LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-CONF-771313

Improving the Visualization Process for Large SPH Particle Datasets Using Dynamic Load Balancing with Adaptive Bin Refinement

K. S. Griffin, B. Hamann

April 5, 2019

VIS 2019
Vancouver, Canada
October 20, 2019 through October 25, 2019

Disclaimer

Improving the Visualization Process for Large SPH Particle Datasets Using Dynamic Load Balancing with Adaptive Bin Refinement

Kevin Griffin, *Member, IEEE* and Bernd Hamann, *Member, IEEE*

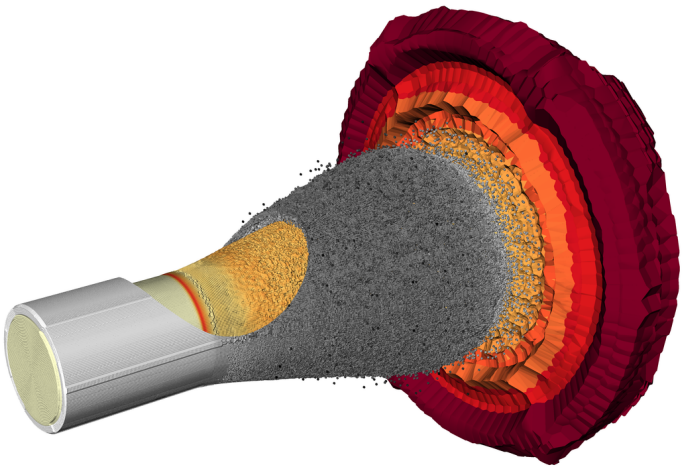


Fig. 1: Visualization of a cylinder explosion simulation dataset using first-order RK-enhanced SPH interpolation and dynamic load-balancing.

Abstract—Smoothed particle hydrodynamics (SPH) has become a popular hydrodynamics simulation method. However, several challenges arise when attempting to visualize the results of these simulations. In this paper, we focus on the problem of load-balancing SPH particle datasets across processors of a high-performance computing (HPC) platform. We show that an imbalanced distribution of particles can lead to poor runtime performance as well as memory scalability issues. We describe a dynamic load-balancing and adaptive bin refinement solution that guarantees that each processor will have approximately the same number of particles to process for each time step, resulting in improved runtime performance and memory scalability.

Index Terms—Load-balance, SPH, HPC

1 Introduction

Since most real-world particle-based simulations produce large, non-uniform datasets, being able to distribute them across processors is very important for data and memory scalability, and improved runtime performance. However, as shown in our previous work [8], doing a simple spatial decomposition of a dataset can lead to an uneven distribution of particles resulting in poor runtime performance and memory scaling. Figure 2 shows a simple example where a spatial partitioning of a non-uniform dataset leads to an imbalanced distribution of particles. Furthermore, in this simple example, if each processor only has memory resources for at most three particles, then it will not be possible to process the dataset since two batch nodes would have exhausted their memory resources and crashed. If we had additional resources, we could decompose the dataset even further, which could potentially result in some processors not having any work at all, thus wasting valuable computing resources. Therefore, load-balancing is very important for efficient post-processing of large parallel simulation datasets.

This paper makes the following contributions:

- We introduce our RK-enhanced SPH interpolation scheme for visualizing SPH simulation datasets.

- *Kevin Griffin is with the Applications, Simulation, and Quality Division, Lawrence Livermore National Laboratory, Livermore, CA 94550, USA. E-mail: griffin28@llnl.gov.*
- *Kevin Griffin and Bernd Hamann are with University of California, Davis. E-mail: kevgriffin, bhamann@ucdavis.edu.*

Manuscript received xx xxx. 201x; accepted xx xxx. 201x. Date of Publication xx xxx. 201x; date of current version xx xxx. 201x. For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org. Digital Object Identifier: xx.xxxx/TVCG.201x.xxxxxxx

- We present a novel approach for load-balancing particle datasets that's not constrained by ghost particle layers.
- We introduce a spatial partitioning algorithm that uses adaptive bin refinement for creating the underlying parallel k-d tree.
- We evaluate our method with a large SPH particle dataset using 36, 72 and 144 processors.
- We compare our method with other well known load-balancing algorithms.

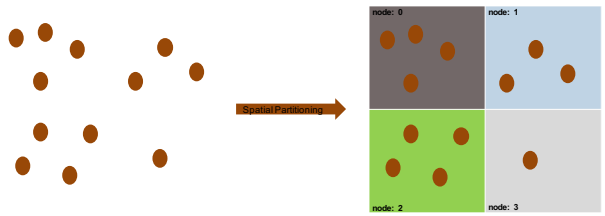


Fig. 2: Load imbalance as a result of doing a simple spatial partitioning that evenly divides the spatial domain.

2 Background

2.1 Smoothed Particle Hydrodynamics

Smoothed Particle Hydrodynamics (SPH) is a Lagrangian alternative to mesh-based schemes for simulating fluid flows. SPH was initially used in astrophysics, but it can also be applied to a wide variety of physical applications. Gingold, Monaghan, and Lucy [7, 12, 15] are given credit for the derivation of SPH. An example image using this method in VisIt, an open source, parallel visualization, and graphical analysis tool, is shown in Figure 3. Other examples include using SPH to model material damage and failure [20], fluid mixing, strong shock, and adiabatic phenomena [6].

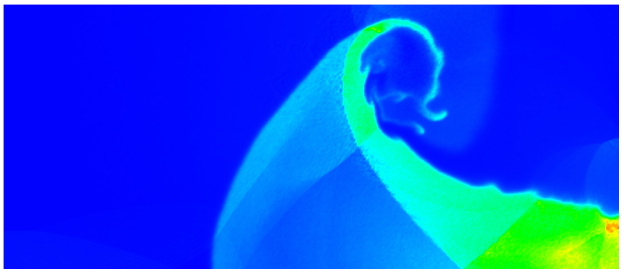


Fig. 3: Visualization of a triple-point shock test simulation dataset in which a high density and high pressure material drives a shock parallel to the density discontinuities of the two regions shown in light-blue and green.

SPH makes possible the recovery of density distributions, body forces, or any other field (scalar or vector) carried by a particle. It does this by convolving the interpolation of the information carried by each particle with a smoothing kernel. Any field value used in the SPH algorithm can be approximated by

$$\langle A(r)\rangle = \int_D A(r_j)W(h_j,|r-r_j|)\,dr_j, \tag{1}$$

where $A(r)$ is the approximated field value and $W(h_j,|r-r_j|)$ is the kernel satisfying the following conditions:

$$\lim_{h\to 0}\int_D W(h_j,u)du = 1, \tag{2}$$

which requires that our basis functions define a partition of unity, and

$$W(h_j,u) \to \delta(u)\ as\ h \to 0, \tag{3}$$

which requires the kernel to approach the Dirac delta function, $\delta(u)$, as h approaches zero. The Dirac delta function is defined by the properties

$$\delta(u) = \begin{cases} 0 & : \quad u \neq 0, \\ \infty & : \quad u = 0, \end{cases} \tag{4}$$

and

$$\int_{-\infty}^{\infty}\delta(u)du = 1. \tag{5}$$

The discrete version of (1) can be written as

$$\langle A(r)\rangle = \sum_j V_j A_j W_j(h_j,|r-r_j|), \tag{6}$$

where A is the quantity to be interpolated at position r, V_j is the volume of particle j, usually given by m_j/ρ_j, W_j is a basis function, h_j is the smoothing length, and r_j is a particle at position j. While the choice of interpolation kernel can be arbitrary, there are some desired characteristics that W_j should have to make the results more precise. These characteristics include the kernel being Gaussian-*like* and the kernel having compact support (results are zero outside of a finite smoothing length). While there are infinitely many kernels one could use, the three most common for SPH are the exponential, the cubic spline, and the quartic spline [2].

The consistency of the continuous form of the kernel approximation (1) has been shown to be zeroth- and first- order consistent by Belytshko *et al.* [2]. However, Belytshko *et al.* also showed that the discrete form (6), while first-order consistent for uniform datasets, is not first-order consistent for randomly displaced particle datasets. Furthermore, the situation tends to be worse on the boundaries where the discrete form is not first-order consistent for uniform datasets. The error associated with SPH was originally derived from associating (1) with a Monte Carlo estimate, making the error proportional to $\frac{1}{\sqrt{N}}$. However, Gingold and Monaghan [7] determined that, in practice, this error is much less than $\frac{1}{\sqrt{N}}$ due to the inability of disordered particles to cause large fluctuations [14].

2.2 RK-enhanced SPH Interpolation

Liu *et al.* [11] introduced the Reproducing Kernel Particle Method (RKPM), taking advantage of mesh-free, Lagrangian methods while at the same time improving accuracy with reproducing kernels. RKPM is based on the theory of wavelets and uses three main ideas from wavelet analysis: the dilation and translation of the window function, integral window transforms, and the continuous and discrete reproducing kernel approximations [11]. The dilation and translation is defined by

$$\Phi(x) = E(a)\Phi\left(\frac{x-b}{a}\right) \quad : \quad a > 0, \tag{7}$$

where $\Phi(x)$ is the window function located at $x = b$ whose support, $B(x)$, is scaled by a. $E(a)$ is a normalizing function such that

$$\int_{R^x}\Phi(x)dR^x = 1 \tag{8}$$

when the support $B(x)$ is within the spatial region of interest, R^x. Furthermore, the window function has compact support within $B(x)$. This means that the window function has the following properties:

$$\Phi(x) \neq 0\ in\ B(x), \tag{9}$$
$$\Phi(x) = 0\ outside\ B(x). \tag{10}$$

The integral window transform is defined as

$$\langle u,\Phi_{ab}\rangle = \int_{R^x} E(a)\Phi\left(\frac{x-b}{a}\right)u(x)dR^x, \tag{11}$$

where $u(x)$ is a real function and Φ_{ab} is a real window function. Discretizing (11) by n points leads to

$$\langle u(x)\rangle \approx \sum_{j=1}^{n} E(a)\Phi\left(\frac{x_j - x}{a}\right) u(x_j)\Delta V_j. \tag{12}$$

Three moments are defined for the window function:

$$m_0(x) = \int_B \Phi(x_j) dR^x, \tag{13}$$

$$m_\alpha(x) = \int_B x_j^\alpha \Phi(x_j) dR^x, \tag{14}$$

$$m_{\alpha\beta}(x) = \int_B x_j^\alpha x_j^\beta \Phi(x_j) dR^x, \tag{15}$$

where α and β range between one and the number of spatial dimensions. The zeroth moment, $m_0(x)$, is less than one when a particle, x_j, is close to the boundary. The first moment, $m_\alpha(x)$, is equal to zero in the interior of a uniform particle dataset and not equal to zero when the particle is close to the boundary. The second moment, $m_{\alpha\beta}$, represents the distribution of the particle mass, when $\alpha \neq \beta$, or the cross moment when $\alpha = \beta$. The moments are used to derive the correction function, $C(a,x,x_j)$. Multiplying the dilation and translation by $C(a,x,x_j)$ reproduces a kernel that provides boundary corrections, produces smoother approximations of the overall solution and its derivatives, and leads to higher accuracy [11]. The reproducing kernel is defined by equations in (16), and (17) provides a highly accurate approximation of $u(x)$ by multiplying $u(x_j)$ by the reproducing kernel and integrating over the region of interest:

$$k(a,x,x_j) = C(a,x,x_j)E(a)\Phi\left(\frac{x_j - x}{a}\right), \tag{16}$$

$$u(x) = \int_{R^x} k(a,x,x_j)u(x_j)R^x. \tag{17}$$

The discretized form of (17) is defined as

$$u(x) = \sum_{j=1}^{n} C(a,x,x_j)E(a)\Phi\left(\frac{x_j - x}{a}\right) u(x_j)\Delta V_j,, \tag{18}$$

$$u(x) \approx \sum_{j=1}^{n} k(a,x,x_j)u(x_j)\Delta V_j \tag{19}$$

where n is the number of particles. Comparing the RKPM formulation to SPH, we see that the dilation and translation is quite similar to the SPH smoothing kernel and has most of the desired properties. Ideally, it should be possible to substitute the dilation and translation with an SPH kernel, multiply it with a similar correction function, and produce approximations that are more accurate than standard SPH. This is in fact, what we do to visualize our SPH simulation datasets.

3 Visualizing SPH Simulation Datasets

Figure 4 illustrates the importance of the decision to use an RK-enhanced interpolation scheme to visualize SPH simulation datasets. Both images are from a Kelvin-Helmholtz simulation dataset, just the one in **(a)** doesn't use any RK corrections while the one in **(b)** does. The stippling seen in the uncorrected version is from particle chaining that is present in the rollup regions. The RK corrections recover the correct values of the density in the rollup regions producing a more accurate visualization.

4 Related Work

Tree methods are popular methods used to partition and index the spatial domain of a simulation based on a tree data structure. There are two candidate methods that we consider. One is based on using a binary tree structure to partition the domain into subdomains, compressing and expanding these domains into balanced groups. The other method is based on k-d trees. K-d trees are multi-dimensional binary search trees, where k is the dimension of the search space [3]. Figure 5 shows the resulting k-d tree for a small dataset with six points and $k = 2$.

D. Zhang *et al.* [22] introduced a method (Algorithm 1) based on partitioning and indexing the spatial region with a binary tree structure. The n levels of the binary tree represent $2^n - 1$ line partitions of the simulation domain. Indices at the top of the tree are smaller than those on the bottom, and the indices of the nodes on the left are smaller than those on the right. Workloads are calculated for each subregion (subtree) and the degree of imbalance, I_l, is quantified based on a formula derived in [22]. If I_l is greater than the *ideal* amount of load imbalance, then each line, represented by the $2^n - 1$ nodes in the binary tree, will be adjusted as follows: If the node represents a vertical line, then it will be moved horizontally; otherwise it will be moved vertically [22]. The offset for each line is defined in [22]. We view this work as a good representative of tree-based methods for load-balancing particle-based datasets and use it to compare to our method.

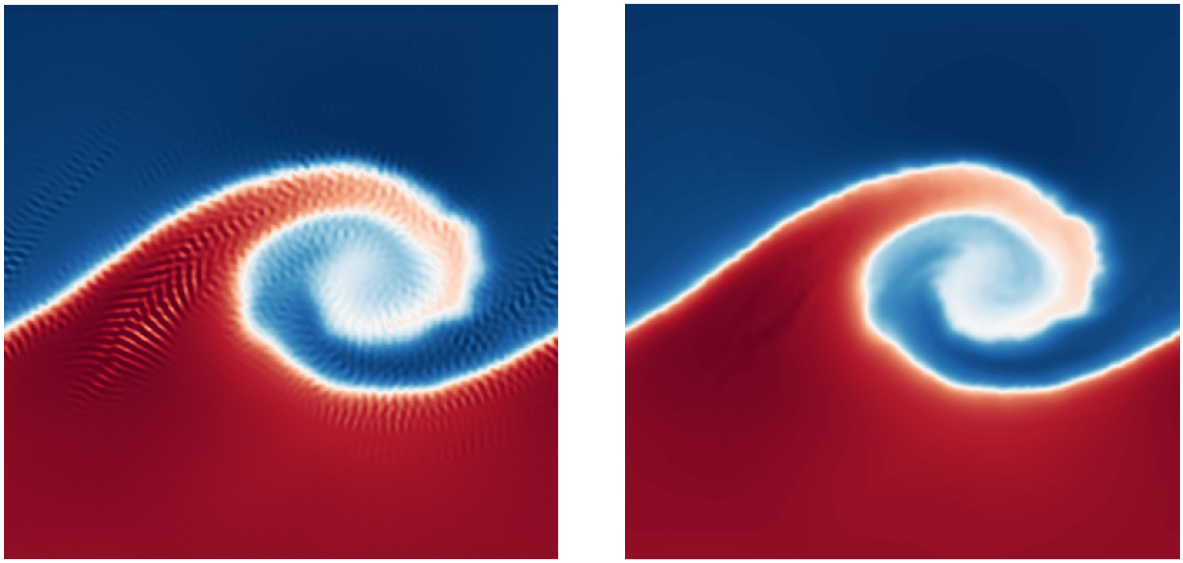
Algorithm 1 D. Zhang *et al.* [22]

```
1: T ← createBinaryTree()  ▷ Create binary tree for particle dataset
2: n = depth(T)
3: W[0...2^n − 1]
4: for i ← 0, 2^n − 1 do
5:     W_i ← calculateWorkLoad(i)      ▷ Calculate work loads of all sub-domains
6: end for
7: I_l ← calculateLoadImbalance(W)  ▷ Calculate imbalance degree defined in [22]
8: if I_l > I_ideal then
9:     for i ← 0, 2^n − 1 do
10:        Adjust line partitions by offset defined in [22]
11:        Goto step 4
12:    end for
13: else
14:    Done
15: end if
```

The work by J. Zhang *et al.* [23] performs load-balancing using k-d trees that are constrained in the overlapping regions of ghost layers. The data is partitioned like most tree methods by alternating the splitting of the domain in each dimension. However, the k-d tree partitioning does not guarantee a balanced distribution of particles since the splitting planes of the k-d trees are limited to the overlapped regions of ghost layers. A histogram approach is used to find the median when creating the k-d tree and a single parameter is manually configured to determine the frequency of particle redistribution.

Morozov and Peterka [19] use a distributed k-d tree to evenly distribute particle data across batch nodes. In their approach, the k-d tree algorithm is executed until the dataset is partitioned into P irregularly sized blocks, where P is the number of processors, containing an even subset of particles contained in the dataset. Initially, the root node of the k-d tree contains all particles. At each level of the tree, an approximate median is chosen using histograms that are calculated by each process and reduced to a root process that determines the median and broadcasts its value to the other processes. Once the median is known, processes exchange data with their partners. In the first iteration, a process with a process id having a least-significant bit of zero sends its points above the median to its partner, and its partner sends the points below the median back to the former process [19]. In subsequent iterations, communication only happens within each newly created group, using the associated least-significant bits. This algorithm cycles through each dimension until the number of subtrees in the k-d tree is equal to the number of batch nodes, see Algorithm 2.

Marzouk and Ghoniem [13] introduce an algorithm based on a weighted k-means clustering for partitioning and dynamic load-balancing of N-body interactions. K-means is a popular clustering algorithm used to divide a dataset into k clusters such that some metric, like the within-cluster sum of squares (WCSS), is minimized for each cluster [10]. The algorithm is designed to run in parallel on



(a) Traditional SPH

(b) RK-Enhanced SPH

Fig. 4: Visualization of a dataset produced from a Kelvin-Helmholtz simulation. Both images are from the exact same dataset, just one uses the RK corrections **(b)** and the other does not **(a)**. The stippling seen in **(a)** is from particle chaining that is present in the rollup regions, the RK corrections recover the correct values of the density there.

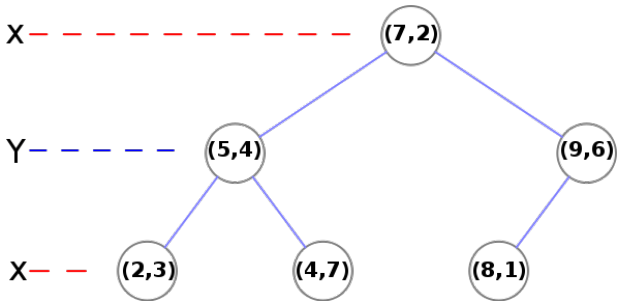


Image Source: `https://en.wikipedia.org/wiki/K-d_tree`

Fig. 5: Resulting k-d tree for a list of six 2D points. The X-dimension is chosen for the initial split. Alternating, one performs splitting with the Y-dimension at each subsequent level, until each child node contains only one point. The median for each dimension is chosen as the split value.

Algorithm 2 Morozoz and Peterka [19]

for all processors **do** ▷ in parallel
  $points \leftarrow$ input points for each process
  $dim \leftarrow 0$
  $group \leftarrow [0 \ldots p-1]$ ▷ p = number of processors
  $pid \leftarrow$ process id
  for round $i \in [0, log_2 p)$ **do**
    $m \leftarrow$ select-median($points$,$group$,dim)
    $i-bit \leftarrow 1 << i$
    $partner \leftarrow gid$ XOR $i-bit$
    $points \leftarrow$ swap-points($partner$,$points$,m)
    $group \leftarrow \{x \in group \mid x$ AND $i-bit = gid$ AND $i-bit\}$
    exchange medians with neighbors
    $dim \leftarrow dim+1 \bmod 3$ ▷ 3D dataset
  end for
end for

distributed memory clusters. In their implementation, instead of minimizing WCSS Euclidean distances, they minimize the within-cluster vorticity magnitudes of each particle. A limitation of this work is that they need to keep copies of all particle positions and weights on each processor. This puts a memory constraint on datasets and eliminates the ability to arbitrarily scale up to larger datasets. Most work in this area has been focused on accelerating the overall k-means algorithm. For example, Fatta and Pettinger [5] proposed a k-d tree based, parallel k-means algorithm for distributed memory systems. The k-d tree used in this work is to accelerate the nearest neighbor searches that are performed for each iteration of the k-means algorithm.

5 Particle Load-balancing With Adaptive Bin Refinement

As stated in the Introduction, a regular partitioning of particle-based simulation datasets can lead to load imbalance, especially during later timesteps when particles tend to congregate in a small section of a spatial region. For load-balancing SPH particle datasets, we use a method based on parallel k-d trees. We have extended this method by eliminating constraints on ghost particles and using an adaptive, parallel histogram approach for finding the median when creating the parallel k-d tree. Sections 5.1 and 5.2 describe the method we use to create the spatial partitioning and the underlying parallel k-d tree, Section 5.3 describes our particle (including ghost particles) exchange implementation, and Section 5.4 discusses our current reload-balancing strategy.

Algorithm 3 SPH Particle Load-balancing

$n \leftarrow$ number of processors
$bounds \leftarrow$ unify-min-max($bounds$,6)
$tree \leftarrow$ kdtree(3, $bounds$, X-AXIS)
for $i \in [0, log_2 n)$ **do**
  $tree \leftarrow$ split($tree$)
end for
relocate-particles($tree$)
$dataset \leftarrow$ sph-resample($tree$)
parallel-render($dataset$)

5.1 Spatial Partitioning

One goal of our method is to decompose a particle dataset into irregularly sized blocks that have approximately the same number of particles assigned to each block, see Figure 6. Algorithm 3 describes our approach. We start with the entire bounds of the dataset as input to create a k-d tree with n leaves, where n is the number of processors. Since each processor is operating with only the information that is stored locally, the bounds information needs to be globally reduced as done in step 2. We initialize the k-d tree with the number of dimensions, the total bounds, and the starting axis to use for splitting. We alternate through each dimension, splitting the bounds into partitions with approximately equal numbers of particles. For large SPH datasets, finding the median to split with respect to an axis is computationally expensive. To offset this cost, we devised a parallel histogram approach, using adaptive bin refinement (ABR), to find the split point in a more efficient way, see Section 5.2. By design, the splitting results in a balanced, parallel k-d tree, where the number of leaves equals the number of processors. Each leaf of the local k-d tree has global metadata that describes the irregularly sized block that each processor is constrained to. Only the particles that are local to this processor are stored in the k-d tree. This is necessary to achieve memory and data scalability. Particles are re-located to the appropriate processor, see Section 5.4, and re-sampled using our first-order, RK-enhanced SPH interpolation scheme. Finally, the resulting dataset is rendered in parallel using a high-performance, sort-last parallel rendering library, called IceT [16–18]. IceT is a popular large-scale compositing solution used in visualization tools like Paraview [1] and VisIt [4].

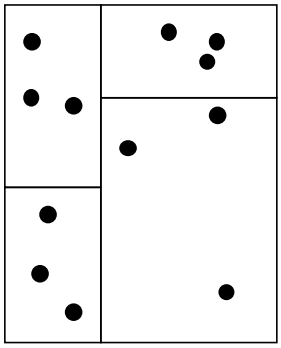


Fig. 6: Spatial decomposition using irregularly sized blocks to achieve load-balancing.

5.2 Adaptive Bin Refinement

Adaptive bin refinement (ABR) is an efficient way to find the split point when creating the k-d tree. We first compute the local histogram for each processor, see line 7 of Algorithm 4. The local histogram is created and initialized with a default number of bins, which, in our case, is six. For each bin, we compute the min and max values for the current splitting dimension. Having determined the dimensions of our local histogram, we perform a distributed integer sum reduction of the number of particles in each region across all processors. We attempt to find the split based on the number of particles in each bin of the histogram, i.e., we use a histogram of the distribution of point coordinates along the current dimension to determine where to split the children. We recursively increase the number of bins until the difference between the distribution of points across the optimal split is less than one percent. This process is repeated, while alternating axes, until the number of leaves is equal to the number of processors. The result is a balanced k-d tree, where each leaf node contains approximately the same number of particles.

5.3 Particle Exchange with Ghost Particles

With the k-d tree structure in place, we have enough information to distribute the particles to the right processors. Each leaf node of the k-d tree contains global metadata for determining where to re-distribute

Algorithm 4 Adaptive Bin Refinement

```
1:  npivots ← 6
2:  maxtries ← 3
3:  attempts ← 0
4:  while True do
5:      pivots ← create-bins(npivots)
6:      tp ← 0                                          ▷ total particles
7:      for i ∈ [0, npivots + 1) do
8:          tp ← tp + nparts_i
9:      end for
10:     particles ← 0
11:     pivotratio ← [0 ... npivots)
12:     pivotidx ← 0
13:     for i ∈ [0, npivots) do
14:         particles ← particles + nparts_i
15:         pivotratio_i ← particles/tp
16:     end for
17:     ratio ← splitcnt/ length(p)          ▷ p = number of processors
18:     error ← abs(ratio − pivotratio_0)
19:     pivotidx ← 0
20:     for i ∈ [1, npivots) do
21:         diff ← abs(ratio − pivotratio_i)
22:         if diff < error then
23:             error ← diff
24:             pivotidx ← i
25:         end if
26:     end for
27:     if (error < 0.01) OR (attempts > maxtries) then
28:         return pivots[pivotidx]
29:     else
30:         npivots ← npivots ∗ 2
31:         attempts ← attempts + 1
32:     end if
33: end while
```

each particle, but only the local particles have been inserted into the tree at this point. It is necessary to distribute the rest of the particles, and their associated variables, to the appropriate processors as described in Algorithm 5. For each particle stored locally, the receiving processor list is derived from the metadata stored in each node.

Since our dataset is being distributed across processors, this introduces problems along each domain's edge/boundary that would not occur if the dataset was considered as being defined over just one large domain. See Figure 7 as an example. If we want to apply our SPH interpolation scheme using edge particles **A** and **B** in domain 1, we will have to use all particles within the smoothing length of **A** and **B**. This requires us to know the values of particles **C** and **D** in domains 2 and 3, respectively. Without these values, the interpolated value for particles **A** and **B** will not be correct. However, if we can maintain a redundant layer of particles around the exterior of each domain, we will be able to perform the proper calculations locally and avoid communication overhead whenever executing the SPH interpolation scheme for a particle near the boundary of a domain. We call these redundant particles *Ghost Particles*. A ghost particle duplicates another particle located in an adjacent domain and within the influence range of the current domain's edge particle(s). Ghost particles are included in the data that is sent to each processor to be used in calculations and reduce communication cost.

Once the list of sending processors is established, we retrieve the *Ghost Particles* for the current point when the influence region lies outside of the bounds of the local processor. The particles, to include *Ghost Particles*, are added to the temporary buffer of particles for each processor; the corresponding labels, which identify a particle as ghost or not, are updated accordingly. The temporary buffers are serialized and a bulk send/receive, using MPI [9], is performed. Once each processor has received its updated particle set and variable data, we create a ghost node array that contains the ghost particles needed for

this set of particles.

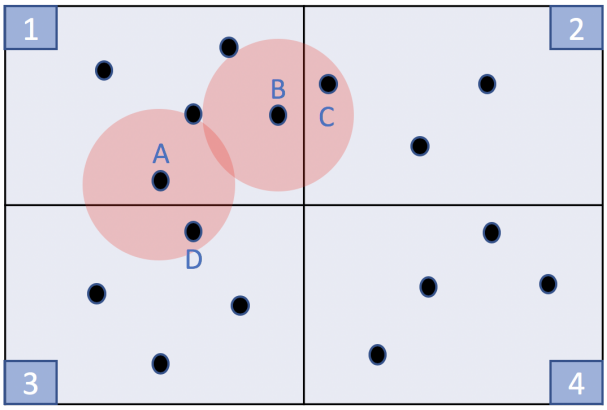


Fig. 7: Domain decomposition example showing four domains labeled 1 to 4. Red circles indicate the smoothing length or influence region of particles **A** and **B**. Particles **C** and **D** fall within the regions of influence of **A** and **B**.

Algorithm 5 Particle Exchange with Ghost Particles

```
sndparticles ← [0...p−1]                          ▷ p = total processors
sndlabels ← [0...p−1]    ▷ tags each particle as regular or ghost
npoints ← get this processor's total point count
for i ∈ [0,npoints) do
    point ← get-point(i)
    proclist ← get-processor-list(point)
    proclistsize ← length(proclist)
    for i ∈ [0,proclistsize) do
        ghostpoints ← get-ghost-particles(point,proclist_i)
        add point and ghostpoints to sndparticles_i
        update labels for sndlabels_i
    end for
end for
send sndparticles data from all to all processors
send sndlabels data from all to all processors
```

5.4 Load-balancing Across Multiple Timesteps

For time-varying datasets, there is no need to perform load-balancing after every timestep since particle movement tends to be rather limited between two timesteps, or a small number of timesteps. It is a challenge to determine a near-optimal rate, α, for performing load-balancing for each dataset. Performing load-balancing too often can lead to unnecessary computational overhead while not balancing enough can lead to a particle distribution that becomes too imbalanced. We experimented with different values for α for each dataset and selected the best one among those considered.

6 Results

6.1 Environment

Lawrence Livermore National Laboratory's RZTopaz Linux HPC Cluster was used for our evaluation. RZTopaz has 748 Intel Xeon E5-2695 nodes with 36 cores/node (2.1 GHz) and 128 GB of memory/node. The nodes are connected with an Intel Omni-Path 100 Gb/s interconnect.

6.2 Kelvin-Helmholtz Simulation Dataset

The Kelvin-Helmholtz instability, shown in Figure 4b, is a hydrodynamic instability that occurs at the interface of two fluids of different velocities and densities [21]. Our two-dimensional dataset is the result of a simulation of this instability, consisting of over 65,000 particles. We used this dataset to understand the effects that the parameters *error* and *maxtries*, used to find the split point for the AGR algorithm, have on overall runtime and load-balancing. An exhaustive grid search was performed, where the two parameters were assigned the following values; $error \in [0.01, 0.1, 1.0, 2.0, 5.0, 6.0, 10.0]$ and $maxtries \in [3, 6, 12, 24, 48]$. Table 1 shows the results of our grid search. The dataset was load-balanced across four processors, $p_1 \dots p_4$, and each column shows the number of particles received by each processor. Columns 5 and 6 display the values used for the *error* and *maxtries* parameters during each run. The median absolute error (*MedAE*) percent is calculated in column 7, where $MedAE(y, \hat{y}) = median(|y_1 - \hat{y}|, \dots, |y_n - \hat{y}_n|)$, y_i is the value in the column associated with p_i, and $\hat{y} = 16384$. The MedAE percent is obtained by dividing *MedAE* by the total number of particles. Total runtime is shown in column 8. A row in Table 1 is created for each combination of *error* and *maxtries*, producing the best runtime and *MedAE* combination. Unless otherwise specified, these are the values for the *error* and *maxtries* parameters considered in all subsequent evaluations.

Table 1: AGR Parameter GridSearch

p1	p2	p3	p4	error	maxtries	MedAE Percent	Runtime
16425	16375	16375	16361	0.01	24	0.02	7.13 s
16211	16375	16375	16575	0.1	24	0.14	7.01 s
15625	16375	16375	17161	1.0	24	0.59	6.79 s
15700	16300	16405	17131	2.0	12	0.59	6.33 s
15021	16580	16580	17355	5.0	6	0.89	5.91 s
14950	16580	16580	17426	6.0	6	0.94	5.24 s
13227	17215	17322	17772	10.0	6	1.77	5.13 s

6.3 Cylinder Explosion Simulation Dataset

To compare our load-balancing method against an unbalanced method we used the cylinder explosion simulation dataset. This dataset is the result of a simulation of the explosion of a steel pipe containing a highly explosive substance and booster pellets. The dataset contains approximately one million particles and was simulated using SPH.

The first evaluation was done to determine the impact of our method on overall runtime for visualizing the dataset using our RK-enhanced SPH interpolation method. As shown in Figure 8a, our method was slower during the initial timesteps when the particles had a more uniform distribution. However, during later timesteps, when the particles were highly non-uniformly distributed, our method outperformed the un-balanced version. Figure 8b shows the associated *MedAE* percent values.

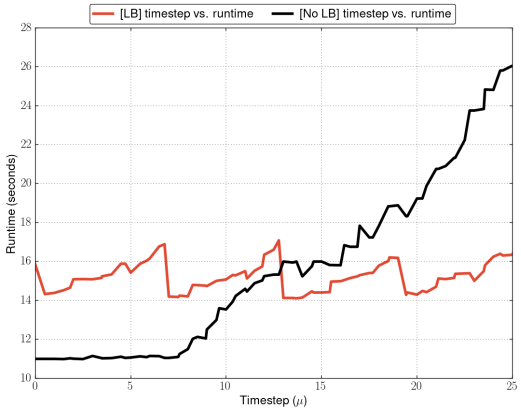
We also performed a strong-scaling test for the average runtime across 36, 72, and 144 processors. The results are shown in Figure 9, with the average theoretical runtime shown as the solid, black line and our average runtime shown as the solid red line. Figure 10 shows the theoretical and achieved runtimes for all timesteps. Our method achieved linear speed-up.

7 Discussion

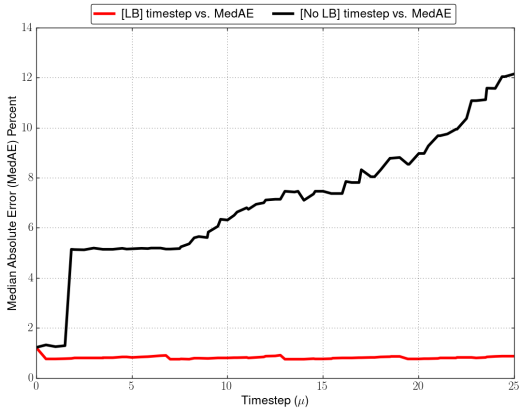
We discuss how our work compares to other load-balancing approaches and provide a comparison summary in a table. We discuss limitations of our method.

7.1 Comparison with Other Methods

Table 2 provides a comparison summary of our method with other algorithms used for load-balancing particle datasets. The column *Ghost Particles* indicates whether an algorithm handles the re-distribution of ghost particles or layers when load-balancing; the column *Adjustable Accuracy* indicates whether an algorithm can achieve various levels of



(a) Runtime: Load-balanced (red) vs. Non-load-balanced (black).



(b) MedAE: Load-balanced (red) vs. Non-load-balanced (black).

Fig. 8: The cylinder explosion simulation dataset was evaluated on the RZTopz Linux Cluster using 36 processors. **(a)** and **(b)** show the resulting runtime and median absolute error (MedAE) percent values, respectively, of the load-balanced and non-load-balanced methods.

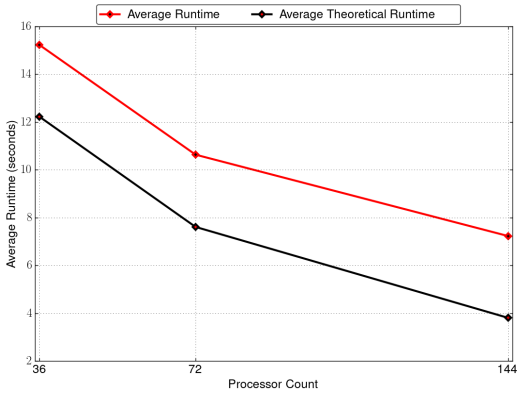


Fig. 9: Average runtime with the number of processors increasing from 36 to 72 and to 144.

accuracy when load-balancing; the column *Centralized Comms* indicates whether an algorithm requires a dedicated node for communication between processors; the column *Workload Calc* indicates whether an algorithm needs to calculate the workload before load-balancing; the column *Particles Only* indicates whether an algorithm only needs to exchange particle data; and, finally, the column *Scalable* indicates whether there are no restrictions to the scalability of an algorithm.

The approach described by J. Zhang *et al.* [23] performs load-balancing using k-d trees that are constrained in the overlapping regions of ghost layers. Our work does not have this constraint, and ghost particles are re-distributed, as needed, at the same time the regular particles are re-distributed. Further, we use an adaptive bin refinement technique when computing the median during tree construction that allows us to change the accuracy of our particle load-balancing based on various trade-offs, e.g., runtime.

The method of D. Zhang *et al.* [22] uses a binary tree structure to partition a spatial region. However, this method must perform workload calculations to quantify imbalance before performing load-balancing. This cost can become computationally prohibitive for large datasets, since it involves centralized communication for adjusting partition lines. Furthermore, this algorithm does not handle ghost layers. Our approach does not require centralized communication or workload calculations, and it handles ghost particles seamlessly.

Morozov and Perterka [19] introduced a method that uses a k-d tree to partition a dataset into blocks. Moving data blocks for dynamic load-balancing can be expensive in parallel, they demonstrated in their research. We only exchange particles, including ghost particles, which are not handled in the work of Morozov and Perterka.

Marzouk and Ghoniem [13] described an algorithm based on weighted k-means clustering for data partitioning and dynamic load-balancing. The limitation of this method is that it keeps copies of all particle positions and weights on each processor. This places a memory constraint on datasets and eliminates the ability to arbitrarily scale to large datasets. Our method does not have this constraint. In addition, a k-means approach generally produces approximately equally sized clusters [10], limiting the accuracy achievable for load-balancing. Again, our scheme does not have this limitation.

Table 2: Comparison summary with other load-balancing methods.

Method	Ghost Particles	Adjustable Accuracy	Centralized Comms	Workload Calc	Particles only	Scalable
Our method	•	•			•	•
J. Zhang *et al.* [23]					•	•
D. Zhang *et al.* [22]			•	•	•	
Morozov and Perterka [19]					•	
Marzouk and Ghoniem [13]			•			

7.2 Limitations and Possible Improvements

Our method assumes that an equal number of particles leads to optimal load-balancing. This assumption works well for our datasets. However, in general, this can lead to non-optimal load-balancing when workload varies significantly across processors.

Our adaptive bin refinement strategy uses two important parameters, *maxtries* and *error*, which determine the level of accuracy of load-balancing. A more expansive grid search should be done to fully understand the optimal values for these parameters.

The reload-balancing rate, α, was derived through trial and error for our datasets. A dynamic technique is highly desirable to determine the proper value of α for any dataset. If the value of α is too small, then the load-balancing overhead will become high, off-setting the gains from a balanced distribution of the data; if the value of α is too high,

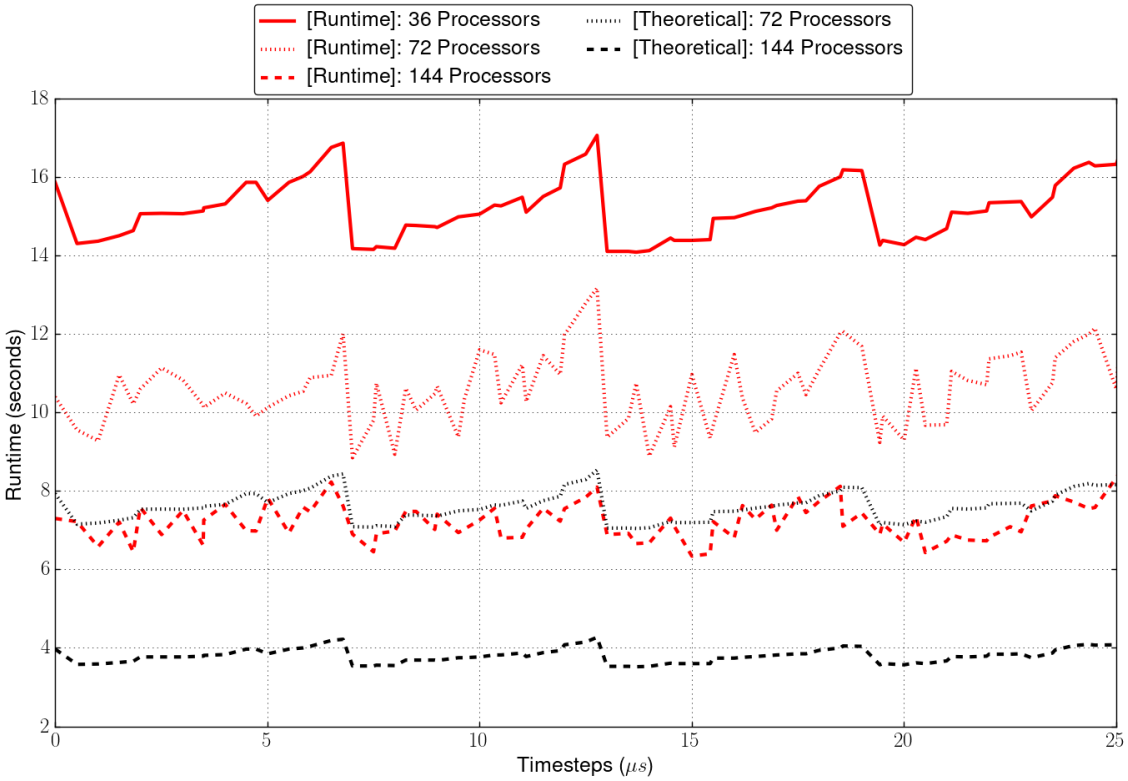


Fig. 10: Strong-scaling evaluation using the cylinder explosion dataset with different numbers of processors. The red solid, dotted, and dashed lines relate to our method when using 36, 72, and 144 processors. The black dotted and dashed lines represent the theoretical runtimes for 72 and 144 processors, respectively.

then load-balancing will not be performed frequently enough, resulting in disproportional processor workloads.

8 Conclusion

We have presented a novel load-balancing strategy that improves the workload distribution and scalability for particle datasets. Our spatial partitioning algorithm, using ABR, allows us to achieve near-optimal load-balancing and work-load distribution. Our technique is not constrained by ghost particles/layers that are seamlessly re-distributed with the normal particle data.

Our future work will focus on improving the optimization procedure for determining the values of the two parameters used in the ABR algorithm by performing a more expansive grid search. Further, we will investigate possibilities for determining an ideal load-balancing rate based on a dynamic approach defining optimal values for a variety of datasets. Finally, we plan to perform a more extensive evaluation of our method using more processors and larger datasets.

Acknowledgments

Auspices: Lawrence Livermore National Laboratory is operated by Lawrence Livermore National Security, LLC, for the U.S. Department of Energy, National Nuclear Security Administration under Contract DE-AC52-07NA27344. LLNL-CONF-XXXX.

References

[1] U. Ayachit. *The Paraview Guide: A Parallel Visualization Application*. Kitware, Inc., 2015.

[2] T. Belytschko, Y. Krongauz, D. Organ, M. Fleming, and P. Krysl. Meshless Methods: An Overview and Recent Developments. *Computer Methods in Applied Mechanics and Engineering*, 139(1):3–47, 1996. doi: 10.1016/S0045-7825(96)01078-X

[3] J. L. Bentley. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975. doi: 10.1145/361002.361007

[4] H. Childs, E. Brugger, B. Whitlock, J. Meredith, S. Ahern, D. Pugmire, K. Biagas, M. Miller, C. Harrison, G. H. Weber, H. Krishnan, T. Fogal, A. Sanderson, C. Garth, E. W. Bethel, D. Camp, O. Rübel, M. Durant, J. M. Favre, and P. Navrátil. VisIt: An End-User Tool For Visualizing and Analyzing Very Large Data. In *High Performance Visualization–Enabling Extreme-Scale Scientific Insight*, pp. 357–372. Oct 2012.

[5] G. Di Fatta and D. Pettinger. Dynamic Load Balancing in Parallel KD-Tree K-Means. In *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, pp. 2478–2485. IEEE, 2010.

[6] N. Frontiere, C. D. Raskin, and J. M. Owen. CRKSPH-A Conservative Reproducing Kernel Smoothed Particle Hydrodynamics Scheme. *arXiv preprint arXiv:1605.00725*, 2016.

[7] R. A. Gingold and J. J. Monaghan. Smoothed Particle Hydrodynamics: Theory and Application to Non-Spherical Stars. *Monthly Notices of the Royal Astronomical Society*, 181(3):375–389, 1977.

[8] K. Griffin and C. Raskin. Scalable Rendering of Large SPH Simulations using an RK-enhanced Interpolation Scheme on Constrained Datasets. In *Large Data Analysis and Visualization (LDAV), 2016 IEEE 6th Symposium on*, pp. 95–96. IEEE, 2016.

[9] W. D. Gropp, W. Gropp, E. Lusk, A. Skjellum, and A. D. F. E. E. Lusk. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, vol. 1. MIT press, 1999.

[10] J. A. Hartigan and M. A. Wong. Algorithm AS 136: A K-Means Clustering Algorithm. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 28(1):100–108, 1979.

[11] W. K. Liu, S. Jun, Y. F. Zhang, et al. Reproducing Kernel Particle Methods. *International Journal For Numerical Methods in Fluids*, 20(8-9):1081–1106, 1995.

[12] L. B. Lucy. A Numerical Approach to the Testing of the Fission Hypothesis. *The Astronomical Journal*, 82:1013–1024, 1977.

[13] Y. M. Marzouk and A. F. Ghoniem. K-means Clustering For Optimal Partitioning and Dynamic Load Balancing of Parallel Hierarchical N-body Simulations. *Journal of Computational Physics*, 207(2):493–528, 2005.

[14] J. J. Monaghan. Why Particle Methods Work. *SIAM Journal on Scientific and Statistical Computing*, 3(4):422–433, 1982.

[15] J. J. Monaghan. An Introduction to SPH. *Computer Physics Communications*, 48(1):89–96, 1988.

[16] K. Moreland. IceT Users Guide and Reference. *Sandia National Lab, Tech. Rep*, 2011.

[17] K. Moreland, W. Kendall, T. Peterka, and J. Huang. An Image Compositing Solution at Scale. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 25. ACM, 2011.

[18] K. Moreland, B. Wylie, and C. Pavlakos. Sort-Last Parallel Rendering For Viewing Extremely Large Data Sets on Tile Displays. In *Proceedings of the IEEE 2001 Symposium on Parallel and Large-Data Visualization and Graphics*, pp. 85–92. IEEE Press, 2001.

[19] D. Morozov and T. Peterka. Efficient Delaunay Tessellation Through K-D Tree Decomposition. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, p. 62. IEEE Press, 2016.

[20] J. Owen. ASPH Modeling of Material Damage and Failure. In *Proceedings of the Fifth International SPHERIC Workshop*, pp. 297–304, 2010.

[21] M. S. Shadloo and M. Yildiz. Numerical Modeling of Kelvin–Helmholtz Instability Using Smoothed Particle Hydrodynamics. *International Journal for Numerical Methods in Engineering*, 87(10):988–1006, 2011.

[22] D. Zhang, C. Jiang, and S. Li. A Fast Adaptive Load Balancing Method For Parallel Particle-Based Simulations. *Simulation Modelling Practice and Theory*, 17(6):1032–1042, 2009.

[23] J. Zhang, H. Guo, F. Hong, X. Yuan, and T. Peterka. Dynamic Load Balancing based on Constrained K-D Tree Decomposition for Parallel Particle Tracing. *IEEE Transactions on Visualization and Computer Graphics*, 2017.