Exploiting Domain Knowledge to Optimize Parallel Computational Mechanics Codes

Chenyang Liu[1], M. Hasan Jamal[1], Milind Kulkarni[1], Arun Prakash[2] and Vijay Pai[1]
[1]School of Electrical and Computer Engineering
[2]School of Civil Engineering
Purdue University
West Lafayette, IN, USA
{liu441, jamal0, milind, aprakas, vpai}@purdue.edu

ABSTRACT

An important emerging problem domain in computational science and engineering is the development of *multi-scale computational methods* for complex problems in mechanics that span multiple spatial and temporal scales. An attractive approach to solving these problems is *recursive decomposition*: the problem is broken up into a tree of loosely coupled sub-problems which can be solved independently and then coupled back together to obtain the desired solution. However, a particular problem can be solved in myriad ways by coupling the sub-problems together in different tree orders. As we argue in this paper, the space of possible orders is vast, the performance gap between an arbitrary order and the best order is potentially quite large, and the likelihood that a domain scientist can find the best order to solve a problem on a particular machine is vanishingly small.

In this paper, we present a system that uses domain-specific knowledge captured in computational libraries to optimize code written in a conventional language (C). The system generates efficient coupling orders to solve computational mechanics problems using recursive decomposition. Our system adopts the inspector-executor paradigm [9], where the problem is inspected and a novel heuristic finds an effective implementation based on domain properties evaluated by a cost model. The derived implementation is then executed by a parallel run-time system (Cilk) which achieves optimal parallel performance. We demonstrate that our cost model is highly correlated with actual application runtime, that our proposed technique outperforms non-decomposed and non-multiscale methods. The code generated by the heuristic also outperforms alternate scheduling strategies, as well as over 99% of randomly-generated recursive decompositions sampled from the space of possible solutions.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Compilers, Optimization; G.1.8 [**Partial Differential Equations**]: Finite element methods

ICS'13, June 10–14, 2013, Eugene, Oregon, USA.

General Terms

Algorithms; Performance

Keywords

Domain-specific Optimization; Multi-scale Method; Recursive Decomposition

1 Introduction

Multi-scale methods for computational mechanics have been an area of significant research interest in recent years [11, 12, 18]. These methods allow mechanical systems, both static and dynamic, to be simulated with vastly differing spatial and temporal scales in different parts of the domain. This allows areas of interest to be investigated at fine granularity but at high computational cost, while other portions of the problem can be approximated with a much coarser-grain simulation. Multi-scale methods thus hold the promise of simulating complex systems at the necessary level of resolution without incurring the cost of such fine-grained simulation throughout the problem domain. The multi-scale strategy is thus even more targetted towards achieving computational efficiency than well-studied strategies such as adaptive mesh refinement (AMR) that only adopt multiple spatial scales.

An attractive approach to multi-scale simulation that has been investigated in recent work is *recursive domain-decomposition* [3, 5, 13, 16, 20, 26]. A large problem is broken up into a set of loosely-coupled subdomains; these subdomains can be solved independently, except for points on potential shared *interfaces* that connect them. These shared interfaces require that the solutions of each individual subdomain be coupled to ensure that their solutions at the interface are consistent. As long as the interfaces are small relative to the subdomain sizes, it is computationally advantageous to decompose a large system versus solving the system as a single entity. For the particular case of recursive bisection adopted in this study, coupling is a pair-wise operation, and as subdomains are coupled, a solution is obtained for the combined, larger domain. Hence, by hierarchically coupling the subdomains, the solution for the overall system can be obtained. We describe this coupling order using a *coupling tree*. Figure 1 shows a sample decomposition, where the problem domain is decomposed into six subdomains that are solved independently and coupled together according to the coupling tree depicted in the figure.

A critical point about this hierarchical approach to solving multi-scale problems is that the *structure* and *topology*

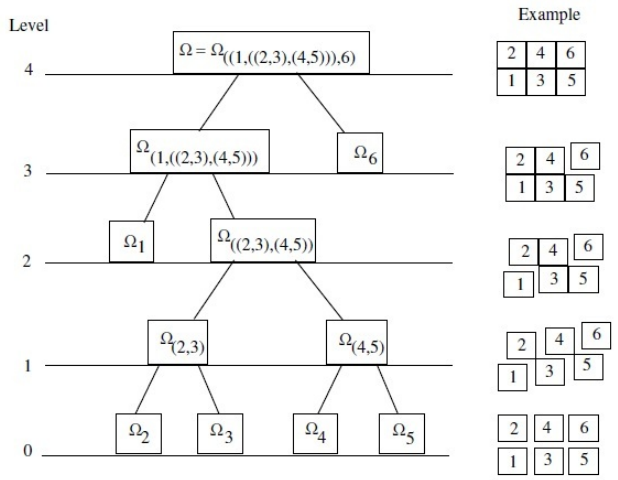


Figure 1: A Decomposed Problem Domain and Corresponding Recursive Coupling Order

of the coupling tree has a significant effect on the performance of the algorithm. Because the coupling operation is both commutative and associative, a vast number of unique coupling trees are possible (945 for a problem with just 6 subdomains), making it highly unlikely for all but the simplest problems that a domain scientist will adopt an effective coupling order. Furthermore, because the various relevant parameters associated with computational costs are problem-dependent, finding the optimal coupling tree becomes even more difficult, as no single approach for finding an optimal tree coupling order may work for all different problems.

To tackle this problem, we have developed a run-time optimization system based on the inspector-executor approach. We first inspect the original program to infer the high level computational structure of the problem. We then use semantic properties of the coupling operation, as well as a domain-specific cost model to obtain cost values that reflect the coupling of multi-scale subdomains. Next, we generate an optimal coupling order based on those cost values. Finally, the optimized schedule is run using the Cilk framework to exploit available parallelism. We find that our approach yields sequential and parallel performance that is significantly better than other available techniques.

Finally, we discuss how our overall approach to optimizing domain applications might be generalized, both in terms of other computational algorithms that can benefit from our system in particular, and more broadly how similar semantics-exploiting approaches might be applicable in other domains.

2 Background

In this section we provide a general description of the physical problems encountered in mechanics that we will be solving using parallel recursive domain decomposition. The dynamical behavior of almost all mechanical systems is generally governed by laws of physics expressed as partial differential equations (PDEs) of continuum mechanics. These PDEs are most commonly solved by expressing them in integral forms and using numerical methods, such as finite elements, finite volumes, finite differences etc. to evaluate these integrals. For instance, the PDE:

$$-\rho\ddot{\boldsymbol{u}} + \text{div}\boldsymbol{\sigma} + \boldsymbol{b} = \boldsymbol{0} \qquad \forall \boldsymbol{x} \in \Omega \tag{1}$$

describes the displacement $\boldsymbol{u}$ and stresses $\boldsymbol{\sigma}$ at *all* points $\boldsymbol{x}$ in a body Ω, being acted upon by forces $\boldsymbol{b}$. The density is denoted by ρ and a superimposed dot ($\dot{}$) represents a derivative with respect to time ($\frac{d}{dt}$). The "div" operator represents vector divergence, or the sum of partial derivatives in each spatial dimension ($\text{div}\boldsymbol{\sigma} = \frac{\partial\sigma}{\partial x} + \frac{\partial\sigma}{\partial y} + \frac{\partial\sigma}{\partial z}$) . Note that Equation (1) is simply an expression of Newton's 2nd law for every point $\boldsymbol{x}$ in Ω. Upon writing this time-dependent PDE in integral form and numerically *discretizing*, one obtains a *system* of ordinary differential equations (ODEs):

$$\boldsymbol{M}\,\ddot{\boldsymbol{u}}(t) + \boldsymbol{D}\,\dot{\boldsymbol{u}}(t) + \boldsymbol{K}\,\boldsymbol{u}(t) = \boldsymbol{p}(t) \tag{2}$$

where $\boldsymbol{M}$, $\boldsymbol{D}$ and $\boldsymbol{K}$ represent the mass, damping and stiffness matrices respectively, associated with the discretization, $\boldsymbol{p}(t)$ is a vector of time-dependent forces acting on the body, and $\boldsymbol{u}(t)$ is the vector of displacements that needs to be solved for. Most commonly, these ODEs are solved by finite-difference schemes by approximating the time derivatives using difference formulas. This allows one to solve for the state $\boldsymbol{u}_{n+1} \approx \boldsymbol{u}(t_{n+1})$ of the system at some time t_{n+1} from a known state $\boldsymbol{u}_n \approx \boldsymbol{u}(t_n)$ by advancing through a small time-step Δt where $t_{n+1} = t_n + \Delta t$.

$$\mathbb{M}\,\mathbb{U}_{n+1} = \mathbb{P}_{n+1} - \mathbb{N}\,\mathbb{U}_n \tag{3}$$

This process can be repeated successively to advance the solution in a *time-stepping* manner [19]. However, depending on the problem, the size of the this system of equations can be very large making its solution as a single complete system very computationally intensive. One way to avoid solving this system as a whole is by using domain decomposition. This approach divides the problem into smaller subdomains, solves them independently (possibly in parallel), and couples them back by enforcing continuity constraints on the interfaces between the sub-domains.

2.1 Solving a Recursively Decomposed Domain

In this section we illustrate a non-iterative hierarchical implementation of the *recursive* domain decomposition method. Given a finite element mesh partitioned into S subdomains, a hierarchy of subdomains can be built by combining two subdomains at a time until the original undecomposed mesh is recreated. Such a hierarchy can be effectively represented by the *tree* structure as shown in Figure 1. The *leaf* nodes in the *tree* represent the subdomains that are not further subdivided. The original undecomposed structure Ω is called the *root* node. In general, for any node $\Omega_{(A,B)}$ in the tree, the problem is substituted with two coupled sub-problems for Ω_A and Ω_B replacing Equation (3) with:

$$\left[\begin{array}{cc|c} \mathbb{M}^A & & \mathbb{C}^A \\ & \mathbb{M}^B & \mathbb{C}^B \\ \hline \mathbb{B}^A & \mathbb{B}^B & \boldsymbol{0} \end{array}\right] \left[\begin{array}{c} \mathbb{U}^A_{n+1} \\ \mathbb{U}^B_{n+1} \\ \hline \boldsymbol{\Lambda}^{(A,B)}_{n+1} \end{array}\right] = \left[\begin{array}{c} \mathbb{P}^A_{n+1} - \mathbb{N}^A\mathbb{U}^A_n \\ \mathbb{P}^B_{n+1} - \mathbb{N}^B\mathbb{U}^B_n \\ \hline \boldsymbol{0} \end{array}\right] \tag{4}$$

Note that the matrices $\mathbb{C}_A, \mathbb{B}_A$ and $\mathbb{C}_B, \mathbb{B}_B$ represent connectivity matrices associated with the subdomains Ω_A and Ω_B respectively for the interface $\Gamma^{(A,B)}$ between them exclusively. The matrix system (4) can be solved in a decomposed

manner (instead of as solving it as a whole), using the following sequence of operations at each time step:
(i) Solve the smaller *uncoupled* problems

$$\mathbb{M}^s\,\mathbb{V}^s_{n+1} = \mathbb{P}^s_{n+1} - \mathbb{N}^s\mathbb{U}^s_n \tag{5}$$

for both subdomains $s = \{A, B\}$.
(ii) Solve for the *interface* Lagrange multipliers $\boldsymbol{\Lambda}^{(A,B)}_{n+1}$:

$$\boldsymbol{H}^{(A,B)}\,\boldsymbol{\Lambda}^{(A,B)}_{n+1} = \boldsymbol{f}^{(A,B)}_{n+1} \tag{6}$$

where the matrix $\boldsymbol{H}^{(A,B)}$, that represents the interface operator required to enforce continuity of the solutions $\mathbb{U}^A_{n+1}$ and $\mathbb{U}^B_{n+1}$ across the interface $\Gamma^{(A,B)}$, is computed only once as discussed in the remark below. The vector $\boldsymbol{f}^{(A,B)}_{n+1}$ is obtained from the uncoupled solution in step (i) above as $\boldsymbol{f}^{(A,B)}_{n+1} = \mathbb{C}^A\mathbb{V}^A_{n+1} + \mathbb{C}^B\mathbb{V}^B_{n+1}$.
(iii) *Update* the individual subdomain solutions:

$$\mathbb{U}^s_{n+1} = \mathbb{V}^s_{n+1} - \mathbb{Y}^s\boldsymbol{\Lambda}^{(A,B)}_{n+1} \tag{7}$$

again for both subdomains $s = \{A, B\}$, and where the matrices $\mathbb{Y}^s$ are also precomputed once as discussed in the remark below.
Remark: The $\mathbb{Y}^s$ matrices are precomputed once for both subdomains as:

$$\mathbb{M}^s\,\mathbb{Y}^s = \mathbb{C}^s \tag{8}$$

and do not change during the rest of the computation (for linear problems), so they are saved and used for every time-step. The interface operator $\boldsymbol{H}^{(A,B)}$ is then simply obtained by matrix multiplication:

$$\boldsymbol{H}^{(A,B)} = \mathbb{B}^A\,\mathbb{Y}^A + \mathbb{B}^B\,\mathbb{Y}^B \tag{9}$$

and it also does not change for the duration of the computation.

A representative pseudocode for solving a general node $\Omega_{(A,B)}$ in the tree is given by the *recursive* subroutine TreeSolve presented in Figure 2(a), where the inputs are $\Omega_{(A,B)}$ (the node to be solved), $\mathbb{P}^{(A,B)}$ (applied external forces to the mechanical system) and $\mathbb{U}^{(A,B)}_n$ (the current state of the subdomain at time-step t_n), and the output is $\mathbb{U}^{(A,B)}_{n+1}$ (the desired state of the subdomain at the next time-step t_{n+1}). Note that each call to TreeSolve requires computation involving the $\mathbb{Y}$ matrices, which are computed with the recursive BuildY subroutine presented in Figure 2(b). This function uses the various degrees of freedom (dof) on the interface $\Gamma^{(A,B)}$ to calculate the $\mathbb{Y}$ matrices. Input to the subroutine is a tree node $\Omega_{(A,B)}$ and the outputs are the $\mathbb{Y}$ matrices for coupling its two daughter nodes Ω_A and Ω_B.

The coupling tree is solved with a single call to the recursive function: TreeSolve(Ω, $\mathbb{P}$, $\mathbb{U}_n$, $\mathbb{U}_{n+1}$) for a given load $\mathbb{P}$ and initial conditions $\mathbb{U}_n$. Leaf nodes are solved for their responses to external forces using a conventional time integration scheme using the Newmark method [19] and these responses are later coupled in the hierarchy to obtain the full system solution. As for the coupling operations, they only require one multiplication on a smaller (interface-sized) matrix $\boldsymbol{H}^{(A,B)}$ and vector $\boldsymbol{\Lambda}^{(A,B)}_{n+1}$. The subroutine BuildY makes calls to TreeSolve which requires the $\mathbb{Y}$ matrices to have already been computed apriori. With our ordering of the recursive calls in the subroutines, we ensure that all $\mathbb{Y}$ matrices below the particular coupling node have already been computed before we attempt to couple its children.

```
TreeSolve(Ω_(A,B), P^(A,B), U_n^(A,B), U_{n+1}^(A,B))
    If Ω_(A,B) is a Leaf Node then
        Solve Ω_(A,B)
    Else
        Call TreeSolve(Ω_A, P^A, U_n^A, V_{n+1}^A)
        Call TreeSolve(Ω_B, P^B, U_n^B, V_{n+1}^B)
        Compute Λ_{n+1}^(A,B) from H^(A,B) Λ_{n+1}^(A,B) = f^(A,B)
        Update U_{n+1}^A = V_{n+1}^A − Y_A^(A,B) Λ_{n+1}^(A,B)
        Update U_{n+1}^B = V_{n+1}^B − Y_B^(A,B) Λ_{n+1}^(A,B)
    End if
END TreeSolve
```

(a) Subroutine to solve a general node $\Omega_{(A,B)}$ in the tree.

```
BuildY(Ω_(A,B), Y_A^(A,B), Y_B^(A,B))
  Do for K = A, B
    If Ω_K is NOT a Leaf Node :
      Call BuildY(Ω_K, Y^K_LeftChild(K), Y^K_RightChild(K))
    Do for each dof J in Γ_b^(A,B)
      If K = A : P^K = +1 load on dof J in Ω_K
      If K = B : P^K = −1 load on dof J in Ω_K
      Call TreeSolve(Ω_K, P^K, 0, column J of Y_K^(A,B))
    End do
  End Do
END BuildY
```

(b) Subroutine to build the $\mathbb{Y}$ matrices for a general node.

Figure 2: Subroutines for Recursive Solution Procedure

The final algorithm for solving a linear dynamics problem can be summarized in the following steps:

1. Construct the hierarchy of subdomains and build their interface matrices.
2. Form the individual subdomain system matrices.
3. Call BuildY(Ω, $\mathbb{Y}_{left}$, $\mathbb{Y}_{right}$).
4. Form and factorize $\boldsymbol{H}$ matrices for all coupling levels.
5. Compute initial acceleration on the global mesh.
6. Loop over number of time steps FOR $n = 1{:}N$
   - Call TreeSolve(Ω, $\mathbb{P}$, $\mathbb{U}_n$, $\mathbb{U}_{n+1}$).

It is important to note here that the initial acceleration calculations and BuildY routines are required to be performed only once at the beginning of the computation. Especially in the case of the BuildY routine, these *initialization* functions are quite costly in terms of computational cost compared to the final TreeSolve routine. However, long term simulation of physical models may require running thousands or millions of timesteps, which amortizes all other costs aside from TreeSolve. Later, when discussing performance of our TreeSolve algorithms, we omit the time taken by other routines.

2.2 Multiple Time Scales

In section 2.1 we discussed a method that utilizes the same time-step for discretizing all the different subdomains in a tree. However, this approach is most beneficial when different subdomains are solved at different timescales. The inclusion of multiple timescales allows for a more fine-grained

simulation of specific subdomains within the global problem which contain physically interesting features. The use of different timescales for different subdomains necessitates some key modifications to the TreeSolve algorithm. The main difference is that subdomains with smaller timescales are solved multiple times in a single TreeSolve to advance by a large timestep. We will briefly discuss how two subdomains with different timestep granularities are coupled together. Finer details of the multi-time-scale method can be obtained from the references [22–24].

We describe the method to couple two subdomains, subdomain A at timestep ΔT and subdomain B at timestep Δt where $\Delta T = m\Delta t$, m being an integer timestep ratio. This method can be extended for more layers of coupling of subdomains at different timestep ratios. Using the Newmark method, we wish to advance both subdomains A and B from time t_0 to $t_0 + \Delta T$, shown in Figure 3. However, subdomain A will only require advancing by one large timestep ΔT, whereas subdomain B needs to advance m times by the small timestep Δt. In the context of our TreeSolve algorithm, this means that subdomain B will perform the leaf solve m times while subdomain A performs it a single time. We handle the different timestep ratios by keeping track of the largest ΔT at each coupling node, and calling TreeSolve m times depending on the timestep ratio between its left and right subtrees.

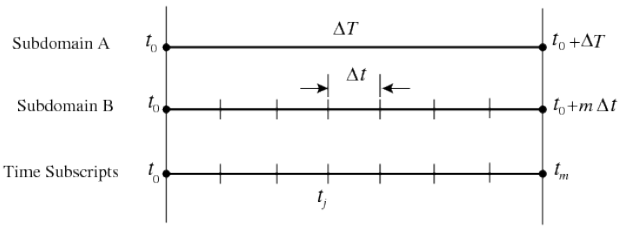


Figure 3: Representation of Time Steps for two Subdomain Case

The multi-time scale coupling requires one other modification in the algorithm which we call *load transfers*. These load transfers are required when there is a common interface between subdomains with different timesteps. Because coupling only occurs at the larger timesteps, linear interpolation is used to approximate the interface reactions at the smaller timesteps. When advancing at a smaller Δt timestep, these load transfers are introduced to preserve equilibrium with the subdomain at the larger time step ΔT. The following equation defines these load transfers $\boldsymbol{S}_j$:

$$\boldsymbol{S}_j = \left(1 - \frac{j}{m}\right) \boldsymbol{\Lambda}_0 \qquad \forall j \in [1, 2, \cdots, m] \qquad (10)$$

where m is the time step ratio ($\Delta T/\Delta t$) for the coupling, j is the intermediate timestep of subdomain B, and $\boldsymbol{\Lambda}_0$ is the value of $\boldsymbol{\Lambda}^{(A,B)}$ obtained from the coupling operation in Equation (6) for the previous timestep. Since the $\boldsymbol{\Lambda}$ vector is computed in our initial acceleration calculation at time zero and at each coupling operation, we are guaranteed $\boldsymbol{S}_j$ will be available at every timestep for every subdomain solve, to allow simultaneous solution of different subdomains in parallel.

The final solution algorithm is presented in Figure 4, where subtrees are solved multiple times depending on their timestep ratio at an interior node, and load transfers are saved at each

```
RecursiveSolve(Node)
    If Leaf(Node)
        LeafSolve(Node) [with load transfers]
    Else
        For(Timestep ratio on Left)
            RecursiveSolve(Left)
        For(Timestep ratio on Right)
            RecursiveSolve(Right)
        Couple(Left, Right)
        Save Load Transfer Matrix Λ
    End If
```

Figure 4: Pseudocode for Multi-Timestep TreeSolve

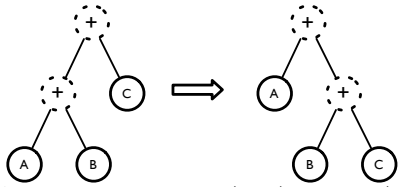
subdomain and handled correctly at every leaf solve. Due to the additional complexity of handling multiple timescales, additional constraints are placed to make our solver feasible. In our problems, we ensure that timestep values are all a fixed ratio of a base timestep value. We also constrain our tree so that child nodes are always at a lower or equivalent time step of its parent node. Finally, in order to correctly handle load transfers required by the multi-timescale method, we require that all subdomains at each timescale be coupled as separately, and each tree then coupled together with the tree of the next smallest timescale, and so forth in the final coupling tree. Although, this restricts the space of trees in the solution space, the entire space still grows exponentially with the number of subdomains.

3 Semantics of Coupling Trees

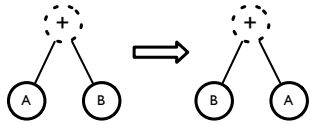
In this section we describe the properties of our coupling trees, which lays the foundation for creating valid tree orderings. Our inspector phase takes these contraints to find a suitable objective function which is optimized to find the best coupling order. We show that the number of distinct valid trees is exponentially large and that it is not an easy task to find an optimal ordering. To obtain a good solution, we introduce a cost model which correlates with the computational time taken to complete the TreeSolve routine, which becomes the target of our optimization.

The important properties of a coupling operation are that they are both associative and commutative. Coupling two subdomains A and B is identical regardless of order, and coupling (A B) with C is no different than coupling A with (B C). Thus, given a particular coupling tree, we can create a new, equivalent tree by re-associating the coupling operations. This is equivalent to performing tree-rotations on any pair of coupling operations, as shown in Figure 5(a) or by performing commutative swaps of the tree, as shown in Figure 5(b).

By performing a series of these commutative and associative manipulations, we see that *any* binary tree with a particular set of leaf nodes represents a valid coupling order for a given problem. However, trees with only their left and right children swapped will have the same behavior in terms of coupling, and will hence have the same performance. The question we ask is "How many possible distinct trees $\mathcal{T}(s)$ are there for a given number of subdomains s?" It can be shown that the number of possible trees for s subdomains, $\mathcal{T}(s)$, is given by:



(a) Associative rotation: $(a \cdot b) \cdot c \equiv a \cdot (b \cdot c)$



(b) Commutative swap: $a \cdot b \equiv b \cdot a$

Figure 5: Possible Optimizing Operations on Trees

$$\mathcal{T}(s) \quad = \quad \frac{(2s-2)!}{2^{s-1}(s-1)!}$$

Intuitively, this is related to the Catalan numbers, which represent (among other things) the total possible configurations for a binary tree. For a tree with s leaves, we start with the $s-1^{th}$ Catalan number, $\frac{(2s-2)!}{s!(s-1)!}$. This must then be multiplied by $s!$ to account for all possible commutative reorderings, and then divided by $2^{(s-1)}$ to disregard left-right reflections that have no impact on performance. This quantity grows very quickly: $\mathcal{T}(8) = 135{,}135$, and for any reasonable number of subdomains (even small problems may have 32 subdomains), the space of possible trees is too large to exhaustively search for an optimal solution. In the multi-time-scale case, the constraint that trees of different timescales must be coupled at a single point somewhat reduces the possible number of orderings, as not all trees adhere to the coupling constraints. Nevertheless, the above expression reflects the number of orderings present in each of the subtrees.

Given the large number of trees, how might we select the correct tree for a given problem? In this paper, we adopt an approach that is based on the inspector-executor model for program optimization. In this approach, an inspector evaluates the given tree coupling order by a user, and uses heuristics to find an optimal tree coupling order that minimizes computational cost, thus reducing execution time and gaining performance. In order to set a performance-based objective, we modeled the computational cost after the sequence of computations being carried out by the algorithm.

4 Cost Model & Heuristics

To automatically find optimal coupling schedules for our decomposed problems, we adopt a model to quantify the cost of any particular coupling configuration. Using this model, we develop an optimization scheme that selects from the configuration space of all coupling schedules one that produces a low cost. This section discusses the cost model and the optimization system we developed based on these models.

4.1 Cost Model

A cost model was developed to accurately reflect the execution time of the TreeSolve code prior to solving the coupling tree. This is done by inspecting the coupling tree and modeling its costs based on the properties of the input, namely the interface and subdomain sizes. Once the tree structure is determined, we then measure the matrix sizes for all operations to estimate the time it will take to run during actual execution. There are several different cost components that contribute to the TreeSolve cost. Most of them are matrix-matrix or matrix-vector multiplications as well as a few matrix-matrix and matrix-vector solves. We characterize their costs based on the sizes of the matrices and the amount of work necessary to compute their solutions. The following variable definitions are used to come up with the costs:

Cost per leaf	Cost per interior node	
n_i^3	$\sum_{i \in C} n_i^2 m_{i:v}$	Update of Y
	$\sum_{j \in C} n_i m_{i:v}^2$	Calculating H
	m_v^3	Calculating λ
	$\sum_{i \in C} n_i m_{i:v}$	Updating X
	$\sum_{v' \in C} \sum_{i \in v'} n_i m_{i:v'} m_{i:v}$	Solving F
	$\sum_{v' \in C} m_{v'}^2 m_v$	Calculating Λ
	$\sum_{v' \in C} \sum_{i \in v'} n_i m_{i:v'} m_{i:v}$	Updating Y

Table 1: Cost Model Contributions from Coupling Tree

1. $\boldsymbol{n}_i$: number of equations in subdomain $\boldsymbol{S}_i$
2. $\boldsymbol{m}_v$: number of equations in the coupling interface for interior node v
3. $\boldsymbol{m}_{i:v}$: number of equations in interface of v that are associated with subdomain $\boldsymbol{S}_i$
4. $\boldsymbol{C}$: Descendant leaf nodes of an interior node
5. $\boldsymbol{v}, \boldsymbol{v}'$: interior nodes

For a given tree (or subtree), the total cost of that tree comes from the leaf nodes in that (sub)tree, which represent the decomposed subdomains, and the interior nodes of the (sub)tree, which represent the coupling cost of the tree. The values of these costs are given in Table 1.

The different matrices $\boldsymbol{Y}, \boldsymbol{H}, \boldsymbol{\lambda}, \boldsymbol{X}, \boldsymbol{F}$ and $\boldsymbol{\Lambda}$ (some temporary) are used in the final TreeSolve and BuildY subroutines. Details of the equations and computations for these matrices can be found in the reference [22]. We note one important fact: each interior node's cost is determined by the number of leaf nodes under that interior node, and hence interior nodes higher in the tree (*i.e.*, coupling operations higher in the tree) have higher costs than those lower in the tree.

With this cost model, our inspector can access cost values of different trees without having to solve them. In our results, we show that the cost values correlate well with actual runtimes, enabling us to use heuristics based on the models to create effective coupling trees.

Figure 6 shows CDF of tree cost for 1000 randomly generated trees, for a given computational mechanics problem decomposed into 8 subdomains. It is observed that by picking only 1000 randomly coupled trees out of the whole tree space ($< 1\%$ of the space), the costs vary substantially, and between two coupling configurations, the costs can differ by an order of magnitude. Our work presents a way to find a configuration that obtains a near-optimal runtime among the space of coupling trees.

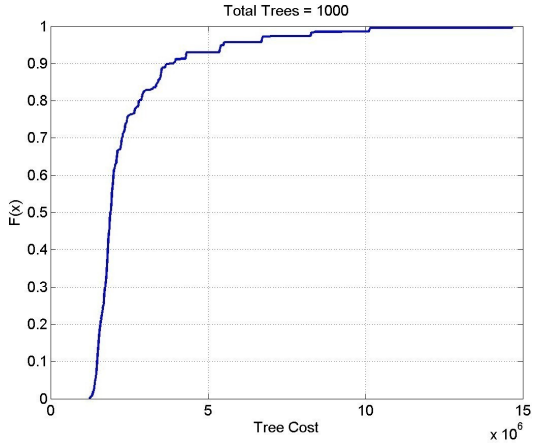


Figure 6: CDF of 1000 Random Trees based on tree costs

4.2 Tree Building Heuristics

With a large space of existing possible trees for a given problem, the probability of a domain scientist choosing a low cost tree is small. A domain scientist with limited computing knowledge might couple subdomains based simply on the partition ordering of the subdomains (*i.e.*, the order in which partitioners such as METIS [14] produce mesh partitions). We call this the Default approach, and it represents the baseline coupling order for a problem. As described above, our system adopts the inspector-executor paradigm that analyzes the input of a given hierarchical multi-scale computational mechanics problem and automatically determines effective coupling orders based on our cost model before solving the coupling tree.

Given that the cost model developed in the previous section computes costs on a per-subtree basis, an intuitive approach would be to develop a greedy algorithm that builds the coupling tree from the bottom up. Recall that when two subdomains are coupled after being solved for a time step, their coupled solution is similar to first joining together the equations and then solving the larger, merged subdomain. As a result, a greedy approach might proceed by coupling together subdomains based on some heuristic such as coupling subdomains with the largest shared interface, and then treating the coupled subtree as a new, larger subdomain and repeating the process. At each step, the cost model can be used to evaluate the coupling heuristic for the current set of subdomains to be coupled.

We experimented with several greedy heuristics; however, there is a fundamental drawback with this greedy approach. Although it optimizes the cost for each instance of coupling, it ignores the future cost of coupling subtrees necessary to solve the decomposed system, causing poor performance. Recall that coupling operations higher in the tree inherently cost more than coupling operations lower in the tree. Hence, bottom-up algorithms, which cannot even predict what the costs higher in the tree will be until the lower levels have been assembled, are not appropriate.

Instead, we adopt a *top down* approach, focusing on minimizing the coupling costs at higher levels of the tree. Beyond this, we would also like to maintain the balance of the tree, with both left and right subtrees for any node having roughly equal solution times; this will improve the potential parallelism of the resulting tree.

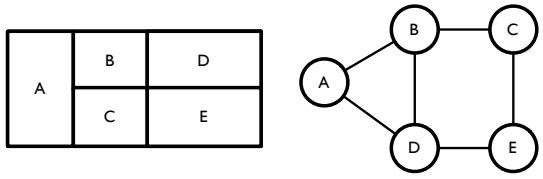


Figure 7: Graph Representing a 5 Subdomain System

To implement our top-down approach, the inspector phase analyzes the input problem and views the partitioned mesh as an undirected graph, as shown in Figure 7. In the graph, each node represents a subdomain, and edges represent shared interfaces between them. In other words, the graph represents the topology of the subdomains. One approach to building a tree top-down given this abstraction is to simply perform a series of graph bisections. The graph is divided into two partitions, with nodes from each partition representing the subdomains that will be in the left and right subtrees of the root. This process can be repeated recursively to construct the entire subtree. In the multi-time-scale case, we produce multiple graphs, one for each timestep in the problem (as subdomains at each time step must be coupled together before moving on to larger time steps).

Note that even performing this top-down bisection relies on domain knowledge: because this tree-building approach can produce arbitrary results, it is only legal *because we know the domain semantics allow for any coupling order.* Nevertheless, despite leveraging domain semantics to build the tree, this approach does not consider that leaf solve and coupling costs are based on properties of the operations such as the number of equations and interface sizes. This bisection technique is what an application programmer might think as optimal for decomposing meshes in a top-down fashion. However, this method (labeled cost-agnostic in our experiments) does not take into account *domain specific cost information.*

One of the key contributions of this work, labeled our domain-specific heuristic, is a scheduling procedure that not only integrates domain-specific semantic information, but also domain-specific cost information. We use domain knowledge obtained from our cost model and apply the coupling and subdomain costs to the edge and node weights in our graph prior to partitioning. In particular, a node's weight is calculated based on the leaf-node cost model, and is the cube of the number of equations in the subdomain. Precise edge weights are harder to determine, as they rely on the overall structure of the subtree rooted at a coupling node. However, the primary cost of a coupling operation is proportional to the size of the interface between the domains being coupled. Hence, edge weights are set to the interface size of the two subdomains connected by that edge.

We then use METIS to perform repeated bisections of this graph, as in the cost-agnostic approach. There are two key differences, however. METIS attempts to create balanced partitions while minimizing the weight of cut edges. The cost of a coupling operation is proportional to the interface size between the two domains being coupled. When the graph is bisected, the two sets of nodes representing the two domains that will be coupled, and the interface be-

tween those domains is exactly captured by the edges that are cut. Hence, because edge weights are determined by interface size, by minimizing the weight of cut edges, METIS naturally produces low-cost coupling operations. Second, the dominant cost in the TreeSolve algorithm is the cost of solving leaf nodes for a tree. Because these costs are exactly captured by the weights on nodes, by attempting to balance the two partitions of a graph, METIS naturally produces balanced trees. Hence, by incorporating knowledge of our domain-specific cost models into the top-down tree-building framework, we can produce low-cost, well-balanced trees. This tree building strategy focuses on minimizing total work, but still performs well for both parallel and serial execution. Intuitively, the domain-specific heuristic produces well balanced trees. Provided with sufficient parallelism, minimizing the total work also reduces the total parallel runtime, resulting in optimal parallel performance as well.

After producing a coupling tree, the inspector transforms this tree into an execution schedule, which is passed on to the executor phase for execution. Our approach provides two executors: a sequential executor and a parallel executor. Both executors use platform-optimized BLAS [6] and LAPACK [2] routines to efficiently compute the matrix solutions for a system using the hierarchical method. For our parallel implementation, we use Cilk [7] to obtain optimal parallel performance. Section 5 describes the parallel execution strategy.

5 Parallelization

This section describes the details of the TreeSolve parallel executor. We briefly discuss the semantics of Cilk and its role in our parallel execution scheme. We then provide details on how TreeSolve is parallelized, highlighting specific issues that arise when running in parallel and how we overcome them.

Cilk is an algorithmic multithreaded language used to optimize parallel programs. We chose to use Cilk because it is a minimally intrusive parallelization tool which can deal with irregular codes without introducing large synchronization overheads. The parallel TreeSolve calls the Cilk multithreaded runtime system using fork-join parallelization at each tree node. This allows each subtree path to run concurrently, while Cilk's work-stealing mechanism effectively handles any load balance issues that arises due to complexities of the the recursive algorithm and multiple timescales.

Running the TreeSolve along with Cilk requires adding some minimally intrusive code to our existing algorithm. Two Cilk constructs, spawn and sync, are used to direct the parallel execution. The following pseudocode shows the modified TreeSolve algorithm using Cilk.

Figure 8 captures the overall structure required by Cilk to run the TreeSolve algorithm in parallel. Running the Cilk version of TreeSolve will spawn new threads to concurrently execute the left and right paths at each interior node of the tree. Threads eventually reach a leaf node, where it then runs the routine for a subdomain solve to advance a timescale. Once threads finish from both the left and right paths, they arrive at the sync point, where the atomic coupling operation updates the subdomains from the left and right subtrees. Additional bookkeeping is required to correctly couple subdomains of different timescales. Static timescale values are stored at each node in our tree structure to validate coupling of subtrees. Since subtrees are decoupled from other subtrees from other paths, we can execute them in parallel; however, each path may be executed different number of times depending on their assigned timescales.

```
Cilk TreeSolve(Node)
    If Leaf(Node)
        LeafSolve(Node) [with load transfers]
    Else
        Spawn TreeSolve(Left)
        Spawn TreeSolve(Right)
        Sync
        Couple(Left, Right)
        Save Load Transfer Matrix Λ
    End If
```

Figure 8: TreeSolve Algorithm using Cilk

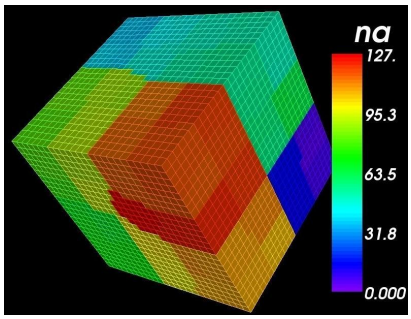
5.1 Difficulties

Optimal parallelization of TreeSolve is not an easy task. Due to the recursive nature of the TreeSolve and multiple timescales, algorithms using a typical fork-join model performs poorly due to load imbalance. Previous versions of TreeSolve which used these methods saw unpredictable parallel performance. Depending on the input and its coupling order, parallel execution was extremely inefficient, sometimes taking an additional order of magnitude for extreme cases. We also implemented a dynamic parallel algorithm which did not use a fork-join model. This dynamic execution scheme maintained a task queue composed of coupling and subdomain operations, and executed them as it became available, then enqueuing new tasks once dependencies were satisfied. This scheme resolved issues caused by load imbalance, but required locks to synchronize at every coupling operation of execution. Our Cilk implementation is very similar to that approach; however, Cilk uses synchronization methods which incur a lower overhead than traditional locks, while using a work-stealing method to remedy load imbalance. For most inputs, Cilk is able to outperform other parallel implementations of the TreeSolve algorithm.

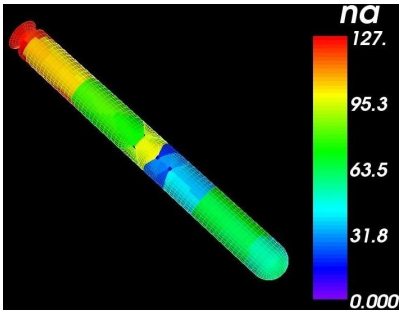
6 Evaluation

In this section, we validate our cost model against actual runtime performance and evaluate the performance of the cost-agnostic and domain-specific heuristics running both sequential and parallel TreeSolve algorithms. We focus on the results of three physical systems, represented as finite element meshes of hexahedral elements, as testing systems: 1) a 128 subdomain cube mesh with 15625 elements 2) a 128 subdomain rocket mesh with 3632 elements and 3) a 128 subdomain stargrain mesh with 37152 elements. Each mesh is partitioned by METIS to give an optimal partitioning. These systems are simulated using the multi-scale solver running on an AMD Opteron 6176 SE system configured with four 12-core processors (a total of 48 cores) running at 2.3 GHz. Each input contains two timescales, with an eighth of the subdomains assigned to run at a smaller timestep and the rest of the system at a larger timestep. The timestep ratio is 10 for the cube and stargrain input, while it is 100 for the rocket input. The timestep values are arbitrarily chosen; however, for the rocket input, the smaller timesteps are assigned to the subdomains of interest, containing cracked

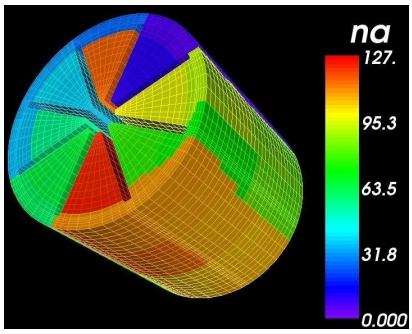
regions of the physical system. Figure 9 shows the three meshes after partitioning, as seen from the various colors.



(a) Example of a Partitioned Cube Mesh



(b) Example of a Partitioned Rocket Mesh



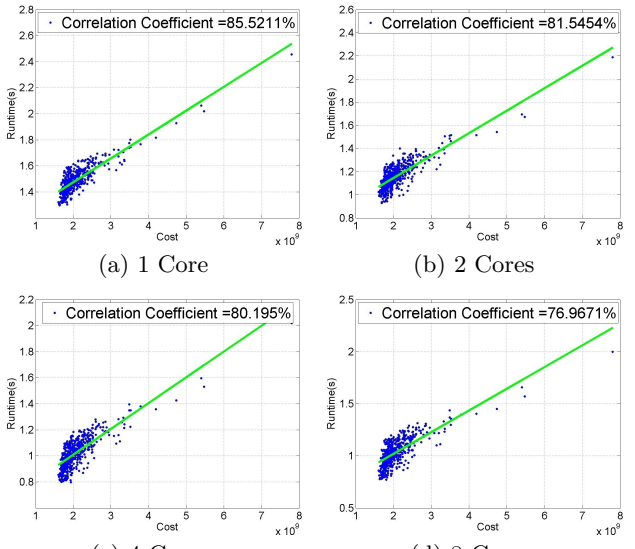
(c) Example of a Partitioned Stargrain Mesh

Figure 9: Visualization of Finite Element Mesh Models

6.1 Cost model validation

To validate our cost model, a set of 500 randomly generated coupling trees was used to represent a sample of the entire space of coupling trees. These trees are created by starting with all subdomains in separate subtrees and randomly selecting two subtrees to be coupled together until the full tree is obtained. Note that this random coupling means that occasionally two subdomains that do not share an interface will be coupled. This schedule is mathematically and semantically correct: the final solution will be computed correctly. Nevertheless, such coupling orders are nonsensical from a performance perspective, as there is no benefit in reduced work to be gained from coupling subdomains that are not adjoining.

The rocket input shown in Figure 9(b), partitioned into a 32 subdomain mesh (as a sample input), was used to explore the space of trees. Figure 10 illustrates the correlation of our cost model with the actual runtimes for solving the 32 subdomain mesh on single and multiple threads. For various number of cores, actual runtimes are plotted against the projected cost of the corresponding trees. The results show that the correlation between the projected cost and expected runtime has an average correlation coefficient of 0.77–0.85 for single and multiple threads. We estimate the parallel costs using the same sequential model. We justify that the parallel runtime correlates with total work during parallel execution. Although we see that correlation does decrease for higher number of threads, our cost model does strongly reflect the execution time of the TreeSolve code. For the remaining test inputs, the cost model shows similar results with high correlation.



(a) 1 Core (b) 2 Cores

(c) 4 Cores. (d) 8 Cores.

Figure 10: Rocket: Cost vs Runtime Correlation of 500 Random Trees

6.2 Performance comparisons

We next compare the execution times for running our heuristics on our three testing inputs. Given each partitioned mesh, our heuristics generate new coupling orders to solve each problem, subject to multiple timescales. In addition to our domain-specific heuristic DS, we evaluate the cost-agnostic tree CA and the Default DT, which is based on the initial METIS numbering. As described in Section 4.2, DT serves as a baseline; the initial coupling schedule produced after METIS is used to partition the initial mesh into subdomains. This coupling schedule also serves as the input to our inspector-executor system. CA uses a top-down approach to produce a new tree, but does not incorporate the domain-specific cost models, while our DS heuristic refines the top-down approach by incorporating knowledge of leaf solve and coupling costs. For each input, we evaluate all three schedules.

Figure 11 shows a CDF of the runtimes for running the 32 subdomain rocket input for various numbers of threads. We compare the execution times of DS, CA, and DT, along with the 500 randomly generated trees for single and multiple threads. Here we observe that DS outperforms all the trees that we tested and we achieved a significant speedup

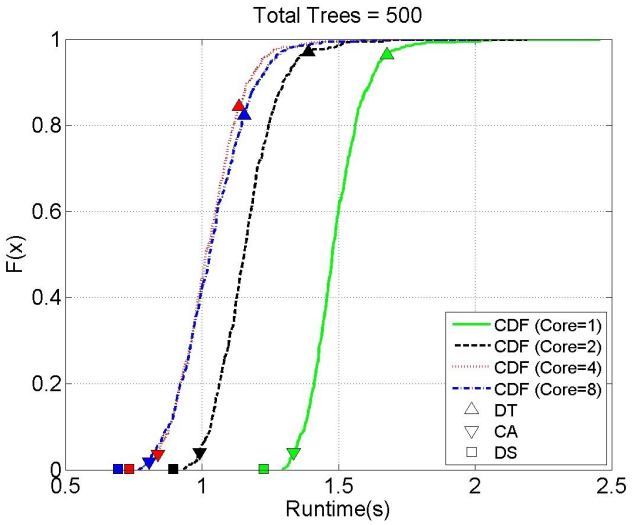


Figure 11: Rocket: CDF of Runtimes for 500 Random Trees with Heuristics

over randomly selected trees in the configuration space. In other words, despite the vast configuration space, by incorporating domain-specific semantic and cost knowledge, we are able to infer a very effective coupling order. We also note that the other evaluated schedules, DT and CA perform worse than our domain-specific schedule. We see that the default schedule is quite slow, while CA yields better results. Nevertheless, our DS approach outperforms the CA schedule by 7 to 20%.

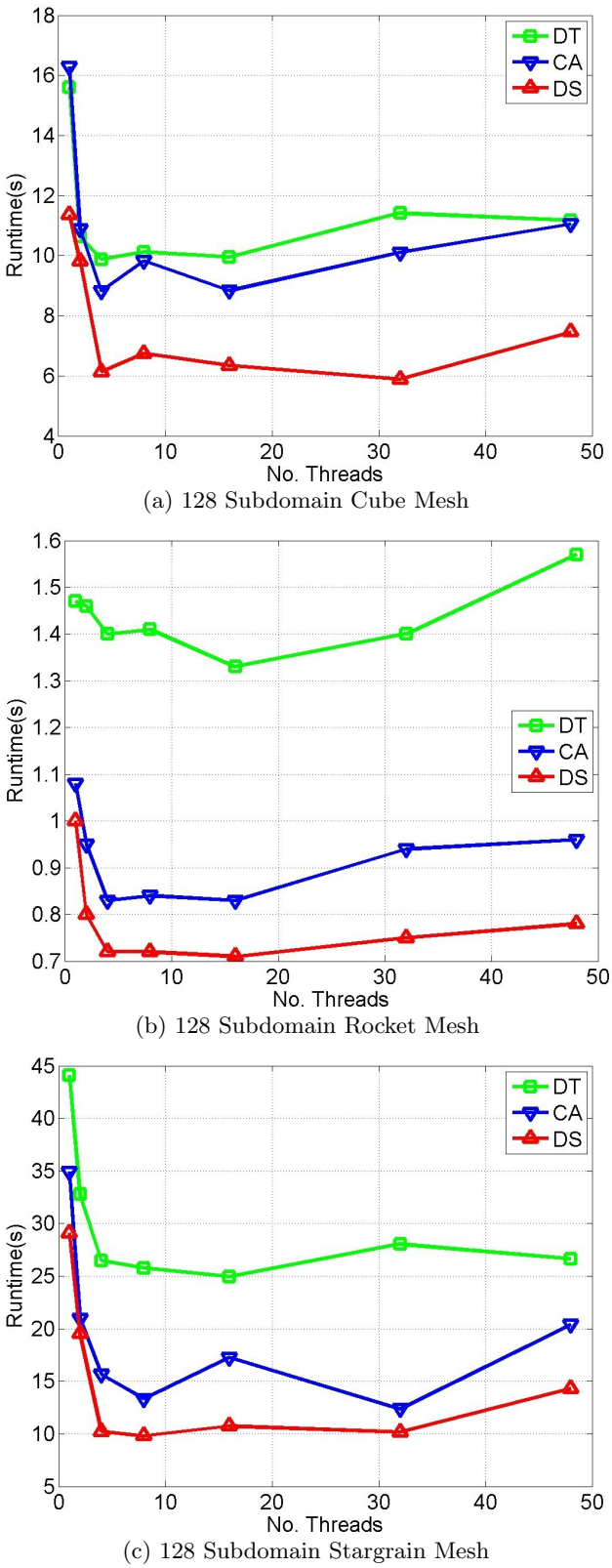
6.3 Parallel performance

Figure 12 compares the execution times for each heuristic across different number of threads for the cube, rocket and stargrain input meshes. We note that in all cases, the DS approach delivers noticeably better performance than the CA and DT. While the specific amount of improvement when using our DS heuristic is problem-dependent, we see that incorporating domain knowledge provides a consistent edge across multiple inputs. Table 2 shows the speedup of CA and DS over the baseline, DT, for each of the inputs at both one and eight threads. Not only does the DS schedule perform the best in single-threaded execution, the advantage increases at higher thread counts; DS provides better scalability.

	No. Thread = 1		No. Thread = 8	
	CA	DS	CA	DS
Cube	0.94	1.33	1.04	1.52
Rocket	1.35	1.47	1.65	1.95
Stargrain	1.26	1.52	1.93	2.62

Table 2: Speedups of CA and DS schedules over baseline DT

For all of the schedules, performance scales up to 4 threads and speedup is obtained when increasing the number of threads. For 8 threads and beyond, the results do not scale well. Using the Cilk profiling tool, we found that for each input, past 4 cores, the runtime of critical path equals the runtime of the solver. This indicates that the parallelism saturates at 8 threads and not much more parallelism can be exploited. Increasing communication costs and parallel overhead plays a role here. This lack of scalability is understandable because the structure of the input is inher-



(a) 128 Subdomain Cube Mesh

(b) 128 Subdomain Rocket Mesh

(c) 128 Subdomain Stargrain Mesh

Figure 12: Execution times across different number of threads

ently imbalanced. Figure 13 shows a tree generated using top-down heuristic using sample 32 subdomain rocket input. We see that the left side of the tree has 4 subdomains and the right side has 28 subdomains. This imbalance is due to the fact that a small number of subdomains are at a lower time scale, which requires fine grain calculations as compared to other parts of the mesh and, due to limitations of the coupling algorithm must be solved separately from the rest of the mesh. In other words, there is a certain amount of inevitable, application-specific load imbalance, especially when there are only a few subdomains run at small time scales. Additionally, the tree-based nature of the algorithm and the fact that coupling occurs atomically limit available parallelism.

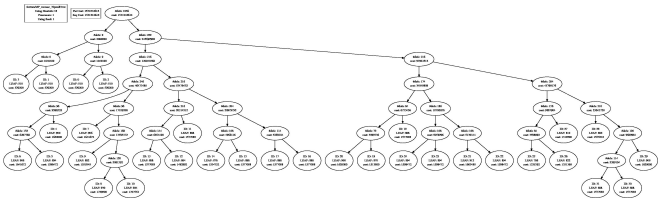


Figure 13: A typical form of a tree generated from input meshes

Available parallelism and scalability across heuristics

A natural question, given the inherent lack of scalability, is whether our scheduling heuristics negatively impact the available parallelism of the program. To investigate this question, we used Cilk to profile the total work, critical path length (span), and amount of parallelism (work/span) afforded by each schedule across all of the inputs. Table 3 summarizes these results.

There are two key take-aways from these results. First, as expected, the DS scheduling heuristic, which incorporates domain-specific semantic and cost information, yields the best single-threaded performance (work), often by a significant margin. This is consistent with the raw performance numbers shown above. Second, despite the heuristics focusing on work minimization, with parallelism only of secondary importance, parallelism is not adversely affected. In fact, on two of the three inputs, DS provides better parallelism than the other two schedules, and on the third input, DS exhibits the same amount of parallelism as CA. The advantage of DS in available parallelism is reflected in the scalability results of Table 2, where DS scales better in practice than the alternative schedules.

Input	Schedule	Work (sec)	Critical Path (sec)	Parallelism
Cube	DT	16.54	10.65	1.55
	CA	17.60	10.02	1.76
	DS	11.94	5.98	2.00
Rocket	DT	2.18	1.55	1.41
	CA	1.93	1.29	1.50
	DS	1.27	0.67	1.90
Stargrain	DT	44.09	23.77	1.85
	CA	34.91	11.12	3.14
	DS	29.12	9.23	3.15

Table 3: Inherent parallelism in coupling schedules

6.4 Inspector phase overhead

The inspector phase, which consists of analyzing the mesh topology and creating the coupling tree, takes approximately as much time as reading the input file. However, long term simulation of physical models may require running thousands or millions of timesteps, which amortizes all other costs aside from the TreeSolve iterations, which is what we measured. Depending on the problem, the time taken by the inspector phase is 0.1% to 1% that of the total executor phase time and less than the time it takes to run TreeSolve for one timestep.

7 Related Work

Domain-specific languages There have been many projects that have leveraged semantic properties of domains to perform domain-specific optimizations on applications that could not be performed by traditional compilers. In the Spiral project, digital signal processing applications are written in a domain-specific language, SPL, and then manipulated using algebraic transformations before optimized code is generated [29]. In the Tensor Contraction Engine, a domain-specific language is used to specify computational chemistry problems as a series of operations on tensors [4]. Because these operations possess simple semantic properties such as commutativity and associativity, the number of possible tensor contraction "schedules" is vast; at compile time, a search procedure identifies the minimal-cost implementation of the given problem and passes that implementation to code-generation routines that perform further optimizations. Other projects have exploited semantic properties in numerically intensive codes written in Matlab—Falcon [25], MaJIC [1, 17] and Telescoping Languages [8, 15]. The key unifying factor in all of these approaches is that they operate at compile-time. In contrast, our optimization approach operates at run-time, as the structure of the problem is not known until the input is seen. Section 8 discusses additional key differences between our approach and prior work, and explores the ramifications of these differences.

Inspector-executor approaches The inspector-executor approach has been used to perform parallelization and optimization in numerous situations where the computational structure of a program is not known until run time. It is often used for parallelization, where the dependence structure of an application is inspected, and a parallelization schedule is determined [21]. In other settings, the dependence structure of a sparse-matrix application can be unfolded to drive *runtime reordering transformations* that aim to improve locality [10, 27]. To our knowledge, most inspector-executor approaches focus on identifying dependences so that an existing computational schedule can be restructured to improve performance. Nevertheless, the underlying algorithm (*i.e.*, the operations performed and the dependence structure) remains the same. In contrast, we use inspection to unfold the computational structure and then *transform* the computation, producing a new computation with an entirely new data flow and dependence structure. Note that because our transformations change the dependence structure, they *necessarily* must exploit semantic properties of the operations (since they violate existing dependences).

8 Lessons Learned

While the study presented in this paper is specific to one method for solving computational mechanics problems, we note that this same general approach applies to a large number of recursive-decomposition-based computational science problems, which arise in domains ranging from computational mechanics to fluid dynamics to peridynamics. Hence, we expect that our inspector-executor system can readily be applied to these other problems.

Moreover, we feel that the approach we took to transforming the problem leads to general lessons that can be applied more broadly. In most instances of domain-specific transformation systems (*e.g.*, [1, 4, 8, 15, 17, 25, 29]), the initial program is written in a domain-specific, high-level language where the semantics needed to drive the optimizations are explicit in the language. In contrast, in our work the computational solver is written in a traditional language (C). The routines that unfold the computational structure are associated with the library interface, and hence can work with any *implementation* of that interface, as well as any application that uses the library.

At a high level, one can view the traditional approach of performing domain-specific optimizations as *optimize-lower*. That is, write an application in a domain-specific language, *optimize* the application using the semantic properties exposed by the DSL then *lower* the program into the target low-level language where other domain-agnostic optimizations can be performed. In contrast, in this work we *started* with an application written in a low-level language. Routines associated with the domain library were used to *lift* the program into a high-level representation that was then transformed and subsequently lowered back into calls to the low-level library routines. Hence, our approach is more aptly termed *lift-optimize-lower*. The primary advantage to a lift-optimize-lower approach is that the source program need not be written in a domain-specific language, but instead can be written using domain libraries familiar to domain scientists.

Another interesting point about our approach is that the transformation routines are completely general. The transformations performed on the computation tree merely take advantage of commutativity and associativity. While these properties are related to the semantics of the domain, the transformations themselves are by no means domain-specific. In fact, the very simplicity of these properties makes them much more broadly applicable. Indeed, the same properties are exploited in systems like the tensor contraction engine (TCE) [4].

This commonality presents an attractive possibility. Consider developing a *high level* intermediate representation, which could represent programs as a composition of abstract operations with domain-agnostic properties such as commutativity and associativity. A program can be lifted into this high level representation, with its domain-specific operations mapped to the appropriate abstract operations (*e.g.*, in our domain, the Couple operation would be mapped to an associative and commutative abstract operation). At this point, a generic transformation engine can act on the high level representation, and the transformed program can be lowered back to the original representation. Even though the transformations are generic, they can be driven by domain-specific cost models as in our application study, effectively performing domain-specific optimizations while remaining domain agnostic. This is similar in spirit to the approach of Willcock *et al.* [28], where compiler transformations that deal with basic types are extended to work on more complex types, allowing for the same transformation pattern to be used in multiple domains; the key difference is that in our scenario, transformations are driven by domain-specific cost models, rather than syntax.

Essentially, rather than performing domain-specific transformations, with new systems being built for each new domain, the kernel of the proposed optimization framework would be completely domain-independent, instantiated only with cost models for operations and routines to translate back and forth from the intermediate representation. Such an approach would lead to a clean separation of concerns, as domain scientists need only consider the high level properties of the operations in their domains, while systems writers can focus on efficient transformations and effective search heuristics. Our case study of a computational mechanics solver is an important proof-of-concept of this approach, as the transformations and search heuristics are oblivious to the details of the domain, relying only on the domain-specific cost model to drive the search. Future work will be to map other domains (such as tensor contraction, fluid dynamics, etc.) to the same run-time transformation framework.

9 Conclusions

This paper presents an effective algorithm for optimizing computational mechanics codes based on recursive domain decomposition for static and dynamic systems. Solving a decomposed problem is represented as solving a binary tree, and the structure of the tree dictates the performance of the solver. We show that the number of possible tree coupling orders is exponentially large for systems even with a moderate number of subdomains. Among the various trees, many do not perform well, and it is not obvious which trees have good performance.

We demonstrate for real problems that our heuristic can generate near optimal coupling trees and execute them efficiently. We show that the inspector's heuristics for building coupling trees consistently produce trees that rank in the 99th percentile of possible trees for problems. Finally, we show that the executor is able to deliver scalable performance on multiple cores as long as there is parallelism to be exploited, providing solutions in less time than randomly selected coupling trees. It automatically provides optimized, parallel implementations of multi-scale computational mechanics problems, allowing domain scientists to take advantage of novel computational techniques without devoting substantial time to the tedious process of optimizing a parallel implementation on a problem-by-problem basis.

Acknowledgments

We would like to thank the anonymous reviewers for their feedback and suggestions. This research is supported by the Department of Energy under contract DE-FC02-12ER26104. M. Hasan Jamal is supported by a Fullbright Fellowship.

References

[1] G. Almási and D. Padua. MaJIC: Compiling MATLAB for speed and responsiveness. In *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, PLDI '02, pages 294–303, 2002.

[2] E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. D. Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen. *LAPACK's user's guide*. SIAM, 1992.

[3] P. Arbenz, U. L. Hetmaniuk, R. B. Lehoucq, and R. S. Tuminaro. A comparison of eigensolvers for large-scale 3d modal analysis using amg-preconditioned iterative methods. *International Journal for Numerical Methods in Engineering*, 64:204–236, 2005.

[4] G. Baumgartner, D. E. Bernholdt, D. Cociorva, R. Harrison, S. Hirata, C.-C. Lam, M. Nooijen, R. Pitzer, J. Ramanujam, and P. Sadayappan. A high-level approach to synthesis of high-performance codes for quantum chemistry. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, Supercomputing '02, pages 1–10, 2002.

[5] J. K. Bennighof and R. B. Lehoucq. An automated multilevel substructuring method for eigenspace computation in linear elastodynamics. *SIAM Journal on Scientific Computing*, 25:2084–2106, 2004.

[6] S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and C. Whaley. An updated set of Basic Linear Algebra Subprograms (BLAS). *ACM Transactions on Mathematical Software*, 28(2):135–151, June 2002.

[7] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An efficient multithreaded runtime system. In *Proceedings of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP '95, pages 207–216, 1995.

[8] A. Chauhan and K. Kennedy. Optimizing strategies for telescoping languages: Procedure strength reduction and procedure vectorization, year = 2001. In *ICS '01: Proceedings of the 15th International Conference on Supercomputing*, pages 92–101.

[9] R. Das, M. Uysal, J. Saltz, and Y.-S. Hwang. Communication optimizations for irregular scientific computations on distributed memory architectures. *Journal of Parallel and Distributed Computing*, 22:462–479, 1993.

[10] C. Ding and K. Kennedy. Improving cache performance in dynamic applications through data and computation reorganization at runtime. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 229–241, 1999.

[11] J. Fish. Bridging the scales in nano engineering and science. *Journal of Nanoparticle Research*, 8(5):577–594, 2006.

[12] V. Gravemeier, S. Lenz, and W. A. Wall. Towards a taxonomy for multiscale methods in computational mechanics: Building blocks of existing methods. *Computational Mechanics*, 41(2):279–291, 2008.

[13] W. Hackbush. On the computation of approximate eigenvalues and eigenfunctions of elliptic operators by means of a multigrid method. *SIAM Journal on Numerical Analysis*, 16:201–215, 1979.

[14] G. Karypis and V. Kumar. Metis - unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report, Department of Computer Science, University of Minnesota, 1995.

[15] K. Kennedy, B. Broom, K. Cooper, J. Dongarra, R. Fowler, D. Gannon, L. Johnsson, J. Mellor-Crummey, and L. Torczon. Telescoping languages: A strategy for automatic generation of scientific problem-solving systems from annotated libraries. *Journal of Parallel and Distributed Computing*, 61(12):1803–1826, 2001.

[16] Z. Li, Y. Saad, and M. Sosonkina. pARMS: a parallel version of the algebraic recursive multilevel solver. *Numerical Linear Algebra with Applications*, 10:485–509, 2003.

[17] V. Menon and K. Pingali. A case for source-level transformations in MATLAB. In *Proceedings of the 2nd conference on Domain-Specific Languages*, DSL '99, pages 53–65, 1999.

[18] J. Michopoulos, C. Farhat, and J. Fish. Modeling and simulation of multiphysics systems. *Journal of Computing and Information Science in Engineering*, 5:198, 2005.

[19] N. M. Newmark. A method of computation for structural dynamics. *Journal of Engineering Mechanics, ASCE*, 85:67–94, 1959.

[20] C. Papalukopoulos and S. Natsiavas. Dynamics of large scale mechanical models using multilevel substructuring. *Journal of Computational and Nonlinear Dynamics, ASME*, 2:40–51, 2007. Transactions of the ASME.

[21] R. Ponnusamy, J. Saltz, and A. Choudhary. Runtime compilation techniques for data partitioning and communication schedule reuse. In *Supercomputing '93: Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*, 1993.

[22] A. Prakash. *Multi-time-step domain decomposition and coupling methods for non-linear structural dynamics*. PhD thesis, University of Illinois at Urbana-Champaign, 2007.

[23] A. Prakash and K. D. Hjelmstad. A FETI based multi-time-step coupling method for Newmark schemes in structural dynamics. *International Journal for Numerical Methods in Engineering*, 61:2183–2204, 2004.

[24] A. Prakash and K. D. Hjelmstad. A multi-time-step coupling method for Newmark schemes in structural dynamics. In *proceedings of McMat2005: Joint ASME/ASCE/SES Conference on Mechanics and Materials*, June 2005.

[25] L. D. Rose and D. Padua. A MATLAB to Fortran 90 Translator and its Effectiveness. In *International Conference on Supercomputing*, pages 309–316, 1996.

[26] Y. Saad and J. Zhang. BILUTM: A domain-based multilevel block ILUT preconditioner for general sparse matrices. *SIAM Journal on Matrix Analysis and Applications*, 21:279–299, 1999.

[27] M. M. Strout, L. Carter, and J. Ferrante. Rescheduling for locality in sparse matrix computations. In *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01, pages 137–148, 2001.

[28] J. J. Willcock, A. Lumsdaine, and D. J. Quinlan. Reusable, generic program analyses and transformations. In *Proceedings of the 8th International Conference on Generative Programming and Component Engineering*, GPCE '09, pages 5–14, 2009.

[29] J. Xiong, J. Johnson, R. W. Johnson, and D. Padua. SPL: A language and compiler for DSP algorithms. In *Proceedings of the Programming Languages Design and Implementation (PLDI)*, pages 298–308, 2001.