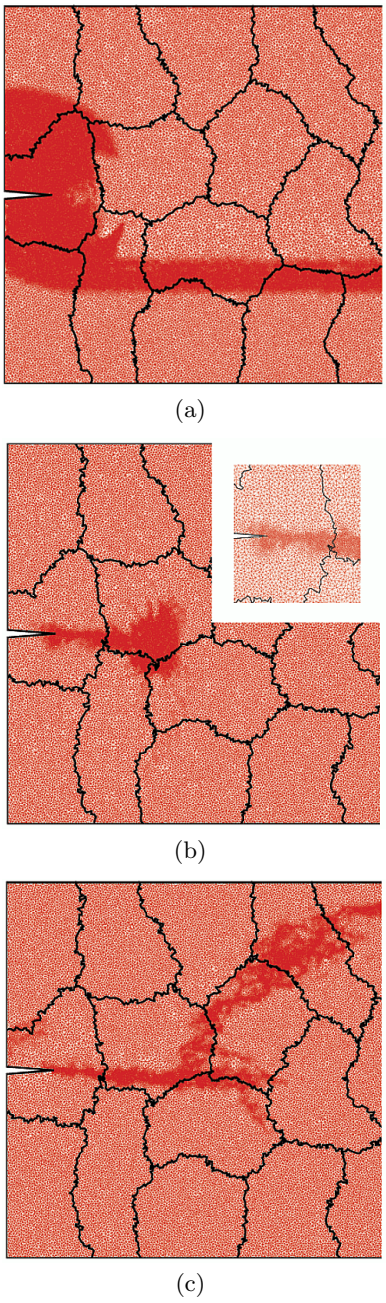

Fig. 10 Adaptive mesh for the dynamic failure problem shown in Figures 8 and 9 at (a) $t = 0.783L/C_d$, (b) $t = 2.613L/C_d$ and (c) $t = 4.180L/C_d$. The inset figure shows details of the refined mesh in (b) in the vicinity of the notch tip.

inset of Figure 10(b) shows a detail of the mesh in the vicinity of the initial notch tip. In this study, the ratio of the coarse to fine mesh was chosen to be approximately 10. The ability of the parallel adaptive framework to capture the diffraction of the planar wave (Figure 10(a)), the stress concentration associated with the propagating crack tip (Figure 10(b)) and the crack tip branching process (Figure 10(c)) is clearly observed.

6 ParFUM Performance

We have shown in Figures 4 and 5 that, in dynamic simulations, the use of ParFUM's adaptivity allows us to achieve considerable simulation accuracy in less time than when using a static fine mesh. In this section, we discuss the performance of ParFUM with adaptivity on the dynamic failure problem. We illustrate how virtualization affects performance and show how the problem scales relative to our best single processor time. We show the results of experimenting with Charm++'s built-in automatic load balancing to improve the performance and scaling of this simulation.

6.1 Benefits of Processor Virtualization

As discussed in Section 2, the concept of processor virtualization in Charm++ provides several benefits to ParFUM applications. The first of these benefits is that virtualization allows for more flexible decomposition of problems into more partitions than there are physical processors. Each partition of the mesh is associated with a user level AMPI thread. Low context switch time and low memory overheads for user level threads [25] mean that multiple partitions on a single processor can be maintained without paying a significant performance penalty. This ability to decompose a mesh into more partitions than there are physical processors makes it possible to achieve an overlap of computation and communication as discussed earlier.

However, there is of course some additional cost to this flexibility. Table 1 shows the overhead of adding additional virtual processors for single processor executions of the dynamic failure application. As the problem domain is broken up into more chunks, more and more messages need to be exchanged in each iteration of the computation. Although these messages are not inter-processor messages they do add some overhead. Since there can be no adaptive overlap between computation and communication in a single processor run, we see very little benefit from virtualization. Thus, single processor runs give us an idea of the cost of virtualization.

VPs	Time (10^3 s)	% Increase
1	7.9	-
4	8.4	6.3
8	9.2	16.5
10	9.7	22.8
16	10.7	35.4
24	11.7	48.1
32	12.0	51.8

Table 1 Execution times (in 1000 seconds) and percentage increase over single VP time for the dynamic failure application on a single processor for varying numbers of virtual processors.

The execution times shown in Table 1 make it apparent that the cost of virtualization is not insignificant. However, the same application run in parallel shows that the benefits of increasing virtualization far outweigh the costs. For the same application, we start to see improvements in run-time with more virtualization using just 4 physical processors. With 16 physical processors, these improvements are quite dramatic. Table 2 shows the same application running on 16 physical processors with varying number of virtual processors per processor. The improvement in performance with increased number of virtual processors is due to a better load balance among physical processors and adaptive overlap between communication and computation. As the number of virtual processors increases, the size of each virtual processor decreases to the point where most or all of it fits into the cache. This also can be an important factor in improving performance with more virtual processors.

VPs per processor	Time	% Decrease
1	1328	-
4	934	29.7
8	835	37.1
10	857	35.5
16	807	39.2
24	769	42.1
32	770	42.0

Table 2 Execution times (in seconds) and percentage decrease from single VP per processor time for dynamic failure application on 16 processors for varying numbers of virtual processors.

In our experiments with this application, it was unclear if a "sweet spot" was reached for virtualization on 16 processors because the time kept improving with the addition of VPs. However, many applications do exhibit a degree of virtualization at which they achieve optimal performance, and increasing the number of VPs beyond that only adds overhead.

The performance gains obtained when using virtualization depend on how this flexible decomposition is used to our advantage. In particular, the method used to initially map the partitions to the physical processors plays a large part in maximizing the potential for communication/computation overlap.

Our experiments with virtualization, initial mappings and load balancing were performed on a range of physical processors with varying degrees of virtualization. For this discussion, we focus on the case of 16 physical processors with 10 VPs initially placed on each processor. The mesh partitioning for this case is shown in Figure 11. The black lines indicate the partition boundaries, while each partition is shaded according to the processor it is on. Solution-directed mesh adaptivity is performed after every 2000 computational timesteps. This involves both refinement in the region of interest, and coarsening in areas no longer of interest. These regions change over time.

Our initial experiments made use of a block mapping of VPs to processors. This mapping, shown in Figure 12, is designed for minimal communication cost across processor boundaries. The dynamic failure problem has two phases of operation: the computation phase and the adaptivity phase. The computation phase is not communication intensive, so the block mapping does not benefit it greatly. To complicate matters, the adaptivity phase alters the loads dramatically on a subset of the partitions causing load imbalance. In a block mapping, these heavily loaded partitions are likely to be adjacent to each other, further exacerbating the load imbalance problem. This imbalance is demonstrated in Figure 13 which shows the utilization of each processor during the computation phase. The majority of the processors are underloaded achieving only about 55% utilization, while a few processors have higher utilization between 75 and 100%.

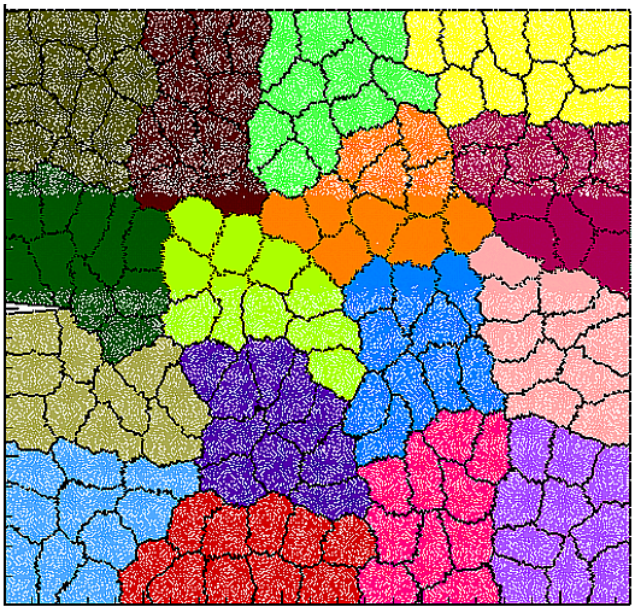

Fig. 11 The adapted mesh used in the dynamic fracture problem is divided among 160 virtual processors. The dark lines indicate partition boundaries. The region of higher mesh density indicates the location of the stress wave.

To make better use of the flexible mesh decomposition made possible via virtualization, we first choose a mapping that breaks the partitions up such that those that are adapted and thereby more heavily loaded are better distributed amongst the physical processors. Block mapping of VPs is the default behavior for AMPI, but several other mapping choices are available. In this case, the round-robin mapping was used. Round-robin mapping provides an initial placement for this mesh as shown in Figure 14. In the figure, each partition's adja-

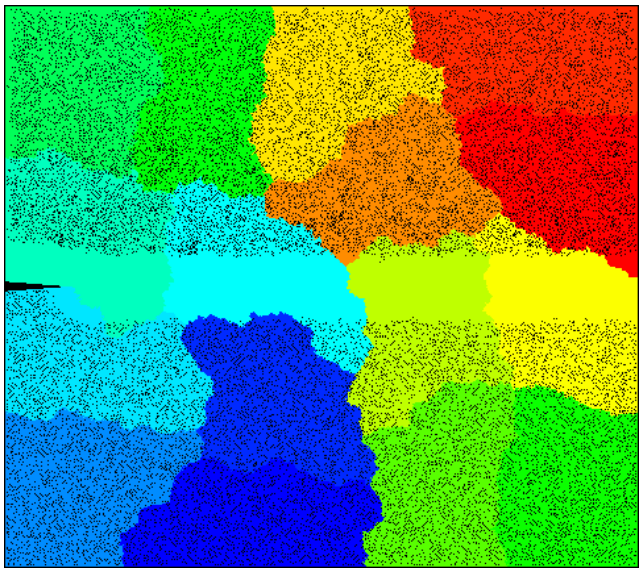

Fig. 12 Finite element mesh after 10,000 timesteps of the dynamic fracture problem. There are 160 virtual processors mapped to 16 processors using block mapping. Different shades indicate different processors.

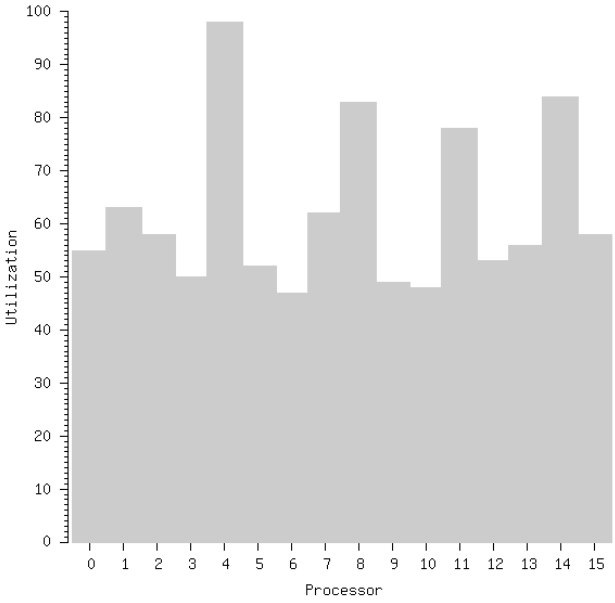

Fig. 13 Processor utilization in the computation phase of the dynamic fracture problem on 16 processors while using block mapping

cent partitions typically have a different shade, indicating that the partitions are on different physical processors. The execution times in Table 2 were obtained using the round-robin initial mapping.

Using the round-robin mapping improves the performance of the parallel application significantly by reducing the chances that several refined mesh partitions fall on the same physical processor. This improvement can

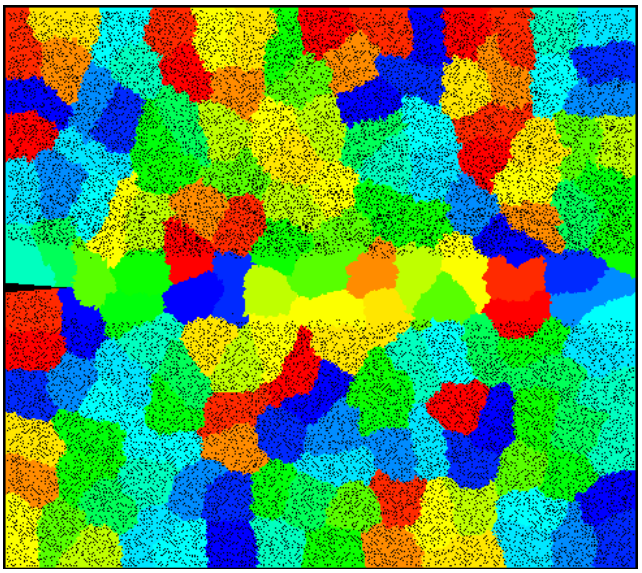

Fig. 14 Mesh after 10,000 timesteps with 160 virtual processors mapped onto 16 processors using round-robin mapping.

be seen in the utilization graph of Figure 15 but there is clearly still a significant load imbalance.

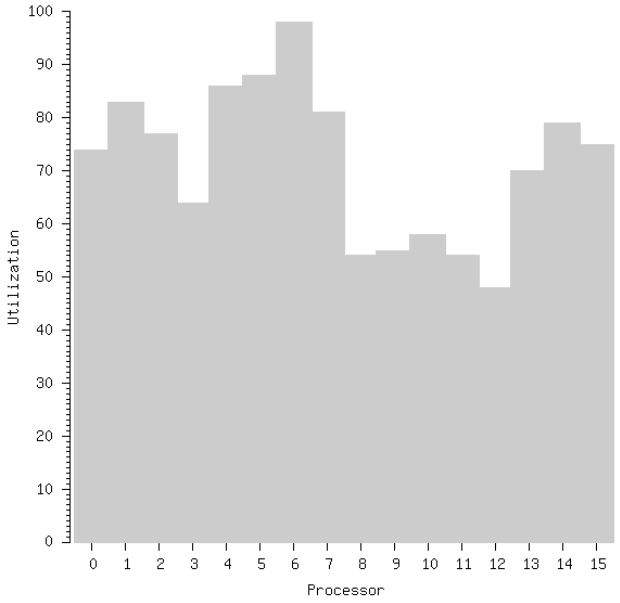

Fig. 15 Processor utilization in the computation phase of the dynamic fracture problem on 16 processors with round-robin mapping.

To handle the load imbalance caused by regional mesh refinement properly, we make use of another feature of Charm++/AMPI made possible by virtualization: automatic load balancing.

6.2 Load Balancing in ParFUM

ParFUM leverages the load balancing framework in Charm++ [26,27] to support automatic measurement-based dynamic load balancing. Our approach relies on the *principle of persistence* [16] that holds for most physical simulations where the computational load and communication structure of (even dynamic) applications tends to persist over time. This principle means that we can use the recent performance of an application to predict its performance in the near future. The load balancing framework instruments the run time system to automatically measure the computation load of each object and the pattern of communication between different objects. This information is stored in a load balancing *database*. After this information has been collected for some time, a load balancing step is initiated. At a load balancing step, this database is then used to determine if there is a load imbalance. If there is a load imbalance, the load balancing framework comes up with a new mapping of virtual processors to processors. The load balancing framework uses a spectrum of sophisticated load balancing strategies to decide on such a mapping. After it has decided on a mapping, the load balancing framework informs various processors about the virtual processors that they need to migrate away to other processors. Once all virtual processors have migrated to their new locations, the load balancing step is finished and the user code can resume.

When a partition migrates between processors, it must move all associated data, including those on its stack and heap. This is automatically achieved in the load balancing framework with *isomalloc stacks and heaps* [15] in a manner similar to that of PM^2 [28]. 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, *isomalloc*ed 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. As mentioned earlier, the user can alternatively write a function to both pack and unpack the heap data during a migration. This allows the application programmer to use application-specific knowledge to reduce the amount of data that needs to be packed during migration. The user can decide that some variables are not live during that stage of the application and do not need to be packed. It is especially helpful dealing with large meshes or meshes with a large amount of data. We use the approach of user-directed packing and unpacking in this application.

The next subsection shows how the load balancing framework was utilized to dramatically improve the performance of the dynamic fracture problem. It also describes the load balancing strategy that was used.

6.3 Handling Load Imbalance for the Dynamic Failure Problem

The dynamic failure problem presents us with unique load balancing challenges. The computation phase of the problem is almost trivially parallel for this problem because per-entity computation remains uniform throughout. However, due to the adaptivity, the number of entities per partition varies. Furthermore, with each new mesh adaptivity phase, the load on the partitions changes as the region of interest for solution accuracy changes over time. Thus, a new region or set of partitions will be refined at each adaptivity phase, and a previous region of interest will be coarsened.

We optimize via load balancing to improve the dominant computation phase of the simulation. Immediately after the completion of mesh adaptation, the Charm++ load measurement is started and used to instrument several timesteps of the computation phase to determine computational load and communication behavior. Then, a load balancing strategy is invoked which migrates partitions in such a way as to distribute the computational load evenly over the physical processors. Timestepping of the computation phase then continues with a mapping that produces a much better utilization of processors and thus performs superior to the initial round-robin mapping. Figure 16 shows how the mesh partitions are mapped after a load balancing step. It is not apparent from this view alone how this new mapping is better than the round-robin mapping, but Figure 17 shows that the mapping enables excellent processor utilization for the following computation phase. When the mesh is adapted again 2000 timesteps later, this mapping will be inadequate again. Thus load balancing is performed after every mesh adaptation.

As mentioned in Section 6.2, the Charm++ load balancing framework provides a number of sophisticated strategies to produce a mapping of virtual processors to physical processors that balances the load across all processors. Different strategies are suitable for different applications. For the dynamic fracture problem of interest here, we adopted the *greedy* load balancing strategy, which is a centralized load balancing strategy that collects the load balancing databases from different processors on one processor. While this approach might seem to incur a high overhead, it only amounts to sending few bytes per VP to one processor. Moreover, the cost of migrating VPs from one processor to another is much higher than the cost of collecting the database on one processor. The greedy strategy sorts all VPs in decreasing order of computation load. The strategy selects the first unassigned VP and assigns it to the least loaded processor. It repeats this step for each VP until every one has been assigned to a processor. The strategy uses a heap to find the least loaded processor at any given step. The overall computational complexity of the strategy can be calculated by looking at the different parts

of the strategy. Since the strategy uses heapsort to sort the VPs the complexity for the sort is $O(n \log n)$ where n is the number of objects. Building the heap of processors is $O(p \log p)$, where p is the number of processors. Selecting the least loaded processor every iteration and maintaining the heap is $O(\log p)$. So for n VPs the complexity is $O(n \log p)$. Since in a CHARM++ application with load balancing the number of virtual processors is frequently much higher than the number of processors ($n >> p$), the computational complexity of the strategy is $O(n \log n)$. This means that even for a large number of VPs, the computation time of the strategy is small.

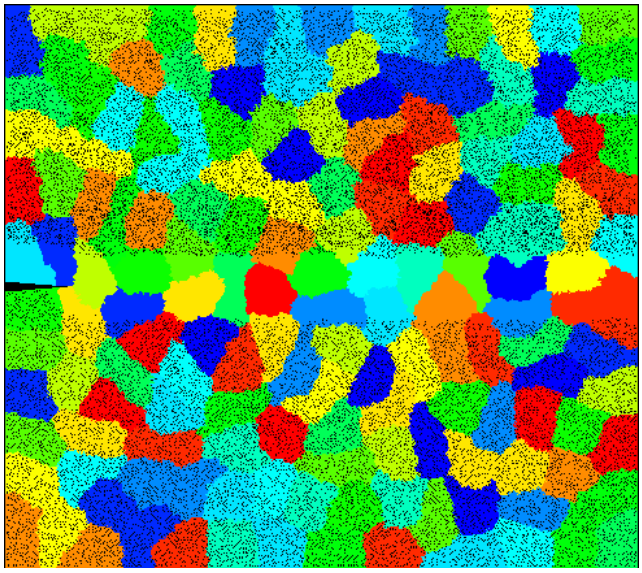

Fig. 16 Adapted mesh after 10,000 timesteps with 160 virtual processors mapped onto 16 processors. The greedy strategy was used to perform load balancing after the mesh modification phase. Different shades indicate different processors.

For this particular simulation, the refined regions move about in the mesh as seen in Section 5. The number of mesh entities also fluctuates over time. Figure 18 compares the performance of the various approaches as these variations occur over the first 20,000 timesteps of the simulation. We used a greedy strategy for our load balancing case, but also gathered results for a random load balancing strategy as a control. While the work required of the computation phase increases over time due to a steady increase in the number of mesh entities, the greedy load balancing strategy does the best job of keeping fluctuations in the mesh discretization from affecting the performance of the computation phase.

6.4 Scaling the Dynamic Failure Problem

In Figure 19, four scaling results are shown for the dynamic failure simulation run for 20,000 computational steps with mesh adaptation applied every 2000

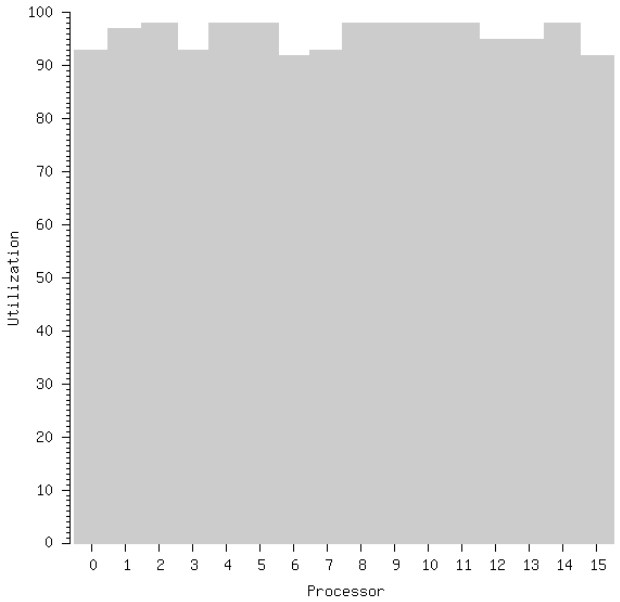

Fig. 17 Processor utilization in the computation phase of the dynamic failure problem on 16 processors after load balancing.

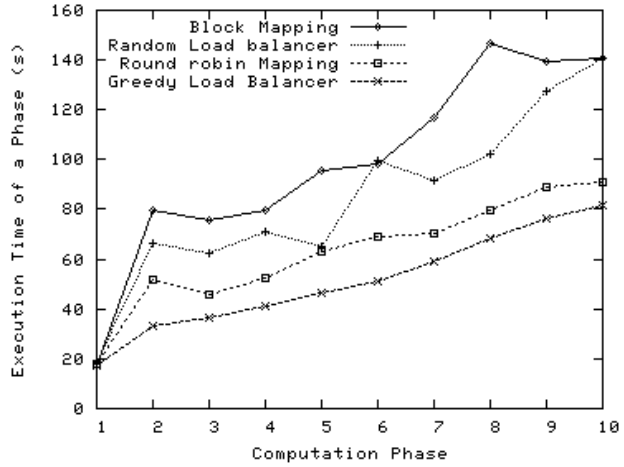

Fig. 18 Execution time for each computation phase of the dynamic fracture problem for block mapping, round-robin mapping, random load balancer and greedy load balancer. Each computation phase lasts for 2000 timesteps and is followed by a mesh modification phase. The data is shown for a run with 160 virtual processors on 16 processors.

timesteps. First, we show the results for the traditional approach of a single partition per processor. For the single processor run, a single VP exhibits minimal overhead over having additional VPs per processor, so this best single processor time is used as the sequential time that the other speedup curves are plotted against. The second set of results is for the round-robin initial mapping of partitions to physical processors. For these runs, the number of VPs per processor used was between 8 and 24. The third set of results is with the greedy load balancing

strategy, and the final set of results shows greedy load balancing coupled with pre-balancing for the adaptivity phase. The initial number of VPs per processor used for these experiments ranged from 4 to 24. We allow partitions per VP to vary because for larger numbers of physical processors, the total number of partitions becomes less manageable if we try to maintain the same number per processor as was used in runs on fewer physical processors.

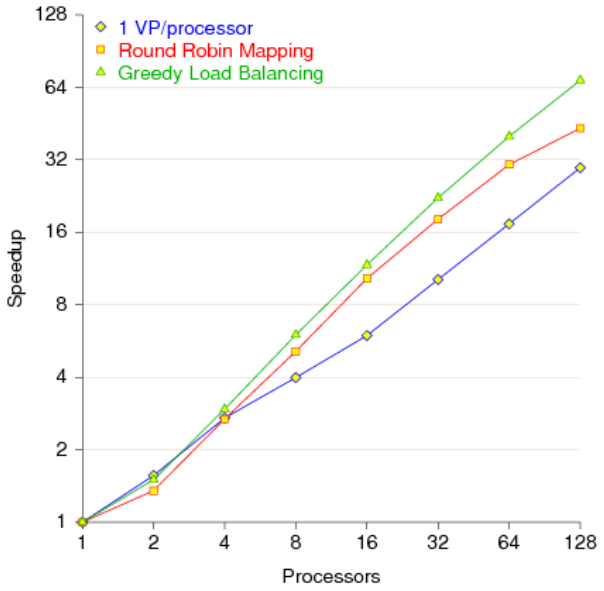

Fig. 19 Scaling results vs. best single processor execution time.

As can be seen from the figure, our best speedup obtained is 69 on 128 physical processors using the greedy load balancing strategy.

6.5 Load Balancing for Adaptivity

So far, we have discussed our techniques for improving the performance of the dominant computation phases of the simulation. We have seen how the load balance of these phases is adversely affected by adaptivity, and have corrected it using Charm++'s automatic load balancing. At this point, an adaptivity phase takes on average about 25% of the time of a 2000-timestep load-balanced computation plus adaptivity phase, as shown in Table 3. Since we ran the simulation for 20000 timesteps, there is no adaptivity phase after the last computation phase lasting from 18000 to 20000 timesteps.

As is apparent in Table 3, adaptivity consumes a significant portion of the total application execution time. The process of solution-directed mesh adaptivity starts by setting the mesh sizing according to current physical attributes on the mesh. Some regions of the mesh require refinement, some regions require coarsening, and some areas are unchanged. Thus, the activity in the mesh partitions is load imbalanced, such that the mesh partitions that require refinement or coarsening are heavily loaded and those that do not are completely idle. Further, the loading of the busy partitions depends on how much change is taking place. We also see considerable behavior differences from refinement and coarsening behaviors with respect to VP load. Finally, the obvious change is in the number of entities on a partition which dynamically changes throughout the process. All of these factors combine to create an extremely challenging load balancing problem.

Timesteps	Compute	Adapt	Total	% Adapt
1-2000	17.047	12.967	30.014	43.203
2001-4000	32.873	9.837	42.710	23.032
4001-6000	36.484	17.073	53.557	31.878
6001-8000	41.186	15.756	56.942	27.670
8001-10000	46.644	14.886	61.530	24.193
10001-12000	50.831	17.302	68.133	25.394
12001-14000	59.155	21.215	80.370	26.396
14001-16000	68.703	25.102	93.805	26.759
16001-18000	76.503	19.897	96.400	20.640
18001-20000	81.763	NA	81.763	NA

Table 3 Execution times (in seconds) of computation and adaptivity phases for the dynamic failure application on 16 processors with 10 VPs per processor.

We discovered that there is a significant load imbalance among processors during refinement. Figure 20 illustrates processor utilization on 16 processors during the refinement portion of the adaptive phase after 10000 timesteps of the dynamic fracture problem. We can see that utilization varies from 7% on processor 6 to 88% on processor 4. Figure 21 shows how average processor utilization of all 16 processors varies with time during the same refinement phase as shown in Figure 20. We can see that processor utilization falls drastically with time as more and more, but not all, processors run out of refinement operations to perform. This happens because most of the refinement operations are performed by a small number of virtual processors concentrated on a few physical processors. This imbalance provides us with an opportunity to significantly improve the performance of the refinement portion of the adaptive phase by distributing the load more equitably among processors.

Because of the rapid change in processor load over time and the relative shortness of the adaptivity phase, instrumentation of the phase is not useful. We instead adopted a new, highly experimental scheme of *prebalancing* according to application-specific instrumentation. As mentioned in Section 2.3, the user provides a desired mesh sizing to ParFUM before adapting the mesh based on properties of the physical solution. This sizing is used to determine the likely average load on each partition during the refinement portion of the adaptiv-

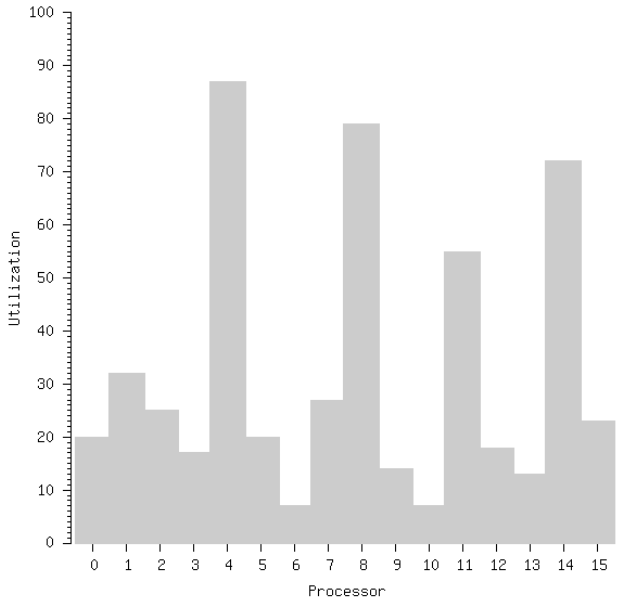

Fig. 20 Processor utilization during mesh refinement in the adaptive phase of the dynamic failure problem solved on 16 processors.

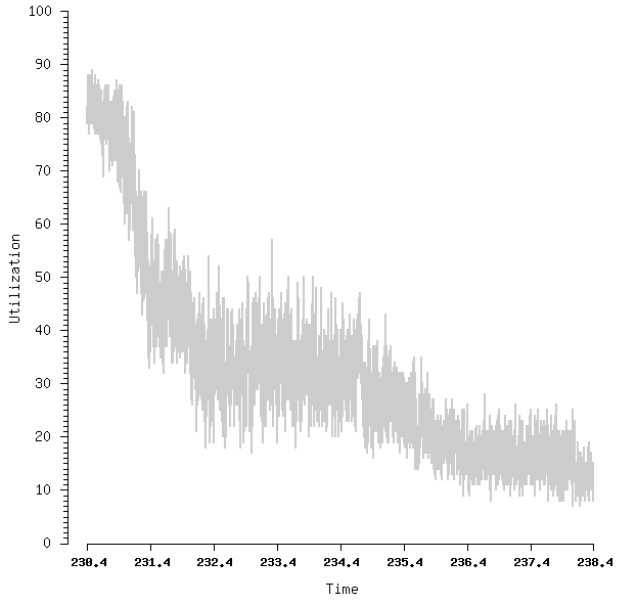

Fig. 21 Time evolution of processor utilization during mesh refinement in the adaptive phase of the dynamic failure problem solved on 16 processors.

ity phase. This load information is then passed to the Charm++ load balancing framework, and the greedy strategy is once again used to migrate the partitions. Once migration is completed, the adaptivity phase is started with a better mapping that exhibits better load balance than before. Figure 22 shows the improved utilization per processor due to pre-balancing for the refinement part of an adaptive phase. Although the utilization is not the same across all processors it is still much more uniform than in Figure 20. The effect of this more uniform load balance can be seen in Figure 23, which shows that after pre-balancing processor utilization during refinement does not demonstrate the dramatic decrease with time seen in Figure 21. During refinement, the average processor utilization over all 16 processors still decreases with time, but the drop is not as early or as fast as it is without pre-balancing. The benefit of the higher processor utilization obtained by pre-balancing can be seen in Table 4, which compares the time for refinement in each adaptivity phase with and without pre-balancing. It shows that pre-balancing improves the performance of refinement significantly in many adaptive phases. In some phases, pre-balancing nearly halves the time for refinement. Pre-balancing helps the performance of refinement when there is load imbalance caused by the fact that refinement occurs only on a small fraction of virtual processors. The benefit of pre-balancing is not as marked in phases where refinement happens on a larger fraction of virtual processors and is already more or less load balanced.

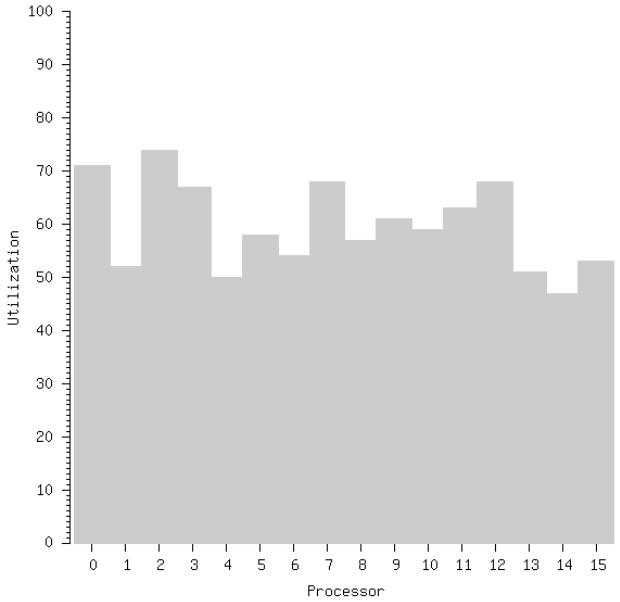

Fig. 22 Processor utilization during refinement in the adaptive phase on 16 processors after pre-balancing.

Although pre-balancing always reduces the time spent in the refinement portion of the adaptivity phase, we find that it does not always improve the total application runtime. In particular, Table 5 shows us that we achieve better performance improvements on fewer numbers of processors. Breaking down the execution times of the various components of the program before and after pre-balancing indicates the source of the problem. In Table 6, we discover that the mapping that works best for the refinement phase is detrimental to the coarsening

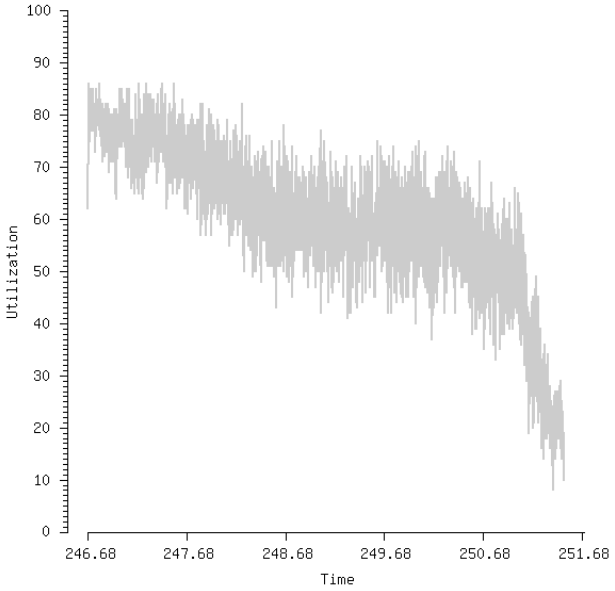

Fig. 23 Processor utilization over time during refinement in the adaptive phase of the dynamic failure problem on 16 processors after pre-balancing.

Adaptive phase	Time without pre-balancing	Time with pre-balancing
1	8.468	3.232
2	7.757	3.924
3	5.700	4.210
4	6.233	4.124
5	6.113	4.276
6	8.160	6.142
7	8.830	8.299
8	9.701	7.841
9	9.658	9.009

Table 4 Time taken (in seconds) by the refinement portion of the different adaptivity phases of the dynamic fracture problem on 16 processors. The simulation was run for 20000 timesteps.

P	Without Pre-balancing	With Pre-balancing
8	1319.573	1288.868
16	675.680	660.762
32	356.113	353.749
64	197.829	221.339

Table 5 Total runtime in seconds for 20000 timesteps of the dynamic fracture simulation with and without pre-balancing for varying number of processors P.

phase. This is to be expected because the regions of the mesh that were to be refined differ completely from those that were to be coarsened.

We are currently studying methods for pre-balancing the coarsening phase of mesh adaptivity. In future work, we will be exploring a technique for simultaneously refining and coarsening during the adaptive phase to avoid the idle time that results from the two phase approach. We will also explore techniques for pre-balancing overlapped mesh refinement and coarsening.

	W/o Pre-balancing		With Pre-balancing		
P	Refine	Coarsen	Refine	Coarsen	Cost
8	118.237	128.561	110.876	129.387	7.342
16	76.773	83.670	58.096	89.225	5.226
32	43.623	45.084	30.586	49.840	3.815
64	32.123	27.126	22.730	52.507	3.385

Table 6 Time spent in seconds on refinement and coarsening phases of adaptivity without and with pre-balancing during 20000 timesteps of the dynamic fracture simulation for varying number of processors P.

7 Conclusions

ParFUM, a Parallel Framework for Unstructured Meshes based on the Charm++ parallel run-time system, was enhanced for adaptive 2D finite element simulations of dynamic fracture events. The framework allows for the dynamic refinement and coarsening of the finite element mesh to capture rapidly propagating wave and crack fronts. Dynamic fine grained decomposition is achieved through encapsulating mesh partitions in virtual processors. Load balancing is accomplished by mapping those virtual processors to actual processors intelligently.

Two dynamic applications have been studied with this framework. The first of these was dedicated to the solution of 1D wave propagation. It aimed at assessing the effect of the mesh refinement parameters, such as mesh refinement frequency, on the precision and computational cost of the finite element simulation. The second application involved the use of a rate-dependent isotropic damage model for the simulation of the initiation and propagation of a crack in a pre-notched fracture specimen. This application demonstrated the ability of the parallel framework to modify the finite element mesh adaptively to capture the rapidly propagating crack.

We studied the benefits of processor virtualization and load balancing in the context of this second application. It was found that even without load balancing, processor virtualization improved the performance of the damage model simulation 30 to 40 percent on average. However, there was still significant imbalance among different processors during the computation phase of the simulation. We instrumented a few steps of computation after each adaptive phase and used the collected information to perform load balancing. This greatly improved the performance of the simulation and allowed us to scale a relatively small problem to a large number of processors. We achieved a speedup of 69 on 128 processors with greedy load balancing compared with a speedup of only 43 for a simple round robin initial mapping. We also

found severe load imbalance during mesh refinement in the adaptivity phases. Due to the short duration and rapidly changing load scenario of the mesh refinement phase, the "principle of persistence" is violated for this computation, making the use of measurement-based load balancing inappropriate. We developed an approach for *prebalancing* load, in which we used the estimated the cost of the refinement phase on each partition to redistribute the virtual processors among the physical processors before the start of the refinement phase. This significantly reduced the time spent in refinement and improved the overall application performance for a range of processors. However, for larger numbers of processors it degraded the performance of mesh coarsening.

We are currently studying the application of the prebalancing approach for coarsening. In the future, we will perform refinement and coarsening simultaneously to enable the overlap of idle times of one phase with the computation of the other. We will also investigate prebalancing in the context of this overlapped approach to adaptivity.

8 Acknowledgements

The authors gratefully acknowledge the support of NSF through grant EIA 01-03645 and of the Center for the Simulation of Advanced Rockets under contract number B341494 by the U.S Department of Energy.

References

1. Gao H, Klein P (1998) Numerical simulation of crack growth in an isotropic solid with randomized internal cohesive bonds. J. of the Mechanics and Physics of Solids 46:187–218.
2. Klein P, Gao H (1998) Crack nucleation and growth as strain localization in a virtual-bond continuum. Engineering Fracture Mechanics 61:21–48.
3. Camacho G T, Ortiz M (1996) Computational modelling of impact damage in brittle materials. Int'l J. of Solids and Structures 33:2899–2938.
4. Needleman A (1997) Numerical modeling of crack growth under dynamic loading conditions. Computational Mechanics 19:463–469.
5. Geubelle P H, Baylor J S (1998) Impact-induced delamination of composites: a 2D simulation. Composites Part B 29B:589–602.
6. Johnson E (1992) Process regions changes for rapidly propagating cracks. Int'l J. of Fracture 55:47–63.
7. Diaz A R, Kicuchi N, Taylor J E (1983) A method of grid optimization for finite element methods. Computer Methods in Applied Mechanics and Engineering 41:29–45.
8. Zienkiewicz O C, Zhu J Z (1987) A simple error estimator and adaptive procedure for practical engineering analysis. Int'l J. for Numerical Methods in Engineering 24:333–357.
9. Ortiz M, Quigley IV J J (1991) Adaptive mesh refinement in strain localization problems. Computer Methods in Applied Mechanics and Engineering 90:781–804.
10. Camacho G T, Ortiz M (1997) Adaptive Lagrangian modelling of ballistic penetration of metallic targets. Computer Methods in Applied Mechanics and Engineering 142:269–301.
11. Lawlor O, Chakravorty S, Wilmarth T, Choudhury N, Dooley I, Zheng G, Kalé L (2006) Parfum: A parallel framework for unstructured meshes for scalable dynamic physics applications. Engineering with Computers .
12. Kalé L V, Krishnan S (1996) Charm++: Parallel Programming with Message-Driven Objects. In G V Wilson, P Lu (eds.) Parallel Programming using C++, MIT Press, pp. 175–213.
13. Edwards H C, Stewart J R (2001) Sierra, a software environment for developing complex multiphysics applications. In K J Bathe (ed.) Computational Fluid and Solid Mechanics. Proc. First MIT Conf., Oxford, UK: Elsevier, Cambridge, MA, pp. 1147–1150.
14. Ollivier-Gooch C, Chand K, Dahlgren T, Diachin L F, Fix B, Kraftcheck J, Li X, Seol E S, Shephard M, Tautges T, Trease H (2006) The tstt mesh interface. In 44th AIAA Aerospace Sciences Meeting and Exhibit, vol. 529.
15. Huang C, Lawlor O, Kalé L V (2003) Adaptive MPI. In Proc. of the 16th Int'l Workshop on Languages and Compilers for Parallel Computing (LCPC 03), College Station, Texas, pp. 306–322.
16. Kalé L V (2002) The virtualization model of parallel programming : Runtime optimizations and the state of art. In LACSI 2002, Albuquerque.
17. Huang C, Zheng G, Kumar S, Kalé L V (2005) Performance evaluation of adaptive MPI. In PPL Technical Report 05-04.
18. Karypis G, Kumar V (1997) A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In Proc. of the 8th SIAM conference on Parallel Processing for Scientific Computing.
19. Rivara M C (1997) New longest-edge algorithms for the refinement and/or improvement of unstructured triangulations. Int'l J. for Numerical Methods in Engineering 40:3313–3324.
20. DeCougny H L, Shephard M S (1999) Parallel refinement and coarsening of tetrahedral meshes. International Journal for Numerical Methods in Engineering 46 7:1101–1125.
21. Lew A, Radovitzky R, Ortiz M (2001) An artificial-viscosity method for the Lagrangian analysis of shocks in solids with strength on unstructured, arbitrary-order tetrahedral meshes. J. of Computer-Aided Materials Design 8:213–231.
22. Benson D J (1992) Computational methods in Lagrangian and eulerian hydrocodes. Computer Methods in Applied Mechanics and Engineering 99:235–394.
23. Cook R D, Malkus D S, Plesha M E (1989) Concepts and Applications of Finite Element Analysis. John Wiley & Sons, fifth edn.
24. Ju J W (1989) On energy-based coupled elastoplastic damage theories: Constitutive modeling and computational aspects. Int'l J. of Solids and Structures 25:803–833.

25. Zheng G, Lawlor O S, Kalé L V (2006) Multiple flows of control in migratable parallel programs. In 2006 International Conference on Parallel Processing Workshops (IC PPW'06), IEEE Computer Society, Columbus, Ohio, pp. 435–444.
26. Zheng G (2005) Achieving High Performance on Extremely Large Parallel Machines: Performance Prediction and Load Balancing. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign.
27. Bhandarkar M, Kale L V, de Sturler E, Hoeflinger J (2001) Object-Based Adaptive Load Balancing for MPI Programs. In Proc. of the Int'l Conf. on Computational Science, San Francisco, CA, LNCS 2074, pp. 108–117.
28. Antoniu G, Bouge L, Namyst R (1999) An efficient and transparent thread migration scheme in the PM^2 runtime system. In Proc. 3rd Workshop on Runtime Systems for Parallel Programming (RTSPP) San Juan, Puerto Rico. Lecture Notes in Computer Science 1586, Springer-Verlag, pp. 496–510.