LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-408940

An analysis of options available for developing a common laser ray tracing package for Ares and Kull code frameworks

S. K. Weeratunga

November 21, 2008

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

An analysis of options available for developing a common laser ray tracing package for Ares and Kull code frameworks

Sisira Weeratunga
AX Division/LLNL

May 06, 2008

Abstract

Ares and Kull are mature code frameworks that support ALE hydrodynamics for a variety of HEDP applications at LLNL, using two widely different meshing approaches. While Ares is based on a 2-D/3-D block-structured mesh data base, Kull is designed to support unstructured, arbitrary polygonal/polyhedral meshes. In addition, both frameworks are capable of running applications on large, distributed-memory parallel machines. Currently, both these frameworks separately support assorted collections of physics packages related to HEDP, including one for the energy deposition by laser/ion-beam ray tracing. This study analyzes the options available for developing a common laser/ion-beam ray tracing package that can be easily shared between these two code frameworks and concludes with a set of recommendations for its development.

Contents

1 Introduction

This is a preliminary analysis of the options available for developing a laser/ion-beam ray tracing (LRT) package that can be easily shared between two or more widely different code frameworks designed to run a variety of high energy density physics (HEDP) applications at the Lawrence Livermore National Laboratory (LLNL). The two code frameworks under consideration are Ares and Kull. Each of these two code frameworks, separately support assorted collections of physics packages relevant to the HEDP domain. In both frameworks, these packages are designed to operate in concert with staggered mesh, arbitrary Lagrangian Eulerian (ALE) hydrodynamics modules. In the past, a majority of these physics packages have been designed and developed to be specific to the targeted code framework, without giving any considerations for being operational in other code frameworks. However, one relevant exception to this state of affairs is the work of Nick Gentile, who initially designed and developed an Implicit Monte Carlo (IMC) radiation transport package to be operational within the Kull framework, but later collaborated with others in successfully adapting that same IMC package to operate within several widely different HEDP code frameworks [1]. Based on this positive outcome, we are exploring the possibility of adopting a similar approach to developing a LRT package that can be operational within multiple code frameworks, potentially with a minimal loss of computational efficiency.

The significant computational similarities that exist between an IMC package and a LRT package forms the basis of why we believe such an approach is likely to be successful. It is our expectation that these similarities will also allow us to leverage heavily from the overall software design concepts/ideas used in the IMC package and to re-use some of the components already in use within the IMC package, with a little or no modification. In addition, given the similarities that exist in the load balancing requirements between the two packages on distributed memory architectures, it is our hope that the LRT package can adapt the same load balancing techniques and the related software infrastructure used in the IMC package. If all these assumptions hold true, we expect to realize a significant saving in the software development costs associated with a common LRT package for the two HEDP code frameworks.

It should be noted that this approach is not without risks. The primary risk factor is that of run-time performance of the common LRT package. It is likely that a LRT package developed in a manner that fully exploits all the salient features of a particular code framework will outperform a package that is intended to be operational in multiple frameworks. We currently do not have any reliable means available to us for estimating this potential loss in performance. However, it is our hope that this loss in performance would be within a tolerable range and will be mitigated to some extent by other benefits, both tangible and intangible, accrued through the adoption of this approach.

In the following sections, we will try to briefly outline some of the pertinent issues, based on our current state of knowledge of the task at hand. The section immediately below is a concise description of the features of the two HEDP code frameworks that have an immediate bearing on the design and implementation of a common LRT package. This is followed by a birds eye view description of the LRT packages that are currently operational within these two frameworks. In this section, we pay particular attention to the strengths and the weaknesses of the existing packages, since they both would have a strong influence on the approach chosen for the common LRT package. Next, we briefly describe our design goals for the common LRT

package and the need for trade-offs among the seemingly conflicting goals. Here, we focus specifically on the issue of performance optimization of the LRT package on large, distributed-memory parallel machines as well as the two main software development options available to us. Finally, we present our recommendations with regards to the strategy to be followed for the development of the common LRT package and a proposal for an implementation plan to be adopted in its development.

2 Host Code Frameworks

One the fundamental ways in which the Ares and the Kull frameworks differ is the types of meshes that are admissible and their respective internal representations. This in turn affects how the various combinatorial mesh entities are accessed and the mesh is traversed. These differences spill over into field data representation and access as well. A direct consequence of such differences is that the two frameworks have very different storage costs and computational efficiencies for a majority of operations involving the mesh and/or the field data.

2.1 Ares Framework

The Ares mesh data base supports a single-level, 2-D/3-D block structured meshes, i.e., locally structured but globally unstructured mesh topology. Within a block, the topological information storage is implicit, based on the logical (k,l,m) index space. All the necessary inter-block connectivity information is an integral part of the mesh data base. The basic cell topology is restricted to be either a quadrilateral in 2-D or a hexahedron in 3-D. However, each such logical cell can have several geometrical degeneracies, ex: collapsed edges and/or faces. In addition, due to ALE hydrodynamics, the cells can be highly deformed as well. For instance, the cells need not be convex; in fact, the presence of cells with moderate to severe concavity is highly likely in the Ares meshes. Also, the cell faces in 3-D hexahedrons need not be planar. More often than not, the faces of a hexahedron are 3-D quadrilaterals, with the potential for concavity as well. However, we are permitted to exclude the possibility of self-intersecting cells, i.e., cells where two non-adjacent edges or faces intersect. The block-structured nature of this mesh allows the use of such block boundaries in the partitioning of the global mesh into smaller sub-meshes. Hence, on distributed-memory parallel architectures, the Ares framework maps one or more of these logically structured blocks onto a single process and each such process is in turn assigned to a single CPU. The Ares framework also supports the definition of various types of distributed field data on this block-structured mesh, with differing centering possibilities as well as functions to access and operate on such field data types.

2.2 Kull Framework

The Kull mesh data base is capable of supporting unstructured, arbitrary polygonal meshes in 2-D and arbitrary polyhedral meshes in 3-D. All the topological information necessary for the mesh definition and traversal are explicitly stored as part of the mesh data base. This enables the Kull framework to be highly flexible with regards to the types of meshes that can be deployed, but this adaptability comes at a considerable price with respect to the storage costs and computational efficiency. In 2-D, basic cell topology is an arbitrary polygon with

N vertices, where N is greater than or equal to 3. This planar polygon can be either convex or concave and may have geometrical degeneracies such as collapsed edges. The arbitrary polyhedron that constitute a cell in 3-D has M faces where M is greater than or equal to 4, and depending on the value of M, such a cell may not be necessarily convex. Each face of the cell is a 3-D polygon with N vertices, where N is greater than or equal to 3. For N greater than 3, the facial polygon may be neither planar nor convex, in addition to having geometrical degeneracies. As in the case of the Ares, we are allowed to assume that the polygons and polyhedrons associated with the cells are not self-intersecting. When mapping this unstructured mesh onto multiple processes in a distributed-memory parallel environment, it is partitioned into more or less equal sized clusters of cells, that may or may not be disjoint, often based on the simplifying assumption that the computational load per zone is nearly the same. Each such cluster of cells is referred to as a domain and there exists a unique mapping between a domain and a computational process. A collection of such processes is then assigned to a set of CPUs through either a one-to-one or a one-to-many mapping. In any case, one and only one domain resides on a given CPU. The case of one-to-many mapping is generally referred to as domain replication. When there is only one domain for the entire mesh, it is called mesh replication. For the the IMC package operating within the Kull framework, the mesh/domain replication serves as the primary mechanism for achieving any semblance of load balance. It should be noted that, in general, the spatial distribution of photons is far from being uniform. Hence, the computational load per cell during the execution of the IMC package is highly non-homogeneous as well. Therefore, the canonical partitioning scheme used to decompose the mesh does not necessarily result in good load balance. Although there are several different approaches one can pursue to achieve a reasonable load balance in such situations, given the fully independent evolution of the individual photons, the domain replication is a viable first step. Similar to the Ares framework, Kull also supports the definition of various types of distributed field data on this unstructured mesh, with differing centering possibilities along with the functions needed to access and operate on such field data types.

While the mesh serves as an entity that differentiates the two code frameworks, the manner in which they support the multi-physics applications serves to unify them. Both frameworks are designed to use operator splitting to deal with multiple physics packages present in a single HEDP simulation. This is achieved through the fully-segregated, sequential time advancement of the individual physics packages through a physical time step, that is determined on the basis of the minimum allowable by all the active physics packages. A physics package, at the beginning of its time step, can choose to use the most recent state data fields available within the framework and at the end of that time step, leave behind the impact of its action by either updating the appropriate state data fields or as various source distributions that subsequently feed into other physics packages. This form of loose coupling of the physics packages facilitates the fully independent development of the individual packages and leaves the task of managing the data flow associated with the inter-package coupling in the hands of the code framework. Hence, the modular operational capability outlined above serves as the key enabler for developing common physics packages for the two frameworks. However, it should be noted that the operator splitting in this manner restricts the temporal accuracy of a multi-physics simulation to be at most first-order. This fact, combined with the other difficult to quantify stability restrictions associated with the operator splitting approach can sometimes significantly reduce the effective time step sizes of the multi-physics simulations.

3 Description of Existing LRT Packages

Both code frameworks currently contain functional LRT packages. However, the degree to which they have undergone real-user driven verification and validation, differ greatly between the Ares and the Kull packages; i.e., the package in Ares has been a participant in a substantial number of realistic simulations involving lasers, while that in Kull has not seen any use beyond the canonical laser-only test problems and a very limited set of laser-driven Lagrangian hydro problems. In addition, while both these packages have evolved from a common ancestor, i.e., the LRT package initially developed within the ICF3D code framework [2], the package currently operational within the Ares framework has been on a course that resulted in a far more aggressive addition of physics capability as well as computational performance enhancement. Consequently, as they currently stand, there are substantial differences between these two packages in almost every important aspect. In light of this, in the following sections, we provide a brief summary of the essential features of these two LRT packages.

3.1 Ares LRT Package

This package, which was first developed by Gary Kerbel to be part of the Hydra ICF code, has subsequently been adapted to be operational within the Ares framework. Although its genesis can be traced back to the LRT package within ICF3D code, Gary has made significant additions to its physics capability, beyond what was available in the original code [3]. However, what distinguishes it from other similar packages are the significant software modifications carried out to substantially enhance its computational performance. These modifications are primarily intended for achieving a better load balance during the LRT phase, on large distributed-memory parallel machines. The package essentially deploys three complementary approaches for reaching this goal:

1. Exploitation of the fact that each computational node of the targeted architecture is made up of multiple CPUs, all with high-speed shared memory access to data residing on that node.

2. Use of Posix threads to maximize the overlapping of computations and communications during the laser ray propagation.

3. Use of a genetic algorithm based scheme to enable the migration of block-structured domains among the available computational nodes, in order to achieve a more equitable distribution of the mesh cells involved in the processing of rays.

In the paragraphs below we summarize the pertinent details associated with each of these optimization approaches. The desire to exploit the shared memory access on a node is based on the assumption that threads spawned by a single process executing on a CPU belonging to the node will incur a lower latency and experience a higher bandwidth when accessing data residing on that node, in comparison to MPI-based process to process communication. However, due to the presence of CPUs with hierarchical memory and the need to preserve cache coherence among the CPUs on a node, it is not clear that the above assumption is always true. The degree to which this assumption holds true will ultimately be determined by the hardware and software protocols used to enforce the cache coherency as well as the nature

of the shared data updating patterns used by the threads executing on a node. Nevertheless, from the perspective of the LRT package, this requires that the block-structured meshes and the associated field data that are initially distributed among the distinct processes running on that node be agglomerated into one process. This is accomplished by requiring all except one process on that node to send the mesh and the field data relevant to the LRT package to that one process through MPI process to process communication. At the end of this operation, one process holds in its address space, all the mesh and the field data needed for LRT within that node. A side effect of this merging of the distributed mesh blocks onto the memory space of one process is the formation of a larger spatial domain, i.e., a super-domain, thus effectively increasing the granularity of the domains associated with the LRT package. A direct result of this increase in granularity is the reduction of inter-process data communication needs during ray propagation. However, the exact nature of the upside associated with this fact depends on the spatial distribution pattern of the rays during their propagation, which in turn is problem-dependent. It should be noted that this domain agglomeration process is greatly facilitated by the block structured nature of the underlying mesh and the field data. The case of fully unstructured meshes will prove to be far more challenging, especially with regards to the memory usage.

Once the agglomeration process is completed, every active process on the node other than the one holding the super-domain, is temporarily deactivated, leaving only one active process on that node. This active process then creates M plus two Posix threads to perform various tasks associated with the ray propagation, based on a producer-consumer model. The user-specified number M is the number of threads dedicated to propagate rays within the super-domain associated with the node. The remaining two threads, referred to as collector threads, are assigned to perform tasks in support of the M threads used for ray propagation. This specific heterogeneous threading structure was chosen with the hope of maximizing the potential for overlapping the computations and the communications as well as minimizing the need for explicit mutex-locks during the ray propagation.

Each of the M ray propagation threads, as they become activated by the OS, picks up a pre-determined number of rays from a double-ended queue used for holding the rays waiting to be processed by the active process associated with the super-domain. As these rays are propagated through a sub-set of cells belonging to the super-domain, both energy and momentum are deposited in these cells and the ray states are updated. In order to avoid the need for placing mutex-locks whenever a given thread needs to update the total amount of energy and momentum deposited in a cell, a double-buffered queue (DBQ) is associated with each of the M threads performing the ray propagation task. Instead of directly updating the energy and momentum deposited in a cell, that information is placed in the write-buffer of the DBQ by the thread propagating the ray in question. Although no explicit mutex-locks are needed during this operation, the length of the write buffer may need to be extended. This buffer length extension needs to be handled by a Posix thread-safe library function, which in turn, certainly needs to place a lock before the requisite memory allocation takes place. Hence, though it appears that an explicit mutex-locking operation is avoided by the application code, a hidden mutex-lock is activated by the thread-safe library function. Nevertheless, by extending the write buffer in chunks, the total number of mutex-locks activated can be minimized, at the cost of increased memory usage. The same procedure is followed when an individual ray attempts to cross the current super-domain boundaries to enter one of the other super-domains

held by another active node, i.e., when the ray propagation task needs to be continued across multiple super-domains, all the pertinent information associated with that ray are placed in the write buffer of another DBQ associated with that thread.

The two collector threads are tasked with collecting the data available in the read buffers of the 2M DBQs and disposing them appropriately. The collector thread tasked to deal with the M DBQs containing the energy and the momentum deposition data begins the process by accessing each of the M DBQs in sequence. While visiting a DBQ, first it transfers the data in the read buffer, if any, to the appropriate energy and momentum deposition containers associated with the cells of the super-domain. Then it activates a mutex-lock and swaps the pointers that point to the read and the write buffers. At this instant, the old read buffer becomes the new write buffer and vice-versa. After all M DBQs have been visited in this manner, the entire process repeats itself, until all the rays associated with the simulation have been processed to completion.

The collector thread tasked to process the M DBQs associated with the rays that cross the super-domain boundaries performs in a similar manner. However, it is responsible for some additional tasks that are not relevant to the energy/momentum collector thread. After the thread has processed all the M DBQs, it has the knowledge of all the rays that needs to be forwarded to other active processes. It does so by posting a series of non-blocking MPI-send calls. Then it checks to see whether any incoming messages from other active processes are waiting to be received. These messages contain the rays forwarded by the other processes to be propagated on the super-domain held by this process. If there are such messages, they are unpacked and those rays are added to the double-ended queue holding the rays waiting to be propagated by this process. After this, the thread returns to clear the buffers associated with the non-blocking MPI-send calls posted earlier. At the end of this cycle, the thread returns to processing the M DBQs yet another around, if necessary. Hence, this thread has the additional responsibility of handling the inter-process MPI message traffic associated with the ray propagation.

Once all the rays associated with the problem have been propagated through the mesh to their natural conclusion, the M plus 2 threads are terminated. Then the set of processes that were placed on temporary inactive status at the beginning of the ray propagation phase are re-activated. Finally, the cell-centered energy and momentum deposition data collected during the ray propagation phase needs to be communicated to the re-activated processes holding the corresponding mesh blocks. This is accomplished via MPI-based process-to-process communications.

As a result of the DBQ approach described above, the number of explicit mutex-locks activated during the ray propagation is reduced while increasing the potential overlap between the computations and the communications. However, this is accompanied by an increase in the memory usage and more importantly, a tremendous increase in the overall code complexity due to the highly intrusive nature of the Posix thread management task necessary to choreograph the functioning of this heterogeneous thread collection. It is common knowledge that the highly-optimized thread based programs pose significant challenges to the task of detecting and correcting the known bugs. The conventional debugging tools provide very little help in this regard. Any attempts to reduce the coding complexity associated with the threading, thereby increasing the readability of the code, is likely to be met with significant reduction in computational efficiency. Consequently, the entire LRT package suffers from substantially reduced

maintainability and extensibility, especially in an environment where the original developer of the package may not be the only developer tasked with such responsibilities. In addition, as M increases, the scalability of the whole approach is in doubt due to the possibility that one or both of the two collector threads can become the bottleneck. Alleviating such a bottleneck may require a much more complex thread management strategy with an accompanying increase in coding complexity.

During the ray propagation phase, the spatial distribution of the computational load is highly non-uniform. However, it changes very slowly from one time step to the next. This fact, combined with the block-structured nature of the underlying mesh allows for the periodic deployment of a dynamic load balancing strategy for the LRT package. A genetic algorithm is used to obtain a near-optimal mesh block distribution by considering all the costs associated with the ray propagation and the mesh re-distribution processes. Since the mesh block re-distribution for better load balance can be carried out independently of the ray propagation task, the dynamic load balancing has little impact on the coding complexity of the LRT package itself and thus has no impact on the maintainability and the extensibility of the package. However, it's impact on the computational performance during the ray propagation phase can be significant.

As described in the paragraphs above, the three complimentary performance enhancing strategies could function in a synergistic manner to produce formidable performance gains for wide array of laser ray tracing problems. In many instances, notwithstanding its downsides with regards to the code maintainability and the extensibility, it is very likely that this approach may set an upper bound on the achievable computational performance for the LRT phase on the current generation of NUMA architectures used at LLNL. The pivotal unanswered question is: are there any other performance optimization approaches that do not share the above-mentioned downsides to the same degree, but are nevertheless capable of producing levels of computational performance substantially equivalent to that achieved by this thread-based approach.

In addition to its computational performance benefits, the Ares LRT package has other desirable features as well. The chief among these are its relative completeness with regards to the ray propagation physics and the acceleration of the search procedure for the first cell boundary face that intersects with a ray emanating from its point of origin. Nevertheless, it does have some well known missing features as well. The most important among these are:

1. True 2-D RZ ray tracing capability.

2. Inclusion of energy deposition through resonance absorption effects.

3. Graceful handling of ray propagation through occasional self-intersecting cells vs. mere assignment to the "lost rays" category.

The package also makes the assumption that all mesh cells are logical hexahedrons. Hence, all the ray propagation procedures used within the package makes explicit use of this fact by hard-coding the functions used, to conform to this particular cell topology. Within the Ares framework, this does not have any negative consequences. In actuality, this fact is central to the ray propagation approach used in the package, since all the cell faces are approximated by bi-linear functions for determining the intersection between a ray path represented by a

quadratic curve and a cell face. Therefore, the extension of this package to a framework such as Kull would require significant modification of all the affected functions.

The lack of 2-D RZ capability warrants further elaboration. In laser ray tracing, the rays always propagate in 3-D although the underlying mesh and the associated fields may posses 2-D RZ symmetry. In general, the LRT package assumes piecewise linear electron density profiles within mesh cells and consequently has piecewise constant electron density gradients within the same mesh cells. In 3-D, this greatly eases the solution of the second-order, vector-valued initial value problem (VIVP) associated with geometric optics approximation used for laser ray tracing. In fact, for 3-D meshes, under the above assumption for electron density distribution, the VIVP has an analytic solution within each mesh cell that is admissible over the entire cell. However, with the 2-D RZ geometry, this state of affairs hold only if the ray path is confined to the meridional plane and such an assumption is clearly unacceptable in most simulations [4]. If this assumption is relaxed, the electron density gradient is no longer constant as the ray traverses the 3-D space. In fact, the x and y components of the VIVP has electron density gradient terms that depend on the instantaneous position of the ray head. The resulting coupled, nonlinear form of the VIVP defies any attempts at an analytic solution and requires the use of a multi-stage numerical integration scheme with step-length control for maintaining an adequate accuracy of the ray path. In addition, one is required to determine the point of intersection of the quadratic ray trajectory with the conical surfaces that represents each side of the mesh cell. This in turn requires the accurate solution of a quartic equation, which in itself is a computationally challenging task in the presence of numerical ill-conditioning. Consequently, the ray propagation in 2-D RZ geometry is significantly more challenging than in 3-D geometry.

In order to avoid this complexity in 2-D RZ geometry, the Ares LRT package pursues an alternate approach for propagating rays in such a geometry. This approach is capable of producing only an approximate solution to the problem but nevertheless may be of sufficient accuracy for certain classes of problems. The method is based on the use of the 3-D ray propagation package in 2-D RZ geometries by creating a wedge-shaped 3-D mesh through the rotation of 2-D RZ mesh about the axis of symmetry. The resulting mesh has only one zone spanning the user-specified wedge angle and the size of that azimuthal angle has a direct influence on the accuracy of the solution. During the ray propagation, specular reflection boundary condition is applied at the cell faces that coincide with the side surfaces of the wedge, which also happens to be one of the two correct applicable boundary conditions when there is only one zone spanning the wedge angle. However, a more serious threat to the accuracy of the solution is posed by the need to impose an additional degree of the symmetry on the problem, due to the inherent nature of this 3-D wedge mesh. In 2-D RZ geometry, the rays emanating from the source are distributed over all azimuthal angles forming a hyperboloid of revolution in vacuum. However a ray that emanates at an azimuthal angle that lies outside the wedge-angle cannot be propagated by this 3-D wedge mesh. Hence to initiate the propagation process, such a ray is subjected to an artificial rotation about the axis of symmetry, until it lies on the mid-plane of the wedge. This rotation imparts an additional degree of symmetry to the problem and can have a non-negligible impact on the accuracy of the solution obtained through this approach.

3.2 Kull LRT Package

As it currently stands, in contrast to the Ares LRT package, the Kull LRT package makes no attempt whatsoever to achieve any degree of success with regards to the tasks of overlapping the computations and the communications as well as balancing the computational load during the ray propagation phase. As a result, the code is remarkably readable, even for someone with a little or no prior familiarity with the package, which is in sharp contrast to the state of affairs with the Ares LRT package. However, this lack of code complexity is certain to be accompanied by significantly lower computational performance, given the highly inhomogeneous distribution of the computational load and the unpredictable inter-process communication patterns encountered during the ray propagation process. Due to these realities with regards to the computational efficiency, it is very unlikely that the Kull LRT package in its current form would be usable for any realistic, multi-physics simulations.

In addition to its short comings with regards to computational performance, the Kull LRT package also suffers from the following deficiencies in its physics capabilities:

1. Lack of support for 2-D RZ geometry.
2. No accounting for diffraction effects, i.e., incoherent scattering processes, during the ray propagation.
3. Energy deposition due to resonance absorption effects not included.
4. Momentum deposition due to ray propagation is not computed.
5. No support for beam types with finite focal spot sizes and arbitrary focal plane power distribution patterns.
6. Very limited support for reflecting wall boundary condition types.

Besides the computational performance issues alluded to in the earlier paragraph, from an implementation point of view, the package fails to address the following concerns:

1. Accelerating the search process associated with the identification of the cell boundary face for initial ray entry.
2. Accounting for the possibility of rays re-entering the mesh after they have exited the mesh through a boundary face.
3. Proper handling of the triangulation process for concave cell faces with more than four vertexes.
4. Support for graceful handling of ray propagation through occasional self-intersecting cells vs. mere assignment to the "lost rays" category.

4 Design Goals for a Common LRT Package

It is apparent from the above analysis of the two existing LRT packages, that neither package in its current form, is in a position to take on the role of being the common LRT package due

to a variety of reasons. Hence, it is useful to examine what are the desirable traits of a common LRT package, without being unduly constrained by the two existing packages. This will be followed by the examination of the options available for realizing a common LRT package with such desirable traits.

Numerous literature on software engineering practices provide an extensive list of required as well as desirable properties for the packages/components used in scientific computing projects. In the context of this study, only a subset of such properties are relevant. Following is a list of such properties which we currently perceive as important, albeit not necessarily in the order of their importance.

Correctness/Accuracy: This is an essential property and addresses the following sub-issues.

1. Given a valid input, does the package deliver the correct output?
2. What is the software testing strategy?
3. Is the package numerically stable and is there an error control strategy?

Robustness: This is a required property and measures the following:

1. How stable is the package for a valid input?
2. How consistent is the error treatment upon the detection of an invalid input?

Computational Efficiency: Generally, this is characterized by the following:

1. Memory usage (temporary work space, need for copying of input data).
2. Runtime serial performance.
3. Parallel scalability.

Interoperability: Measures how well a physics package can cooperate with other packages and also how easily does it operate within diverse code frameworks.

Flexibility: Refers to the ability of the user to control parameters that influence the internal execution, debug level or the exchange of algorithmic blocks.

Maintainability: Measured by the ease with which to detect and correct known bugs, enhance or extend the package, without major modifications to the host framework.

Extensibility Measured by how easy is it to extend the functionality of the package, while keeping the host framework interface as intact as possible.

Stability: Given that most physics packages are not static entities, but evolve over time, stability measures the frequency of changes that impacts a host framework, such as interface syntax, pre- and postconditions, resource consumption. In general, it is acceptable if pre-conditions are weakened, post-conditions are strengthened, memory and time consumption decreases and interfaces get more general, leaving the use within existing frameworks valid.

It is very apparent that these desirable traits are sometimes in conflict with each other. Especially, the computational efficiency seems to contradict every other trait. However, in the scientific computing arena, the computational efficiency is frequently and rightfully assigned a great deal of importance. Therefore, very often compromises are made in favor of efficiency. This may be an acceptable trade-off for a physics package developed for a highly focused code project that involves perhaps a very small and stable software development team. However, if the intended goal is the development of a physics package that can operate relatively easily within multiple code frameworks that are developed and maintained by relatively large group of developers, while serving the needs of a diverse set of users/applications, then this lopsided trade-off in favor of the efficiency may not be judicious. Therefore we argue that an appropriate level of consideration should be given to other traits as well. In this regard, it is our opinion that, interoperability, maintainability and extensibility also should rank high, without compromising the accuracy and robustness. We are under no illusion that under this modified trade-off regime, we can still retain the maximum possible level of computational efficiency. However, it is our hope that by sacrificing a modest degree in efficiency, we can gain substantially in other desirable traits.

Unfortunately, there are no means available to reliably test this hypothesis prior to implementing the complete physics package under this new set of guidelines. In addition, while the computational efficiency can relatively easily be quantified, there are no readily accepted, objective metrics for traits such as maintainability and extensibility. Nevertheless, this approach is not completely without any basis. The existence of the Kull IMC package, coupled with its similarity to the LRT package in all the key computational traits that impact performance, provides us with at-least some degree of justification necessary to pursue this approach. It is already proven that the software design philosophy followed in the implementation of the Kull IMC package allows it to operate within a diverse set of code frameworks. The fact that more than one person is actively engaged in its development testifies to its maintainability and extensibility. This in no way argues that its design cannot be further improved. We fully admit that there may be opportunities available to further improve its design. However, there is one important missing piece of information, i.e., the degree to which the computational performance has been lost as a result of pursing the approach chosen for its implementation. Given the absence of any such concrete information, in the next section, we briefly describe performance optimization strategies available for the physics packages that share computational traits similar to the IMC package.

5 Performance Optimization Strategies for LRT

In this section, we consider the performance optimization strategies available when a LRT package is implemented under the following constraints:

1. The LRT package is only one of the components active within an integrated, multi-physics simulation framework. The computational characteristics of the active packages may share very little commonality with each other.

2. The entire simulation framework as a whole, is expected to operate with a reasonable efficiency on a large, distributed-memory parallel machine.

As a result, we are not in a position to consider the optimization strategies for the LRT package in isolation. The presence of other physics packages that are simultaneously active, imposes memory usage restrictions. The use of large, distributed-memory parallel machines necessitates the use of a spatial domain decomposition strategy for the shared mesh and field data, that has to take into consideration the needs of the entire spectrum of physics packages expected to operate within the integrated framework. Consequently, it is highly likely that one or more more physics packages might find the resulting domain decomposition strategy to be less than ideal for their respective computational needs.

Several inherent features of the ray propagation algorithms associated with LRT package further exacerbate this situation. They are:

1. Spatially inhomogeneous nature of the ray propagation problem producing a corresponding computational load imbalance among the spatial domains assigned to different processors.

2. Temporally diminishing size of the computational load over the ray propagation cycle, as the initial ray bundle gets depleted due to various sources of ray attrition.

3. The items (a) and (b) induce non-deterministic, inter-process communication loads as well as patterns.

Therefore, any performance optimization strategy targeted at a LRT package operating on distributed-memory architectures needs to be able to cope with these challenges. It should be emphasized that these computational characteristics are not limited to LRT packages. They are likely to appear in many other algorithms that involve particle-mesh interactions, ex; Implicit Monte Carlo radiation transport. However, the severity of these challenges may differ among the various physics packages.

The proven approaches, that are designed to deal with these challenges, fall broadly into two distinct categories. Both categories rely on the central algorithmic fact that each active particle/ray can be processed independently of all other active particles/rays. In other words, there are no ray-to-ray data dependencies, and hence the rays can be processed in an arbitrary order. The optimization strategy used in the existing Ares LRT package belongs to the first category. The details of this heterogeneous thread-based approach and its numerous advantages as well as disadvantages were discussed in a previous section.

The second approach to the performance optimization is typified by the domain replication strategy deployed in the Kull IMC package. Under this approach, the computational mesh is partitioned into a collection of non-overlapping sub-domains and each sub-domain is assigned to at least one unique process. The optimization criteria used for cell clustering during the partitioning of the mesh generally does not take into account the cell-wise computational load differences. As a result, the resulting domains may have widely differing computational loads, especially when certain physics packages that involve particle-mesh interactions are processed. To alleviate this load imbalance, domains with heavier particle-related computational burdens are redundantly mapped onto more than one process. In general, the number of processes associated with a domain should increase in direct proportion to its computational load. However, the computational resource limitations in a realistic run-time environment may limit such complete flexibility with regards to domain replication. In any case, the replicated domains are

assigned unique chunks of the particle stack for processing and the energy and momentum deposition statistics are appropriately combined after all the particles have reached census. This load-balancing approach is supplemented by a non-blocking inter-processor message passing scheme [5]. Such an asynchronous message passing scheme facilitates the overlapping of the computations and the communications at-least to some degree, thereby partially mitigating the adverse effects of the non-deterministic, inter-processor communication loads and patterns. As is the case with any approach, there are numerous advantages and disadvantages associated with this approach as well. Among its advantages vis-a-vis the thread based approach are;

1. Greatly reduced software complexity of the physics package, thereby enhancing its maintainability and extensibility.

2. Full re-usability of the software infrastructure associated with the domain replication, thereby reducing the development costs for multiple physics packages. In addition, continual improvements to this infrastructure can occur orthogonal to the development of the physics packages and will benefit all the relevant packages when such improvements become available.

3. Due to similarities in the tracking of a particle through the mesh and the tracing of a ray across the mesh, the non-blocking inter-processor communication infrastructure developed for exchanging the photons across the domain boundaries can be re-used for exchanging the rays across the domain boundaries.

In addition to these advantages, this approach is encumbered with some significant disadvantages as well:

1. On parallel architectures based on clusters of shared-memory machines, for a fixed number of processors, the thread-based approach is more likely to achieve a better load balance and improved data locality than the domain replication approach. In fact, there is a high probability that the thread-based approach may set the upper bound to the achievable computational performance in a significant number of cases, barring problems due to memory availability constraints.

2. Domain replication results in redundant processing of all the physics packages involved in the simulation, and some packages derive no benefit from such redundancy. In fact, for certain packages, domain replication may result in a small reduction in performance due to increased inter-processor communication requirements.

3. Need for another layer of infrastructure to manage and execute the domain replication process, whose complexity is non-trivial, especially if dynamic domain replication is desired. Ideally, in the case of dynamic domain replication, the identity of the replicated domains and their multiplicity should be determined based on the current domain-wise distribution of the computational load associated with the one or more physics packages that require domain replication.

6 Software Development Options for a Common LRT Package

Given that the Ares and the Kull frameworks are based on two very different mesh types and the associated internal data representations, a LRT package implemented in a manner

that tightly couples it to one particular framework has a very low probability of being used efficiently within the other framework. If the laser ray tracing is to be performed on the native mesh of the framework, this likelihood is practically zero, unless the LRT package in question has been implemented with respect to the framework with the most general mesh type. Under such circumstances, the traditional avenue for sharing a physics package is limited to the use of an API with respect to the mesh and the field data. In this approach, the mesh and the relevant field data has to be copied into the data structures pre-defined by the client physics package, and possibly vice versa to complete the necessary two-way coupling. There are several drawbacks associated with this mode of operation:

1. The programming effort and the computational cost for converting the data structures may not be trivial.

2. The memory requirements might become excessive, especially if the host framework employs a light-weight mesh data structure with implicit connectivity.

3. Since the client physics package needs to be written to accommodate the most general mesh type, it is very likely to suffer undue performance penalties, when operating within the framework with the simpler mesh type. This is mostly due to increased costs for accessing the mesh and field data.

4. Loss of opportunities for exploiting the algorithmic efficiencies associated with the host framework's native mesh type and the internal data representation. This is especially true for the client package operations that are dependent on the topological and/or geometrical properties of the underlying mesh.

There is an alternative solution that alleviates some of these drawbacks. First, it uses the underlying algorithmic structure of the physics package under consideration to realize a generic (i.e., a framework-independent), implementation of the package, based on a common, abstract notion of the topological entities associated with the underlying mesh representation as well as the fields defined over that mesh. Then it relies on a set of host framework specific objects, that serves as a relatively thin adaptation layer between the generic physics package and the particular host framework, to provide a concrete representation of the abstract notions used within the generic implementation. While the class declarations associated with these objects define the function interfaces required by the generic implementation, the details of the specific functionality to be supported by these functions needs to be provided through appropriate documentation. It should be noted that the computational workload granularity associated with these functions can vary over a wide range, from being quite small to very substantial, with a corresponding variation in the algorithmic complexity. In effect, these objects provide a two-way communication interface between the client package and the host framework, enabling the client package to out-source from the host framework, a subset of the algorithmic steps, whose implementation is inherently tightly coupled to the internal data representation used by the host framework. As a consequence, that subset of operations can be implemented in a manner that is best able to exploit all the salient features of the host framework, thus preserving the computational efficiency vis-a-vis a direct, ad-hoc implementation within that framework. However, this flexibility comes at a price. Whenever the client package is to be adopted to operate within a code framework that has hitherto not been a host, an implementation of

the interface classes tailored to that specific code framework needs to be made available. In general, this is a non-trivial task, given that some of the out-sourced functionality can pack substantial algorithmic complexity.

Compared with the API-based approach, this approach essentially reverses the flow of information that occurs with the traditional approach. In the case of an API, the host framework has to learn to deal with the internal data representation used by the client package. In this approach, the host framework's internal data representation is exposed to the client package in an implicit and structured manner. However, the key to a successful realization of a physics package based on this approach is the correct identification of the minimalist set of abstract notions needed for the generic implementation and the set of functionalities that needs to be out-sourced from the client to the host.

As mentioned before, a concrete example of this approach is the Kull IMC package [1]. A specific example of a functionality that is outsourced to the host framework is the "distanceToBoundary" function used to track particles through the mesh. The algorithmic approach used for implementing this function and its computational cost is highly dependent on the host framework's mesh type. With this approach, each framework is free to implement this function in a manner that is best suited to its mesh type and the internal data representation used for that particular mesh type. In addition, the framework has the flexibility needed to use special features of the mesh type to improve the efficiency of the implementation.

The existence of the Kull IMC package provides us with several benefits. Since the IMC package has already been successfully adopted to operate within multiple code frameworks with very different characteristics, it provides us with an unambiguous validation of this approach. In addition, given the close similarity between the primary computational characteristics of the IMC and the LRT packages, the IMC package provides us with an overall design that has been refined over the years, after undergoing many successive iterations. The opportunity to re-use the IMC package's design would enable us to realize a significant saving in the time needed for the development of a common LRT package. In addition, given the similarities in the computational characteristics associated with tracking particles over a mesh and tracing rays through a mesh, some of the key components with substantial complexity, already developed for the IMC package can be re-used in the LRT package with little modification. This is especially true for the non-blocking inter-process communication infrastructure used for exchanging the particles among domains [5].

7 Conclusions

When the multitude of factors, both pro and con, cited in the previous sections are taken into consideration as a whole, it is our conclusion that in the current time frame, the design of the common LRT package should pay due attention to the issue of computational efficiency as well as other considerations such as interoperability, maintainability and extensibility. We are mindful of the tangible benefits offered by a LRT package with a potential for superior performance under certain circumstances. However, if that performance is to be achieved at the expense of all the other traits that we consider to be important as well, then we are compelled to seek an alternate design-point in the abstract parameter space spanned by these traits. We are equally mindful that such a design-point may force us to compromise a little on the computational efficiency. Currently, we do not have reliable means to quantify the

loss in efficiency associated with such an alternate design-point. It is likely to be dependent on the type of physical problem, the number of processors used as well as the capabilities of the software infrastructure tasked with the implementation of the domain replication strategy. However, as previously mentioned, there is at least one package that serves as an example, of a case where such an alternate design-point has already been adopted. Therefore, by adopting a similar design strategy for the LRT package, we can at least claim consistency with respect to the design of other physics packages with similar computational characteristics. Of course, as elaborated above, the benefits of such an approach is by no means limited to design consistency across physics packages. Therefore, we conclude that the design and development of the common LRT package should be based on the following considerations:

1. At least initially, the laser/ion beam physics capabilities and the end user interface should mirror what is currently available in the Hydra/Ares LRT package, to the extent it is appropriate. As a consequence, we expect to borrow extensively from the ideas embodied in the implementation of the ray propagation algorithm in that package. There may be additional opportunities to re-use certain sections of the code as well, possibly with minor modifications.

2. The overall design of the package will be based on the design concepts used within the Kull IMC package. Accordingly, we expect to be able to follow closely the design of the majority of the classes used within the IMC package. In the case of some components, we anticipate outright re-use of the existing IMC classes.

3. The performance optimization strategy for parallel machines will be based on the concept of load balancing through domain replication. It is assumed that the software infrastructure needed for achieving this is either already available or will become available in the near future, within both code frameworks. The functionality and the associated interfaces of this software should conform to what is currently needed to support the Kull IMC package. This however leaves the host frameworks with the freedom to choose how best to implement the necessary functionality.

A preliminary analysis of the structure of the LRT package suggests the following breakdown of the software components:

Mesh data base: Class design will be very similar to the current mesh data base used by the IMC package. However, additional functionality will be needed to support the quadratic ray paths within a mesh zone.

Material data base: Although the field data needed for laser ray tracing is somewhat different than what is needed by IMC, the class design will be very similar to what is already in use within the IMC package. Since laser ray tracing treats mixed material cells through homogenization, there may be opportunities for simplifying the implementation of this class. However, additional functionality may be needed to compute the limited zone-centered gradients needed for ray tracing.

Refractive Index data base: This class may be needed, if a user wishes to specify a non-standard refractive index profile, that could be either analytic or intensity-dependent. If so, it can be configured in a manner similar to the opacity data base class of the IMC package.

Boundary conditions: The classes related to the specification and application of boundary conditions share lot of similarities with those in use within the IMC package and present ample opportunities for code re-use. They may require some minor modifications to accommodate a variety of reflecting boundary condition types available within the Hydra/Ares LRT package. However, the implementation details associated with such modifications can be directly borrowed from the Hydra/Ares laser package.

Beam data base: Shares a lot of similarities with the classes related to the source data base of the IMC package, and hence can mimic its design. However, the actual beam type specifications will be borrowed from the Hydra/Ares laser package.

Domain exchange: Needed functionality is very similar to that of the IMC package. Hence, expect full re-usability of all the classes available in the IMC package, since they have a template parameter for the particle type.

Laser class: The primary laser class design should be based on the design of the IMC class. However, the code implementing ray tracing physics should be borrowed mostly from the Hydra/Ares laser package, sans the Posix thread related constructs. For example, a significant fraction of the code used for ray propagation can be re-used here.

Based on the three pre-conditions outlined earlier, we propose the following implementation plan for the common LRT package:

1. Implement a common LRT package after the design of the Kull IMC package.
2. Verifiy the implementation using stand-alone 1-D, 2-D and 3-D orthogonal Cartesian meshes. This should include the domain replication aspects as well.
3. Expose the design and the implementation for review by appropriate Ares and Kull representatives. If necessary, fine tune the design and implementation based on the input received during the review, followed by repetition of step(2).
4. Implement the Kull specific mesh and material data bases for 3-D geometries and verify all aspects of the implementation using laser-only problems.
5. Implement the Ares specific mesh and material data bases for 3-D geometries and verify all aspects of the implementation using laser-only problems. Use the existing Hydra package for additional comparisons.
6. Characterize the performance of the domain-replicated LRT package vis-a-vis the existing thread-based implementation.
7. Test the common LRT package in multi-physics problems of programmatic relevance using the Ares framework. Compare against the results obtained using the existing Hydra package.
8. Test the common LRT package in multi-physics problems of programmatic relevance using the Kull framework.
9. Repeat the steps (2)-(8), for 2-D RZ geometries.

8 Acknowledgements

The author is grateful to Patrick Brantley, Nick Gentile, Rich Hornung, Tom Kaiser, Gary Kerbel, David Miller and Alek Shestakov for many helpful discussions.

References

[1] N.A. Gentile, N. Keen, J. Rathkopf, The KULL IMC package, Tech. Rep. UCRL-JC-132743, Lawrence Livermore National Laboratory, Livermore, CA, 1998.

[2] Thomas B. Kaiser, Laser ray tracing and power deposition on an unstructured three-dimensional grid, Phys. Rev E, Vol. 61(1), pp. 895-905, 2000.

[3] Gary Kerbel (private communication), 2008.

[4] Alex Friedman, Three-Dimensional Laser Ray Tracing on an r-z Lagrangian Mesh, *Laser Program Annual Report 83*, Lawrence Livermore National Laboratory, Livermore, Calif., UCRL-50021-83, pp. 3-51 to 3-55, 1984.

[5] T.A. Brunner, T.J. Urbatsch, T.M. Evans, N.A. Gentile, Comparison of four parallel algorithms for domain decomposed implicit Monte Carlo, Journal of Computational Physics, vol. 212, pp. 527-539, 2006.

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.