2021 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW) | 978-1-6654-3577-2/20/$31.00 ©2021 IEEE | DOI: 10.1109/IPDPSW52791.2021.00103

Load balancing for distributed nonlocal models within asynchronous many-task systems

Pranav Gadikar
Computer Science Department
Indian Institute of Technology Madras, Chennai, India
cs16b115@smail.iitm.ac.in

Patrick Diehl

Center for Computation and Technology
Louisiana State University, Baton Rouge, USA
patrickdiehl@lsu.edu

Prashant K. Jha

Oden Institute for Computational Engineering and Sciences
The University of Texas at Austin, Austin, USA
pjha@utexas.edu

***Abstract*—In this work, we consider the challenges of developing a distributed solver for models based on nonlocal interactions. In nonlocal models, in contrast to the local model, such as the wave and heat partial differential equations, the material interacts with neighboring points on a larger-length scale compared to the mesh discretization. In developing a fully distributed solver, the interaction over a length scale greater than mesh size introduces additional data dependencies among the compute nodes and communication bottleneck. In this work, we carefully look at these challenges in the context of nonlocal models; to keep the presentation specific to the computational issues, we consider a nonlocal heat equation in a 2d setting. In particular, the distributed framework we propose pays greater attention to the bottleneck of data communication and the dynamic balancing of loads among nodes with varying compute capacity. For load balancing, we propose a novel framework that assesses the compute capacity of nodes and dynamically balances the load so that the idle time among nodes is minimal. Our framework relies heavily on HPX library, an asynchronous many-task run time system. We present several results demonstrating the effectiveness of the proposed framework.**

***Index Terms*—Nonlocal Computational Models, Load Balancing, AMT, HPX, Parallel and Distributed Computing**

I. Introduction

Nonlocal models are seen in various fields; peridynamics for modeling fracture and damage in solid media [1], [2], nonlocal heat and diffusion equation [3], cell-cell adhesion in computational biology [4], [5], and recently application of peridynamics to granular media [6]. Unlike the models based on the local interaction, for instance, wave and heat partial differential equation, the nonlocal models include the interaction of material points over a finite length scale; as a result, after the spatial discretization of the model, the discrete points interact over a length scale that is larger than the mesh size (minimum spacing between the discrete points). In contrast, in the discretization of wave or heat equations, the interaction is local, *i.e.*, the discrete point only interacts with the nearest neighbor points. The larger interaction length scale introduces a major challenge in implementing a *distributed solver*; a node has stronger data dependency with the neighboring node which means that the message size to exchange the ghost zones increases. Thus, we need to overlap communication with computation to address this challenge. We carefully look at this issue in the context of the nonlocal heat equation, which is a simple nonlocal model, but still general enough that framework developed in this work can be applied to more complex models such as peridynamics [1], [2]. We investigate the following: (*i*) *Minimizing data exchange* – here we use the METIS library [7] to generate optimal partitions, with a minimum data exchange between nodes. (*ii*) *Hiding data exchange time* – the partitions are divided such that we have independent sub-partitions with sub-partitions depending on the ghost zones on a different node. Computations on the independent partitions (partitions which do not depend on the data of other compute nodes) are performed asynchronously while waiting for the data from the neighboring nodes for computations on the dependent partitions. (*iii*) *Load balancing* – the partitions are redistributed among the nodes to minimize the waiting time for the faster nodes. The shape of the partitioning obtained by the METIS library is preserved to the maximum possible extent, in order to reduce the data dependencies.

For the asynchronous function computation and synchronization, we rely heavily on the HPX [8] library; HPX is the C++ standard library for parallelism and concurrency for high-level programming abstractions and provides wait-free asynchronous execution and futurization for synchronization. One key feature, we utilize is that HPX resolves the data dependency and generates an asynchronous execution graph, which allows us to implicitly overlap the communication and the computation.

The paper is structured as follows: We present the related work in Section II. In Section III we briefly introduce the nonlocal heat equation and in Section IV we highlight

978-1-6654-3577-2/21/$31.00 ©2021 IEEE
DOI 10.1109/IPDPSW52791.2021.00103

Authorized licensed use limited to: University of Texas at Austin. Downloaded on July 08,2021 at 02:18:11 UTC from IEEE Xplore. Restrictions apply.

the key challenges in the development of *distributed solver*. We discuss the key features of HPX used in our implementation in Section V and then present our core technical contributions in Section VI. We propose a novel load balancing algorithm in Section VII. We demonstrate the scaling results of our *distributed solver* and load balancing algorithm in Section VIII and conclude in Section IX.

II. Related Work

Asynchronous many-task systems (AMT): Many asynchronous many-task (AMT) systems have been developed recently. However, we focus on those with distributed capabilities since the current work focuses on the domain decomposition and load balancing for the distributed case. Here, the most notable are: Uintah [9], Chapel [10], Charm++ [11], Kokkos [12], Legion [13], and PaRSEC [14]. According to [15], only HPX has a C++ standard compatible API and has the highest technology readiness level.

Load balancing: To the best of our knowledge, there are no load balancing algorithms specifically designed to handle the constraints of minimizing the data dependencies across multiple computational nodes in *distributed solvers* for nonlocal models. [16] proposed a load balancing algorithm in the Charm++ library that shares the underlying implementation concepts with HPX. [16] proposed a hardware-topology aware load balancing to minimize the application communication costs. In a slightly different context of single computational node multi-core systems, [17] proposes to use the ideas of work-stealing and dynamic coarsening to improve the data locality on multiple cores while load balancing.

III. Brief Introduction to the Nonlocal Heat (Diffusion) Equation

In this section, we briefly introduce the nonlocal heat equation. We chose this model due to its simplicity, however, the algorithms and ideas discussed in this work should work for a more complex nonlocal fracture model such as peridynamics. We consider a two dimensional nonlocal diffusion equation for the temperature $u : [0, T] \times D \mapsto \mathbf{R}$ over an square domain $D = [0, 1]^2$ for the time interval $[0, T]$. We impose a zero temperature boundary condition on the boundary of D and specify a heat source $b : [0, T] \times D \mapsto \mathbf{R}$ as a function of points on the domain and time. Let $\epsilon > 0$ be the nonlocal length scale. The temperature field satisfies

$$\frac{\partial u(t,x)}{\partial t} = b(t,x) + c \int_{B_\epsilon(x)} J\left(\frac{|y-x|}{\epsilon}\right)(u(t,y) - u(t,x))\,\mathrm{d}y, \quad (1)$$

where $|y-x|$ denotes the Euclidean norm of vector $y-x$ in 2d, $B_\epsilon(x) = \{y : |y-x| \leq \epsilon\}$ the ball of radius ϵ centered at x, and $J = J(r)$ the influence function. We consider $J = 1$ for simplicity. Here, the constant c is related to the heat conductivity k of the material as follows

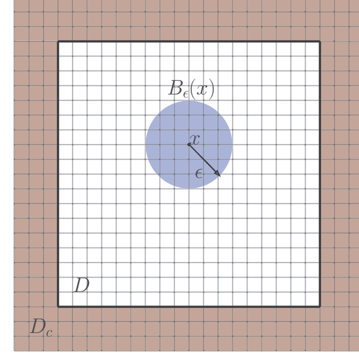

Fig. 1. Material domain D with the nonlocal boundary D_c surrounding D. Figure shows typical material point $x \in D$ and its neighborhood region, a ball of radius ϵ centered at x, $B_\epsilon(x)$. We also show the spatial discretization of domains D and D_c through a uniform grid.

$$c = \begin{cases} \frac{k}{\epsilon^3 M_2}, & \text{when dimension } d = 1 \\ \frac{2k}{\pi \epsilon^4 M_3}, & \text{when dimension } d = 2, \end{cases} \quad (2)$$

where $M_i = \int_D J(r) r^i \,\mathrm{d}r$ is the ith moment of the influence function. (2) can be derived by substituting the Taylor series expansion, $u(t,y) = u(t,x) + \nabla u(t,x) \cdot (y-x) + \frac{1}{2}\nabla^2 u(t,x) : (y-x) \otimes (y-x) + O(\epsilon^3)$, in (1) and comparing the Laplacian term with that from the classical heat equation. We refer to [3] for more discussion on the nonlocal diffusion equation. The initial condition is given by

$$u(0,x) = u_0(x) \qquad \forall x \in D. \quad (3)$$

The boundary condition is typically prescribed over the region of finite area (in 2d) or volume (in 3d). Let $D_c = (-\epsilon, 1+\epsilon)^2 - D$ is the annuls square obtained by removing D from the bigger square $(-\epsilon, 1+\epsilon)^2$, see Figure 1. We apply the zero temperature boundary condition on D_c, *i.e.*,

$$u(t,x) = 0 \qquad \forall x \in D_c \text{ and } \forall t \in [0, T]. \quad (4)$$

To solve for the temperature u for a given source b, we consider a finite difference in space and forward-Euler in time discretization of (1). We discuss the discretization next.

A. Finite Difference Approximation

We discretize the domain using the uniform grid with grid size $h > 0$, see Figure 1. Let D_h and D_{c_h} denotes the spatial coordinates of grid points after discretization of D and D_c. We consider an index set $K \subset \mathbf{Z}^2$ such that for $i \in K$, $x_i = hi \in D_h$. Similarly, we consider an index set $K_c \subset \mathbf{Z}^2$ such that for $i \in K_c$, $x_i = hi \in D_{c_h}$. Let $\{0, t_1 = \Delta t, t_2 = 2\Delta t, ..., t_N = N\Delta t\}$, such that $t_N \leq T$, is the discretization of the time interval $[0, T]$. Here Δt is the size of the timestep.

Authorized licensed use limited to: University of Texas at Austin. Downloaded on July 08,2021 at 02:18:11 UTC from IEEE Xplore. Restrictions apply.

Let $\hat{u}_i^k$ denote the temperature at grid point $i \in K \cup K_c$ and at time $t_k = k\Delta t$. Using the finite-difference approximation and forward-Euler time-stepping scheme, we write the discrete system of equations for $\hat{u}_i^k$, for all $i \in K$ and $1 \leq k \leq N$,

$$\frac{\hat{u}_i^{k+1} - \hat{u}_i^k}{\Delta t} = b(t^k, x_i) + c \sum_{\substack{j \in K \cup K_c, \\ |x_j - x_i| \leq \epsilon}} J(|x_j - x_i|/\epsilon)(\hat{u}_j^k - \hat{u}_i^k)V_j. \quad (5)$$

The boundary condition over D_c translates to

$$\hat{u}_i^k = 0, \qquad \forall k \in K_c,\, 0 \leq k \leq N.$$

To apply the initial condition, we set $\hat{u}_i^0 = u_0(x_i)$ for all $i \in K$. In the above, V_j is the area occupied by the grid point j in the mesh. For the uniform discretization with the grid size h, we have $V_j = h^2$.

B. Exact Solution and Numerical Error

To test the implementation of the discretization scheme in Subsection III-A, we consider a method of constructing the exact solution. Let

$$w(t, x) = \cos(2\pi t)\sin(2\pi x_1)\sin(2\pi x_2)$$

when $x \in D$ and $w(t, x) = 0$ when $x \notin D$. We consider an external heat source of the form:

$$b(t, x) = \frac{\partial w(t, x)}{\partial t} - c \int_{B_\epsilon(x)} J(|y - x|/\epsilon)(w(t, y) - w(t, x))\, \mathrm{d}y. \quad (6)$$

With b given by above and the initial condition

$$u_0(x) = w(0, x) = \sin(2\pi x_1)\sin(2\pi x_2),$$

we can show that $u(t, x) = w(t, x)$ is the exact solution of (1). Next, we define the numerical error.

Suppose $\bar{u}(t, x)$ is the exact solution and $\hat{u}_i^k$ for $0 \leq k \leq N$ and $i \in K$ is the numerical solution. The error at time t^k is taken as

$$e^k = h^d \sum_{i \in K} |\bar{u}(t^k, x_i) - \hat{u}_i^k|^2, \quad (7)$$

where $d = 1, 2$ is the dimension. The total error can be defined as the sum of errors e^k, *i.e.*, $e = \sum_{0 \leq k \leq N} e^k$.

IV. Problem Statement and Formalism

Fully parallelizing a serial implementation of the nonlocal equation to a distributed version deployed on an Asynchronous Many-Task System (AMT) involves numerous challenges. These challenges are primarily related to the work distribution across multiple computational nodes, minimizing the idle time spent on data exchange among computational nodes and ensuring maximum efficiency across all computational nodes. Designing and implementing a *distributed solver* that addresses and overcomes the challenges listed below is very critical to ensure optimal performance:-

1) ***Mesh partitioning*** – breaking down the main problem into smaller sub-problems that can be distributed across multiple computational nodes. Each computational node may contain multiple such sub-problems. This simple idea of unitized work greatly helps to simplify the work distribution and load balancing in AMTs, where there are multiple computational nodes within a cluster and each computational node consists of multiple CPUs. In such a scenario, each of the sub-problems can be easily assigned to different threads within a single computational node to utilize multiple CPUs on a single computational node.
2) ***Minimizing data exchange*** – distributing the sub-problems among multiple computational nodes to achieve minimum data dependencies. The efficiency of a *distributed solver* is limited by the data dependency across the computational nodes. Minimum data dependency across the computational nodes ensures minimum data exchange time across different computational nodes and better scaling.
3) ***Hiding data exchange time*** – doing useful computation while waiting for the data from the neighboring computational nodes. Despite the optimal work distribution, the computational nodes need to exchange data required for the computation of the nonlocal solver. To avoid keeping the computational nodes idle while the data is being exchanged, it is possible to perform computations on portion of owned domain that do not depend on the data of other nodes. This asynchronous-style computation helps us to hide the data exchange time and to ensure optimal performance of the *distributed solver*.
4) ***Load balancing*** – redistribution of sub-problems across multiple computational nodes according to their load to minimize the waiting time across the faster nodes. Compute capacity of the individual computational nodes may vary with time, either due to scheduling of some other task or due to the intrinsic behaviour of the nonlocal model. In such a scenario, the updated compute capacity of individual nodes needs to be accounted and the load needs to be balanced so that the faster nodes do not sit idle.

We describe our solution to address all challenges above in Section VI and propose a novel load balancing algorithm in Section VII.

Formalism

We define the following terms used throughout this and the following sections:

- ***Distributed solver*** – the time-stepping scheme (5) where we start with the given initial temperature at discrete points, $\hat{u}^0$, in the domain and apply (5) to compute the temperature at successive times $t_1 = \Delta t, t_2 =$

Authorized licensed use limited to: University of Texas at Austin. Downloaded on July 08,2021 at 02:18:11 UTC from IEEE Xplore. Restrictions apply.

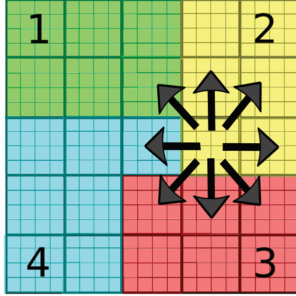

Fig. 2. Discretization of the domain (grid with smaller spacing) and SDs (grid with larger spacing, note the thicker lines). Here, we consider 4 compute nodes and 25 square SDs. Color indicates the owner of the SDs. As we can see a typical SD consists of 4×4 DPs (discretized points). To compute the updated temperature at DPs within an SD_i, the data from neighboring SDs within the ϵ distance is required. If the size of the SD is bigger than ϵ, we see that any SD needs to communicate only with the immediate neighboring SDs.

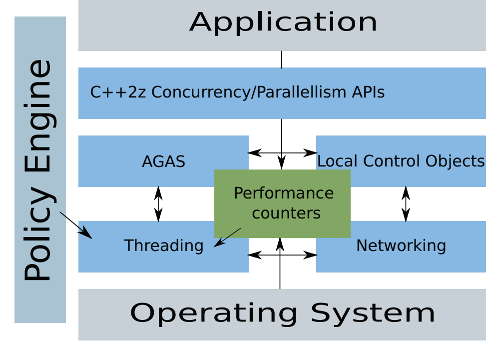

Fig. 3. Sketch of the HPX's architecture and its components: Local Control Objects (LCOs), Threading sub-system, Networking, performance counters, and Active Global Address Space (AGAS). We briefly introduce the performance counter and the local control objects components in this section. For all other components, we refer to [8]. Adapted from [8].

$2\Delta t, .., t_k = k\Delta t$. At each timestep k, the right-hand side term of (5) must be computed first to compute the temperature at the next timestep t_{k+1}. This involves an underlying challenge of the data dependency for all the points within ϵ distance of a given point x_i. Since ϵ is typically higher than $2h$ where h is the grid size, each computational node needs to scatter and gather the temperature values at the degree of freedoms owned by the neighboring computational nodes.

- ***Sub-problem*** (SP) – the region of the material domain managed by a computational node; *i.e.* the computational node is the owner of the degrees of freedom in that region and computes the temperature following scheme (5). For example, in Figure 2, the region colored with green, yellow, cyan, and red shows four SPs.
- ***Sub-domain*** (SD) – the region of the material domain that is processed and computed independently within a given computational node. SDs are subsets of SP in a given node; *i.e.* SP is further divided into multiple SDs. In this work, we always consider a square subdomain. By the size of SP, we mean the number of SDs it consists of. In Figure 2, the SDs are grids with larger spacing (see thick lines); for instance, node 1 has 6 SDs.
- ***Discretized point*** (DP) – each SD is responsible for the discrete points within it. To be able to apply (5) and compute the temperature at the discrete points in a given SD_i, SD_i will have to exchange the data from neighboring SDs. When all neighboring SDs of SD_i are in the same computational node as SD_i, no special data exchange method is required. However, when some SDs are in neighboring computational nodes, a proper communication method is required. For instance, in Figure 2, the green SD (owned by node 1), on top and near yellow region, depends on the data owned by node 2.

V. HPX Basics

Figure 3 sketches HPX's architecture and its components. We briefly introduce the local control objects and the performance counters that we use in our implementation. For all other components, we refer to [8]. The performance counters component provides a uniform API of a globally accessible counter to access the performance in-situ [18]. All counters are registered within the Active Global Address Space (AGAS) [19] to poll the counters during the run time of the application.

The Local Control Objects (LCOs) component provides the features `hpx::async` and `hpx::future` for the asynchronous execution and synchronization. Listing 1 shows how to compute $a + b + c + d$ asynchronously using HPX. In Line 2 the function to compute two integers is defined. In Lines 6–7 the function `add` is launched asynchronously using `hpx::async` which means the function is executed on one thread and immediately returns a `hpx::future<int>` which contains the result of the computation once the thread is finished. Thus, the second function call is launched and the two function calls are executed concurrently. In Line 9 a blocking synchronization of the two asynchronous function calls is needed to obtain the results using the `.get()` function.

Listing 1. Example to illustrate the usage of hpx::future and hpx::async

```
1: // Function to add two integers
2: int add (int one, int second){
3:       return one + second;
4: }
5: // Launch two function calls asynchronously
6: hpx::future<int> a_add_b = hpx::async(add, a, b);
7: hpx::future<int> c_add_d = hpx::async(add, c, d);
8: // Synchronization to compute the result
9: int result = a_add_b.get() + c_add_d.get();
```

VI. Distributed Solver Implementation

In this section, we present our approach to solve the various challenges associated with the *distributed solver* discussed in Section IV. To understand the challenges and

Authorized licensed use limited to: University of Texas at Austin. Downloaded on July 08,2021 at 02:18:11 UTC from IEEE Xplore. Restrictions apply.

compute the gain in going from a serial (single computational node and single-threaded) implementation to a fully distributed and load-balanced implementation, we first implemented a single-threaded version. Second, we extended the serial implementation to a multi-threaded version using asynchronous execution, *e.g.,* futurization. Third, we implemented a fully distributed asynchronous solver for (5). In this extension, we rely on the HPX's implementation of the distributed asynchronous computation. We demonstrate the schematics of the fully distributed and load-balanced time-stepping scheme of nonlocal models (5) in Figure 4. In Figure 4(right), various components that tackle the challenges listed in Section IV and their interactions are shown. We now discuss our approach that address the challenges described in Section IV:

A. Mesh Partitioning

We propose a simple decomposition of the complete square domain into smaller squares (SDs). As shown in Figure 2, we divide the discretized square domain into 5 x 5 SDs, *i.e.*, total 25 SDs are created. Each SD consists of 4 x 4 DPs and is responsible for the computation associated to these DPs. We consider the computation of an SD as a unit of work; the size of SD controls the communication burden and the size of work (larger SD will have more DPs and hence more computation) and therefore the size of an SD can be tuned to achieve maximum performance.

B. Minimizing the Data Exchange

The efficiency of a *distributed solver* is limited by the data dependency across the computational nodes. For instance, in Figure 2, the SDs belonging to a computational node *2* depend on the data with the neighboring SDs belonging to computational nodes *1*, *3* and *4*. We propose to use the METIS library [7] to address this challenge. METIS is a set of serial programs for providing fast and high-quality partitioning of finite element meshes and graphs. Specifically, we use the `METIS_PartMeshDual` function which ensures that the resulting partition is optimal and results in minimum data exchange across the computational nodes during the execution. For instance, we use `METIS_PartMeshDual` to distribute 25 SDs across 4 nodes as shown in Figure 2. The contiguous partitioning ensures that most of the bordering elements would exchange data with the SDs belonging to the same computational node, thus, reducing the data exchange.

C. Hiding the Data Exchange Time

To hide the data exchange time, we propose to divide the set of DPs of any SD into two cases (also see Figure 5):

- ***Case 1*** consists of DPs that depend on the data from SDs of other nodes. Consider a point x_i near label **C1** in Figure 5; the ϵ neighborhood of x_i consists of some DPs on SDs of other computational nodes. At every timestep, to compute the right-hand side in (5), the updated temperature at of DPs such as x_j of neighboring SDs must be collected.
- ***Case 2*** consists of the remaining DPs. The computation associated with these DPs is independent of the DPs belonging to other computational nodes.

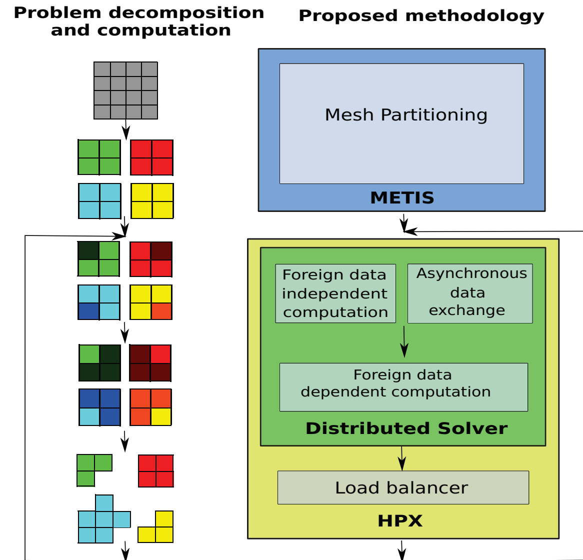

Fig. 4. Sketch of the proposed distributed framework (on right) and the decomposition and computation problem flow in a distributed cluster (on left). **Right**: For the decomposition of the domain and the distribution of the problems across computational nodes, METIS library was utilized. Foreign data (foreign data is the data that is not available on the current node) independent computation is done while waiting for the data to be available from the neighboring nodes. Foreign data-dependent computation is done once the data from the neighboring nodes is available. Our load balancing algorithm ensures that all the computational nodes have the SP (sub-problem) size in proportion to their computational power. **Left**: The problem is first decomposed into multiple SPs distributed to nodes (colors show the node responsible for the SP). In the second step, see the third figure from the top, while waiting for data from the neighboring nodes, we perform computation on those SDs (sub-domains) which do not depend on the data of neighboring processors; the dark-colored square indicates that computation is being performed on the DPs (discretized points) within it. In the third step, see the fourth figure from the top, we now perform computation on those regions of SP that depended on the neighboring nodes' data; the dark-colored squares are now different. Finally, at the end of the timestep, we check for the load on each compute node, and if needed redistribute the SDs (*i.e.* change the SP of individual nodes) by applying the load balancing step discussed in Section VII.

At every timestep, we propose to compute the data for the DPs belonging to *Case 2* first, while the data for DPs belonging to *Case 1* becomes available. This ordering ensures an effective hiding of the data sharing time for the points corresponding to *Case 1*.

VII. Load Balancing

The issue of load balancing often becomes crucial in nonlocal models, especially in nonlocal fracture models [1]. In

Authorized licensed use limited to: University of Texas at Austin. Downloaded on July 08,2021 at 02:18:11 UTC from IEEE Xplore. Restrictions apply.

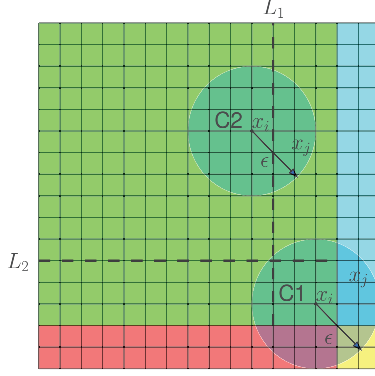

Fig. 5. Different cases of computation in one SD (green square). The neighboring SDs owned by different nodes is also shown. The DPs in green SD can be divided into two parts: 1) DPs on right of line L_1 or bottom of L_2 (see typical point x_i near label **C1**) and 2) remaining DPs in green SD (see typical point x_i near label **C2**). At timestep k, we can perform computation on DPs independently for case 2. Whereas for DPs associate to case 1 the computation depends on the data from neighboring nodes.

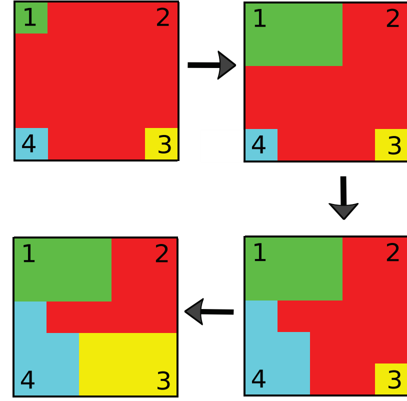

Fig. 6. Redistribution of the *sub-domains* (SDs) among the computational nodes to balance the load in the topological order. Node 1 borrows the data from the non-visited node 2, followed by nodes 4 and 3. Each node borrows SDs uniformly in all the directions to retain a contiguous locality of the SDs.

fracture/crack problems, the computation in regions containing the crack is different from the region not containing it; the crack line (in 2d or surface in 3d) divides the two regions such that the points on either side of the crack line do not interact with each other. In our terminology, the SDs which contain the portion of crack will have reduced computational burden as compared to SDs not containing the crack. Another instance where load-balancing could be crucial is when the computational capacity of a node is time-dependent. To the best of our knowledge, there are no well-known load balancing algorithms for nonlocal models in AMT systems. In this section, we propose a novel load balancing algorithm useful for the cases discussed above. The only assumption we make is that the data exchange times are negligible compared to the amount of time spent on the computation. Our method of hiding the data exchange times in Subsection VI-C makes this assumption realistic and reasonable. We now discuss the key aspects of the load balancing algorithm:

*a) **Calculating the Load Imbalance**:* To measure the load imbalance, we use `hpx::performance_counters ::busy_time` which reports the fraction of the time the node was busy doing computation against the total time the node was active. Between successive load balancing iterations, the performance counter is reset to have the same total time span for all the computational nodes in the cluster. Significantly different busy times for the different computational nodes indicate a load imbalance. In an ideal case, the busy time should be the same for all nodes. To achieve the close to perfect load-balanced state, it is necessary to assign SDs to individual nodes based on their compute capacity. To measure the compute capacity of a particular computational node N_i, we consider the formula:

$$\text{Power}(N_i) = \frac{\bar{\text{SD}}(N_i)}{\text{Busy Time}(N_i)}, \tag{8}$$

where $\bar{\text{SD}}(N_i)$ denotes the number of SDs on node N_i. Clearly, a more powerful node can handle a larger number of SDs. Thus, *Power* can be seen as an accurate measure to quantify the compute capacity. We calculate the load imbalance in terms of SDs using the following formula:

$$\text{LoadImbalance}(N_i) = E(N_i) - \bar{\text{SD}}(N_i), \tag{9}$$

where $E(N_i)$ is given by:

$$E(N_i) = \text{Total no. of SD} \times \frac{\text{Power}(N_i)}{\sum_j \text{Power}(N_j)}. \tag{10}$$

*b) **Balancing the Load**:* We want to distribute the total work, *i.e.*, all SDs, keeping in mind the compute capacity, see (9), of the individual nodes. Towards this, we propose our novel load balancing algorithm; the main idea is to borrow/lend SDs from/to a computational node, that owns the adjacent SDs. For instance, in Figure 6(top-left), the computational node 1 borrows SDs from 2 to reduce the compute burden from 2. Note that the SDs belonging to 1 share their boundaries with the SDs of 2. This helps in preserving a contiguous locality of SDs, thereby minimizing the data exchange across computational nodes.

The load balancing algorithm is presented briefly in Algorithm 1. We calculate the load imbalance for each computational node by (9) in Lines 2–12. In Lines 13–18, we model the data dependencies across computational nodes using a tree T; each node N_i in the tree has a one-to-one correspondence with the computation node, and, an edge e between two nodes of the tree denotes the data dependencies among them. An edge between nodes N_i and N_j can exist only if there is an SD in one of the

Authorized licensed use limited to: University of Texas at Austin. Downloaded on July 08,2021 at 02:18:11 UTC from IEEE Xplore. Restrictions apply.

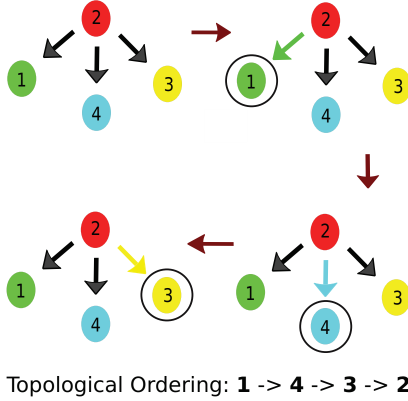

Fig. 7. Data exchange among different pair of the nodes of the data dependency tree in the topological order. Topological ordering generated for the above figure is $1 \to 4 \to 3 \to 2$. Hence, node 1 borrows SDs from its neighbouring node 2 to balance the load. Similarly, node 4 and then node 3 borrow SDs from their neighbouring node.

nodes such that SD has data dependencies with SDs of the other node. For instance, the scenario in Figure 6(top-left) translates to the tree shown in Figure 7(top-left). Note that there are multiple such trees possible; we use one of the possible trees. In Line 19 of Algorithm 1, we obtain an ordering of the nodes to perform the data redistribution in a *least data-dependency first* fashion. The topological ordering used for the tree in Figure 7 is $1 \to 4 \to 3 \to 2$. Coming back to the algorithm, in Lines 13 - 18, we perform the actual data redistribution among the nodes in the topological order. Topological ordering helps to borrow / lend SDs from different computational nodes optimally without unbalancing the already balanced nodes. Figure 7 shows the redistribution of the SDs among the tree nodes in the topological order and Figure 6 shows the change in the SD distribution across nodes. Note that each node N_i borrows / lends SDs to all the non-visited adjacent nodes N_j, uniformly in all the spatial directions in the domain. METIS generates an optimal distribution of SDs across computational nodes for minimum data exchange. It is important to preserve the shape of SPs obtained using METIS mesh partitioning in Subsection VI-B to obtain the best scaling. For instance, the node 1 in Figure 6 borrows SDs of 2 from all spatial directions to minimize the data dependencies. At the end of the load balancing iteration, we reset the performance counters to obtain the correct performance measurements for the new load distribution for the next load balancing step; see Line 35.

VIII. Results and Discussion

All simulations were run on nodes with Intel Skylake CPU containing 40 cores and 96 GB of memory. The code was compiled with MPI 1.10.1, GCC 10.2.0, and HPX 1.4.1.

Algorithm 1: Load Balancing Algorithm

```
1:  ▷ Let N is the number of computational nodes
2:  ▷ Compute number of SDs (sub-domains) on each node
3:  compute NumSubDomains[i] for i ∈ [0, N − 1]
4:  ▷ Compute computational power of node using (8)
5:  compute Power[i] for i ∈ [0, N − 1]
6:  ▷ Compute expected number of SDs using (10) and Power
7:  compute ExpSubdomains[i] for i ∈ [0, N − 1]
8:  ▷ Compute load imbalance using (9). Positive value indicate
    the load on node is smaller and negative otherwise.
9:  for each integer i ∈ [0, N − 1] do
10:     LoadImbalance[i] = ExpSubdomains[i]
11:         - NumSubDomains[i]
12: end for
13: ▷ Set minimum imbalanced load node as root R of tree T
14: Root R = argmin(LoadImbalance)
15: ▷ Each node is represented by some node in the tree T
16: ▷ An edge e between two nodes exists if there is SD that is
    in one of the node and is adjacent to the SP (sub-problem,
    set of SDs) of other node
17: Tree T = construct_tree(R)
18: ▷ Give root R and tree T, get_topological_sort(R, T)
    returns node ids to be processed in next step
19: orderedNodes[N] = get_topological_sort(R, T)
20: ▷ Each node n borrows SDs only from non-visited nodes to
    increase its territory (set of SDs) uniformly in all the
    directions
21: for each node i ∈ orderedNodes[0, N − 1] do
22:     if LoadImbalance[i] == 0 then
23:         continue
24:     end if
25:     ▷ Compute list of non-visited adjacent nodes of node i
26:     ▷ suppose AdjacentNonVisitedNodes contains L nodes
27:     compute AdjacentNonVisitedNodes
28:     ▷ Number of SDs to borrow from each adjacent node
29:     XchngNum = LoadImbalance[i] / L
30:     for each node m ∈ AdjacentNonVisitedNodes(i) do
31:         LoadImbalance[m] -= XchngNum
32:     end for
33:     LoadImbalance[i] = 0
34: end for
35: reset_all(hpx::performance_counters::busy_time)
```

A. Validation of the Implementation

We first validate the solver by considering a test setting and comparing the numerical solution with the analytical solution; we refer to Subsection III-B for the analytical solution test details where the specific form of the external heat source b is chosen so that the model has an analytical solution. We compute the error due to the numerical discretization following (7); the error depends on the timestep size and the mesh size and should decrease as the timestep size and mesh size decrease. From Figure 8 we see that the numerical error is decreasing with a decrease in mesh size and this serve as a validation of the serial and *distributed solver*.

B. Shared Memory Implementation

We now study the speedup based on the instruction-level parallelism using multiple threads. The main idea is to distribute the SDs uniformly among multiple threads

Authorized licensed use limited to: University of Texas at Austin. Downloaded on July 08,2021 at 02:18:11 UTC from IEEE Xplore. Restrictions apply.

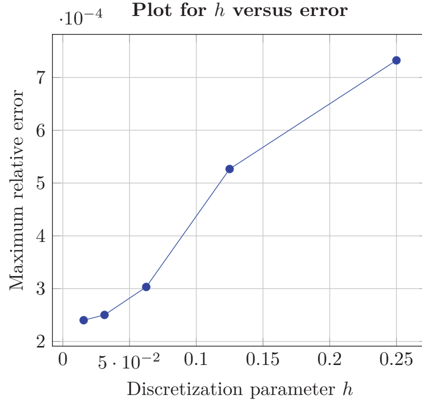

Fig. 8. Plot of the total error $e = \sum_k e^k$, see (7) for different mesh sizes $h = 1 / 2^n$, where $n = 2, 3, .., 6$. We expect the numerical error to decrease as the mesh size decreases.

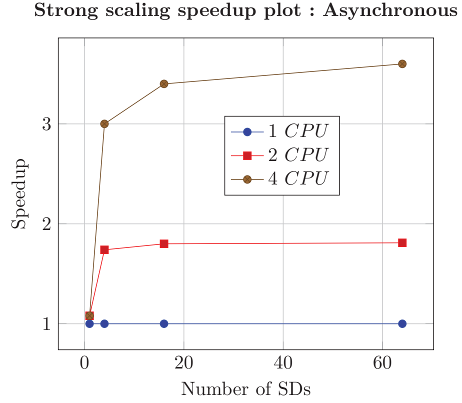

Fig. 9. Strong scaling results of the *asynchronous solver* for mesh size = 400×400 with $\epsilon = 8h$ and no. of timesteps = 20 (*i.e.* $N = 20$ in $N\Delta t = T$). The entire mesh of size 400×400 is divided into equal sized SDs. The size of the SDs is varied to keep the total mesh size constant, *e.g.* For number of SDs = 16, the partitioning is as follows: 4×4 SDs (*i.e.* 4 SDs along X and Y directions) each of size 100×100 (*i.e.* 100 DPs along X and Y directions in each SD).

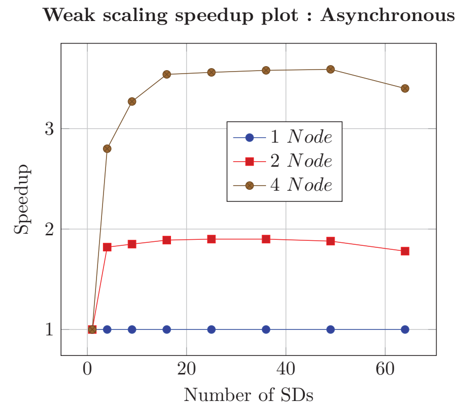

Fig. 10. Weak scaling results of the *asynchronous solver* for SD size = 50×50 (*i.e.* 50 DPs along X and Y directions), $\epsilon = 8h$ and no. of timesteps = 20 (*i.e.* $N = 20$ in $N\Delta t = T$). The number of SDs is increased along both X and Y directions, keeping the size of the SDs constant. The total mesh size is given by $50n \times 50n$, where n is the number of SDs.

located on the same computational node while sharing a common data structure. At every timestep, each thread is responsible for computing the temperature and updating the common data structure for the allocated SDs. We study the speedup for 1, 2, and 4 CPU scenarios for fixed and variable problem sizes. The single CPU execution time is the baseline for the speedup plots.

a) ***Strong scaling of asynchronous 2d nonlocal equation:*** In Figure 9, we present the scaling results for the asynchronous implementation of (5) for the fixed problem size. We consider a fixed 400×400 mesh and measure the effect of the decomposition into SDs of different sizes. We consider SDs of four sizes in four cases: 1) 1×1 *i.e.*, the entire square domain, 2) 2×2 *i.e.*, entire square domain is divided into a total of 4 partitions (2 along each of the axes), 3) 4×4, and 4) 8×8. The strong scaling plot in Figure 9 indicates a linear dependence of the execution time on the number of CPUs.

b) ***Weak scaling of asynchronous 2d nonlocal equation:*** In Figure 10, we present the scaling of the asynchronous implementation of (5) variable problem size. We study the effect of increasing the mesh size by increasing the number of SDs of fixed size 50×50 along the X and the Y axes. Eight different types of problem sizes that we considered are illustrated using the following examples: 1×1 *i.e.*, the total problem size is 50×50; 2×2 *i.e.*, the total problem size is 100×100; 3×3 *i.e.*, the total problem size is 150×150; and 8×8 *i.e.*, the total problem size is 400×400. The weak scaling plot in Figure 10 indicates a linear dependence of the execution time on the increase in problem size irrespective of the number of CPUs.

C. Distributed memory implementation

We now study the speedup based on data-level parallelism using multiple computational nodes. We distribute the SDs uniformly across multiple computational nodes. Each computational node is responsible for computing the temperature corresponding to its SDs. In this process, the computational nodes might exchange data to satisfy the nonlocal dependencies. In our experiments, we study the speedup for 1, 2, and 4 computational node scenarios for fixed and variable problem sizes. Single computational node execution time is the baseline for the speedup plots.

The distribution of SDs across 1, 2 and 4 computational nodes is as follows: 1 Node: Entire square domain is on a single node; 2 nodes: Entire square domain is divided into 2 equal sized halves. For the number of SDs = 4×4, we divide the square domain into 2 halves of equal size – 2×4 and 2×4. Each half is assigned to different computational nodes; and 4 nodes: Entire square domain is

Authorized licensed use limited to: University of Texas at Austin. Downloaded on July 08,2021 at 02:18:11 UTC from IEEE Xplore. Restrictions apply.

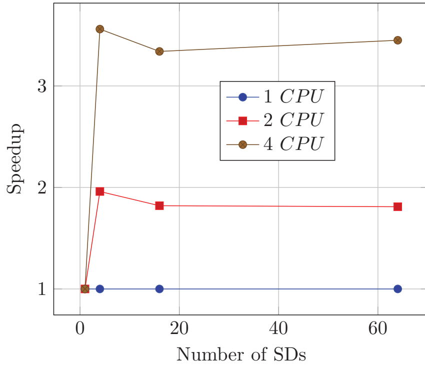

Fig. 11. Strong scaling results of the *distributed solver* for mesh size = 400×400 with $\epsilon = 8h$ and no. of timesteps = 20 (*i.e.* $N = 20$ in $N\Delta t = T$). The entire mesh of size 400×400 is divided into equal sized SDs. The size of the SDs is varied to keep the total mesh size constant, *e.g.* For number of SDs = 16, the partitioning is as follows: 4×4 SDs (*i.e.* 4 SDs along X and Y directions) each of size 100×100 (*i.e.* 100 DPs along X and Y directions in each SD).

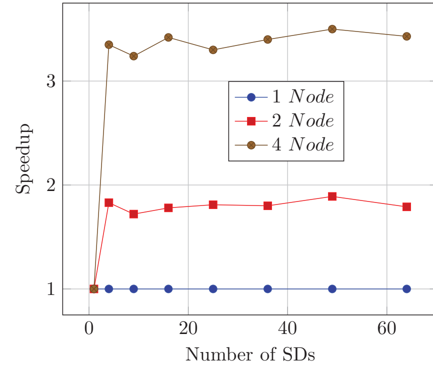

Fig. 12. Weak scaling results of the *distributed solver* for SD size = 50×50 (*i.e.* 50 DPs along X and Y directions), $\epsilon = 8h$ and no. of timesteps = 20 (*i.e.* $N = 20$ in $N\Delta t = T$). The number of SDs is increased along both X and Y directions, keeping the size of the SDs constant. The total mesh size is given by $50n\times50n$, where n is the number of SDs. The distribution of SDs across the computational nodes is done using METIS library.

divided into 4 equal sized squares, each assigned to distinct computational nodes.

*a) **Strong scaling of the distributed 2d nonlocal equation:*** In Figure 11, we present the scaling of the distributed implementation of (5) for a fixed problem size. We study the effect of decomposition of a mesh with fixed size 400×400 into a different number of SDs to demonstrate the speedup. We consider SDs of four sizes in four cases: 1) 1×1 *i.e.*, the entire square domain, 2) 2×2 *i.e.*, entire square domain is divided into a total of 4 partitions (2 along each of the axes), 3) 4×4, and 4) 8×8. The strong scaling plot in Figure 11 indicates a linear dependence of the speedup on the number of computational nodes.

*b) **Weak scaling of the distributed 2d nonlocal equation:*** In Figure 12, we present the scaling of the distributed implementation of (5) with variable problem size. We study the effect of increasing the mesh size by increasing the number of SDs where each SD is of fixed size 50×50. Eight different types of problem sizes that we considered are illustrated using the following examples: 1×1 *i.e.* total problem size is 50×50; 2×2 *i.e.* total problem size is 100×100; 3×3; and 8×8. The weak scaling plot in Figure 12 indicates a linear dependence of the speedup with an increase in the number of computational nodes, irrespective of the problem size.

*c) **Distributed scaling using METIS for mesh partitioning:*** We now study the scaling of the *distributed solver* where we keep the size of the problem fixed (*i.e* the mesh size is fixed) and increase the number of computational nodes; see the plot of speedup with different number of nodes in Figure 13. We consider a fixed 800×800 mesh (800 grid points in X and Y directions); from this mesh we generate 16×16 square SDs with each SD consisting

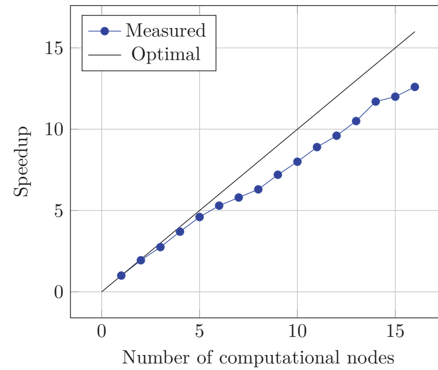

Fig. 13. Distributed scaling results of the *distributed solver* for mesh size = 800×800, SD size = 50×50 (*i.e.* 50 DPs along X and Y directions), $\epsilon = 8h$ and no. of timesteps = 20 (*i.e.* $N = 20$ in $N\Delta t = T$). Total no. of SDs is 16×16 (*i.e.* 16 SDs along X and Y directions). The distribution of SDs across the varying no. of computational nodes is done using METIS library.

of 50×50 grid points. We then use METIS library for distribution of SDs across multiple computational nodes. Advantages of distributing the SDs instead of the original fine mesh are as follows: the partitioning using METIS is very fast, since the number of SDs is much smaller than the number of grid points in the original mesh; I/O time is reduced since we only need to read and write the SD allocation information; and, lastly, SDs owned by a node can further be distributed across multiple threads within that node for parallel computation.

The distributed scaling plot in Figure 13 indicates a linear relationship between the speedup and the number of computational nodes. As the number of computational

Authorized licensed use limited to: University of Texas at Austin. Downloaded on July 08,2021 at 02:18:11 UTC from IEEE Xplore. Restrictions apply.

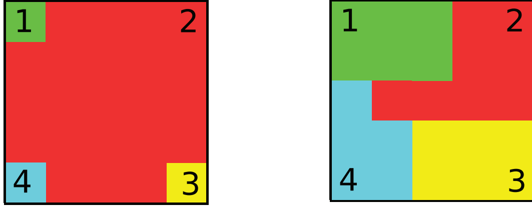

Fig. 14. Redistribution of 5×5 SDs across 4 computational nodes. Four colors denote the SDs belonging to the four distinct nodes indicated by the numbers 1, 2, 3, 4. Starting from a very imbalanced load distribution(left), within 3 iterations, the load balancing algorithm is able to achieve a very balanced distribution across 4 computational nodes(right).

nodes increases, the number of boundary SDs requiring data exchange increases. This increase in the data exchange leads to a slight deviation from the straight line.

*d) **Validation of the load balancing algorithm:*** To verify the load balancing algorithm in Algorithm 1, we consider 4 symmetric nodes and assign a highly imbalanced load to each of them in the beginning. We demonstrate in Figure 14 that within 3 iterations, the load balancing algorithm is able to redistribute the SDs among various nodes with nearly balanced load distribution.

IX. Conclusion and Future Work

We proposed the ideas of – (*i*) coarsening the mesh for higher granularity, (*ii*) efficient mesh partitioning using METIS, (*iii*) hiding data exchange time using asynchronous computation and demonstrated good weak and strong scaling for the distributed implementation. We presented a novel load balancing algorithm by using HPX to schedule the tasks asynchronously (using `hpx::future` and `hpx::async`) and local control objects for synchronization. The major contribution is the novel load balancing algorithm that utilizes HPX's performance counters to address the specific challenges of nonlocal models. We used SDs (sub-domains) as a unit of exchange to ensure simplicity in modelling the data exchange and to preserve the data locality. We proposed to preserve the contiguous locality obtained using the METIS mesh partitioning by borrowing the SDs uniformly in all the spatial directions; the redistributed load after the load-balancing step minimizes the data exchange time. In one example, we could show that the proposed algorithm balanced a largely unbalanced domain within three iterations. For future work, we intend to investigate the addition of specific performance counters and networking counters. Larger node counts will be investigated which was not possible with our current allocation for this project. From the geometry perspective, a more complex non-square domains and three dimensions will be investigated. From the model perspective, a more complex models, *e.g.* nonlocal mechanics [1], [2] will be investigated.

Acknowledgements

We are grateful for the support of the Google Summer of Code program which funded P. Gadikar's summer internship.

Supplementary materials

The source code to reproduce the results are available on GitHub[1].

References

[1] P. Diehl *et al.*, "A review of benchmark experiments for the validation of peridynamics models," *Journal of Peridynamics and Nonlocal Modeling*, vol. 1, no. 1, pp. 14–35, 2019.
[2] P. K. Jha *et al.*, "Kinetic relations and local energy balance for lefm from a nonlocal peridynamic model," *International Journal of Fracture*, vol. 226, no. 1, pp. 81–95, 2020.
[3] N. Burch *et al.*, "Classical, nonlocal, and fractional diffusion equations on bounded domains," *International Journal for Multiscale Computational Engineering*, vol. 9, no. 6, 2011.
[4] N. J. Armstrong *et al.*, "A continuum approach to modelling cell–cell adhesion," *Journal of theoretical biology*, vol. 243, no. 1, pp. 98–113, 2006.
[5] C. Engwer *et al.*, "On a structured multiscale model for acid-mediated tumor invasion: the effects of adhesion and proliferation," *MMMA*, vol. 27, no. 07, pp. 1355–1390, 2017.
[6] P. K. Jha *et al.*, "Peridynamics-based discrete element method (peridem) model of granular systems involving breakage of arbitrarily shaped particles," *arXiv preprint arXiv:2010.07218*, 2020.
[7] G. Karypis *et al.*, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.
[8] H. Kaiser *et al.*, "Hpx - the c++ standard library for parallelism and concurrency," *Journal of Open Source Software*, vol. 5, no. 53, p. 2352, 2020.
[9] J. D. d. S. Germain *et al.*, "Uintah: A massively parallel problem solving environment," in *Proceedings the Ninth International Symposium on High-Performance Distributed Computing*. IEEE, 2000, pp. 33–41.
[10] B. L. Chamberlain *et al.*, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.
[11] L. V. Kale *et al.*, "Charm++ a portable concurrent object oriented system based on c++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 1993, pp. 91–108.
[12] H. C. Edwards *et al.*, "Kokkos: Enabling manycore performance portability through polymorphic memory access patterns," *Journal of Parallel and Distributed Computing*, vol. 74, no. 12, pp. 3202–3216, 2014.
[13] M. Bauer *et al.*, "Legion: Expressing locality and independence with logical regions," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.
[14] G. Bosilca *et al.*, "Parsec: Exploiting heterogeneity to enhance scalability," *Computing in Science & Engineering*, vol. 15, no. 6, pp. 36–45, 2013.
[15] P. Thoman *et al.*, "A taxonomy of task-based parallel programming technologies for high-performance computing," *The Journal of Supercomputing*, vol. 74, no. 4, pp. 1422–1434, 2018.
[16] E. Jeannot *et al.*, "Communication and topology-aware load balancing in charm++ with treematch," in *2013 IEEE CLUSTER*, 2013, pp. 1–8.
[17] J. Lifflander *et al.*, "Optimizing data locality for fork/join programs using constrained work stealing," in *SC14*, 2014, pp. 857–868.
[18] P. A. Grubel, *Dynamic Adaptation in HPX: A Task-based Parallel Runtime System*. New Mexico State University, 2016.
[19] P. Amini *et al.*, "Assessing the performance impact of using an active global address space in hpx: A case for agas," in *2019 IPDRM*. IEEE, 2019, pp. 26–33.

[1] https://github.com/nonlocalmodels/nonlocalheatequation

Authorized licensed use limited to: University of Texas at Austin. Downloaded on July 08,2021 at 02:18:11 UTC from IEEE Xplore. Restrictions apply.