A highly scalable massively parallel fast marching method for the Eikonal equation

Jianming Yang*, Frederick Stern

IIHR – Hydroscience and Engineering, University of Iowa, Iowa City, IA 52242, USA

Abstract

The fast marching method is a widely used numerical method for solving the Eikonal equation arising from a variety of scientific and engineering fields. It is long deemed inherently sequential and an efficient parallel algorithm applicable to large-scale practical applications is not available in the literature. In this study, we present a highly scalable massively parallel implementation of the fast marching method using a domain decomposition approach. Central to this algorithm is a novel restarted narrow band approach that coordinates the frequency of communications and the amount of computations extra to a sequential run for achieving an unprecedented parallel performance. Within each restart, the narrow band fast marching method is executed; simple synchronous local exchanges and global reductions are adopted for communicating updated data in the overlapping regions between neighboring subdomains and getting the latest front status, respectively. The independence of front characteristics is exploited through special data structures and augmented status tags to extract the masked parallelism within the fast marching method. The efficiency, flexibility, and applicability of the parallel algorithm are demonstrated through several examples. These problems are extensively tested on grids with up to 1 billion points using different numbers of processes ranging from 1 to 65536. The effects of memory bandwidths, domain decomposition configurations, and strides sizes are carefully investigated. In these computations, sustained super-linear parallel speedups are obtained using thousands of processes and remarkable parallel efficiencies are achieved using tens of thousands of processes. Detailed

*Corresponding author. Tel.: +1 319 335 5749; Fax: +1 319 335 5238. Email address: jianming-yang@uiowa.edu (Jianming Yang)
pseudo codes for both the sequential and parallel algorithms are provided to illustrate the simplicity of the parallel implementation and its similarity to the sequential narrow band fast marching algorithm.

**Keywords:** Eikonal equation, Static Hamilton–Jacobi equation, Distance function, Level set, Reinitialization, Fast marching method, Restarted narrow band approach, Parallel algorithm, Domain decomposition, Massively parallel implementation

1. Introduction

The fast marching method [7] is a widely used numerical method for solving the Eikonal (static Hamilton–Jacobi) equation, which is a first-order hyperbolic partial differential equation arising from a variety of applications, such as computational geometry, computational fluid dynamics, computer vision, materials science, optimal control, etc. It is a non-iterative algorithm based on upwind difference schemes, which resembles Dijkstra’s method [1] for finding the shortest path on a network. The fast marching method has theoretically optimal complexity in its operation count by exploring the causality of the Eikonal equation and adapting a one-pass updating strategy. The Eikonal equation describes nonlinear boundary value problems in which the information from the boundary propagates away along characteristics. In the fast marching method, upwind difference schemes are used to discretize the Eikonal equation at a given grid point, such that the stencil contains only neighboring points with valid values (or, upwind points) and the causality of the equation is strictly followed. Moreover, a heap priority queue is used to march the solution in a rigorous increasing (decreasing for the negative solution) order. Therefore, the number of times that a point is visited is minimized and no iterations are involved in the whole process. Since the run-time complexity of reordering of a heap of length \( n \) is \( O(\log n) \), the fast marching method has a total operation count of \( O(N \log N) \) for a case involving \( N \) grid points.

Numerous improvements and extensions have been developed since the introduction of the fast marching method. It has been applied to a wide range of scientific and engineering problems. The reader is referred to [6] for details. The extensive usage of the fast marching method in large-scale applications, however, has been severely limited due to the lack of an efficient parallel algorithm. Two major advantages of the fast marching method
over other techniques that makes its parallelization particularly desirable are
the narrow band formulation and the monotonic increasing order of the solution. In many cases, only the solution within a narrow band very close
to the boundary/interface is of interest and the number of these points usually is much smaller than the total number of grid points. In addition, the
monotonic order of the solution, consistent with the propagation direction of
the information from the boundary, is inevitable in some applications, e.g.,
field extension of information (e.g., velocity, scalars, etc.) from the boundary/interface to the surrounding domain. The outstanding issue of parallelizing the fast marching method has received much less attention. One
possible reason is that the fast marching method is long deemed inherently sequential and has no straightforward parallelism as found in an iterative
method such as the fast sweeping method [11, 12]. Note there are several
parallel implementations of iterative methods (e.g., [3] and [9], among others) for shared memory parallel architectures, especially, graphics processing
units (GPU) most recently. On the other hand, Tsitsiklis [8] developed two
single-pass algorithms using an optimal control approach: an $O(N \log N)$
algorithm with a binary heap data structure (a Dijkstra-like method similar
to the fast marching method) and an $O(N)$ algorithm using a bucket
data structure, and provided a shared-memory parallel implementation for
the latter. In general, these shared-memory based parallel algorithms are
very difficult to be extended to distributed memory parallel architectures, on
which the coarse-grain parallelization prototypes, usually based on domain
decompositions, are prevalent.

The first attempt to parallelize the fast marching method based on a
domain decomposition technique was reported by Herrmann [2]. Essentially,
the computational domain was decomposed into non-overlapping subdomains
and ghost points (one layer for a first-order scheme) are used for communications between subdomains. For instance, if a just-accepted grid point is
also in the ghost point region of a neighboring block, then the information of
this point will be sent to the target neighbor. In the mean time, each process
repeatedly checks for ghost point updates from neighbors. A rollback mech-
anism was introduced to revoke the valid status of all pre-accepted points
whenever a ghost point turned to valid status with a smaller value than these
points. The asynchronous communications in this algorithm were quite in-
volved and difficult to implement. The rollback operations introduced signif-
icant communication and computation overheads and considerably limited
the parallel performance.
A major problem with the parallel algorithms for the Eikonal equation in the literature, either developed for shared memory architectures or for distributed memory platforms, is their very limited scalability for solving large scale practical problems. Most approaches were developed for small problems running with a few or tens of cores/threads and a couple of hundred cores/threads at their best. In addition, many parallel algorithms were formulated very different from their sequential counterparts and could not be implemented by simply extending the sequential algorithms. On the other hand, the Eikonal equation is usually a less-critical component in many scientific and engineering applications; hence its parallelization has to conform with those of the major components instead of requiring a stand-alone treatment.

In this study, a highly scalable parallel algorithm of the fast marching method based on a domain decomposition technique, which was first briefed in [10], is discussed in detail. Developed with serious large-scale practical applications in mind, this massively parallel fast marching method can give a remarkable parallel performance on billion-point grids using tens of thousands of processes, whereas its implementation is surprisingly simple and straightforward. Actually, the sequential fast marching algorithm is directly incorporated into the parallel algorithm with only a few minor plain modifications. In particular, the procedure central to our parallel algorithm is a novel restarted narrow band approach, in which the fronts advance at a specified stride during each restart. Therefore, it is fully consistent with the narrow band idea in the fast marching level set method [7]. Basically, for each restart of front advancing, a global bound is first determined according to the given stride size, the (essentially sequential) fast marching algorithm is then executed; updated points in the overlapping regions of neighboring subdomains are collected and exchanged; with the new data from neighboring subdomains the fast marching algorithm is carried out once more to bring the fronts everywhere up to the designated bound. Only simple synchronous communication modes are employed in the whole process. This algorithm exploits the independence of front characteristics to extract the parallelism deeply buried under the apparent sequentiality of the fast marching method. For example, special data structures are designed for two-sided interface problems, such that both the positive and negative fronts of the interface can be advanced concurrently. In addition, when a subdomain receives updated function values for grid points in the overlapping regions, none of the grid points with larger (absolute) values in this subdomain will
be reset uniformly as if with a rollback mechanism. This is because many of these larger values might be computed following other characteristics that are independent of points with lower values received from neighboring processes. Augmented tags are introduced to define the point status precisely. Associated with a few slight modifications in the sequential algorithm, points that are influenced by the incoming data can be refreshed without being singled out for any special treatments.

The rest of this paper is organized as follows: In the next section the sequential fast marching method is given. Then the data structures for two-sided interface problems are introduced. In the parallel fast marching method part the parallel algorithm is described thoroughly, detailed pseudo codes are provided for a side-by-side comparison and straightforward implementations of the sequential and parallel algorithms. In the results section, three cases with different interface configurations are considered. The $O(N \log N)$ algorithm complexity and the first-order accuracy of the fast marching method are verified first. A series of tests on different grids ranging from less than $40k$ to more than $1$ billion points using one to $65536$ CPU cores are performed to demonstrate the parallel speedups and efficiencies. The effects of memory bandwidths, domain decomposition strategies, and the stride sizes are also studied. Some concluding remarks are provided in the final section.

2. Sequential fast marching method

2.1. Eikonal equation and finite difference discretization

The fast marching method solves the stationary boundary value problem defined by the Eikonal equation:

$$|\nabla \psi(x)|F(x) = 1, \quad x \in \Omega \setminus \Gamma, \psi(x) = 0, \quad x \in \Gamma \subset \Omega,$$  \hspace{1cm} (1)

where $\Omega$ is a domain in $\mathbb{R}^n$, $\Gamma$ is the initial interface (boundary), and $F(x)$ is a positive speed function, with which the interface information propagates in the domain.

To solve Eq. (1) numerically, domain $\Omega$ has to be discretized first. Here a regular domain in $\mathbb{R}^3$ defined by $\Omega = [x_{\text{min}}, x_{\text{max}}] \times [y_{\text{min}}, y_{\text{max}}] \times [z_{\text{min}}, z_{\text{max}}]$ is partitioned as $\Omega = \bigcup \Delta \Omega_{i,j,k}$, where $1 \leq i \leq nx, 1 \leq j \leq ny, 1 \leq k \leq nz$, and $\Delta V_{i,j,k} = [x_{i-1/2}, x_{i+1/2}] \times [y_{j-1/2}, y_{j+1/2}] \times [z_{k-1/2}, z_{k+1/2}]$. $x_{1/2} = x_{\text{min}}, x_{nx+1/2} = x_{\text{max}}, y_{1/2} = y_{\text{min}}, y_{ny+1/2} = y_{\text{max}},$ and $z_{1/2} = z_{\text{min}}, z_{nz+1/2} = z_{\text{max}}$.
\[ z_{\text{max}} \] define \( \partial \Omega \), the boundary of \( \Omega \). Notice that the function \( \psi \) is defined at the center of each computational cell (i.e., \( \Delta \Omega_{i,j,k} \)). Therefore, no function is defined at the domain boundary \( \partial \Omega \), ghost points are used to facilitate the imposition of boundary conditions. Fig. 1 shows the interface \( \Gamma \) and the computational domain, \( \Xi = \Omega + \Theta \) (i.e., the combination of the discretized physical domain \( \Omega \) and ghost point zone \( \Theta \)). To simplify the discussion, uniform grid distribution in each direction is considered in this study, i.e., \( \Delta \Omega_{i,j,k} = \Delta x \times \Delta y \times \Delta z \), although the methodology to be discussed is not limited to a uniform grid.

\[ \begin{array}{c}
\text{A grid point inside the physical domain} \\
\text{A ghost point outside the physical domain}
\end{array} \]

**Figure 1:** The computational domain \( \Xi \) and the interface \( \Gamma \).

The Godunov-type finite difference scheme given in [4], which satisfies the entropy condition in hyperbolic conservation laws, can be used to approximate Eq. (1) on the computational domain as follows

\[
\left[ \begin{array}{c}
\max(D_{i,j,k}^{-x} \psi, -D_{i,j,k}^{+x} \psi, 0)^2 \\
\max(D_{i,j,k}^{-y} \psi, -D_{i,j,k}^{+y} \psi, 0)^2 \\
\max(D_{i,j,k}^{-z} \psi, -D_{i,j,k}^{+z} \psi, 0)^2
\end{array} \right]^{1/2} = \frac{1}{F_{i,j,k}}, \quad (2)
\]

where the operators \( D_{i,j,k}^{-x} \) and \( D_{i,j,k}^{+x} \) define the backward and forward dif-
ference approximations to the spatial derivative \( \partial \psi / \partial x \), respectively. In this study, first-order schemes are used:

\[
D_{i,j,k}^{-x} \psi = \frac{\psi_{i,j,k} - \psi_{i-1,j,k}}{\Delta x}, \quad D_{i,j,k}^{+x} \psi = \frac{\psi_{i+1,j,k} - \psi_{i,j,k}}{\Delta x}.
\] (3)

The operators for the \( y \) and \( z \) directions are defined similarly. Eq. (2) gives a quadratic equation for \( \psi_{i,j,k} \).

2.2. Algorithm

![Figure 2: Interface initialization.](image)

Observe that Eq. (2) has a very special upwind structure, i.e., \( \psi_{i,j,k} \) only depends on the neighboring points of smaller value. The fast marching method takes advantage of this fact by solving Eq. (2) using only the upwind points and building up the whole solution following a systematical manner from the point of the smallest value. To identify the upwind directions and establish the order of updating, each grid point \( i, j, k \) is labeled by a status tag \( G_{i,j,k} \): i) \( G_{i,j,k} = \text{KNOWN} \), if this point contains a final function value, thus give the upwind direction; ii) \( G_{i,j,k} = \text{BAND} \), if this point contains a function value updated by its neighboring \text{KNOWN} point(s), but may be further
updated by any new KNOWN neighbors; and iii) $G_{i,j,k} = \text{FAR}$, if this point is in the downwind side and does not have any KNOWN neighbors yet. The point with the smallest value in the BAND set is located and moved into the KNOWN set, and then its neighboring BAND and FAR points can be updated and re-categorized. This step repeats until all points in the domain or within a pre-defined narrow band become KNOWN.

**Algorithm 1** Interface initialization:

**INITIALIZE_INTERFACE.**

1: $\psi \leftarrow +\infty$
2: $G \leftarrow \text{FAR}$
3: for all $(i, j, k) \in \Xi$ such that $\psi_{i,j,k} \in \psi^0$ is adjacent to $\Gamma$ do
4: \hspace{1em} $\psi_{i,j,k} \leftarrow \psi^0$
5: \hspace{1em} $G_{i,j,k} \leftarrow \text{KNOWN}$
6: end for

*Figure 3: Heap initialization.*

In the present study, detailed pseudo codes are given to emphasize the similarities and differences between the sequential and parallel methods. In many applications, such as the first-arrival traveltime calculation for seismic
wave propagation, wall distance calculation for turbulence modeling, etc., only the solution in the positive region $\Omega^+$ is required. Thus the negative values in $\psi^0$ are not needed for a first-order scheme. For brevity, in this section the algorithms are presented for one-sided boundary value problems. The algorithm for two-sided interface problems will be addressed in the next section.

As a boundary value problem, the boundary or interface condition has to be specified for the discretized Eikonal equation in the solution procedure. In the actual implementation, this is fulfilled by the interface initialization procedure. Fig. 2 shows the interface initialization step, and the corresponding Algorithm 1 details the operations for initializing the interface. Initially, $+\infty$ and FAR are assigned to each point $\psi_{i,j,k}$ as its function value and status tag, respectively. A FAR point is identified by the blanked and grey-shadowed cells shown in Fig. 2 and other figures. Then, all grid points immediately adjacent to the interface are assigned values $\psi^0$ (analytical solution in the present study) and tagged as KNOWN as shown in Fig. 2.

**Algorithm 2** Heap initialization:

**INITIALIZE_HEAP.**

1: $\text{size}_0 \leftarrow 0$
2: for all $(i, j, k) \in \Xi$ such that $G_{i,j,k} = \text{KNOWN}$ do
3: \hspace{1em} UPDATE_NEIGHBORS($i, j, k$)
4: end for

The fast marching method relies on a binary heap structure, which has to be initialized too. In the heap initialization step given in Algorithm 2, the initial heap size is set to zero; then, for each KNOWN point in the computational domain, its neighboring points are updated by solving Eq. (2) and tagged as BAND points. It should be noted that all points in the domain including both physical and ghost regions are treated in the same manner as shown in Fig. 3.

The procedure for updating neighboring points of a KNOWN point is described in Algorithm 3, which is the major operation involved in the fast marching method. In 3D, for a KNOWN point $(i, j, k)$, its neighbors to be considered are $(i - 1, j, k)$, $(i + 1, j, k)$, $(i, j - 1, k)$, $(i, j + 1, k)$, $(i, j, k - 1)$, and $(i, j, k + 1)$. For each point among these neighbors, if it is inside the computational domain (ghost points are included) and it is not a KNOWN point, then Eq. (2) is solved at this point to obtain a new function value. If the new
value is smaller than the present value at this point, then its function value will be updated with the new value. The final step is to check the status of this point, if it is a FAR point, it will be added to the heap; otherwise, as an existing BAND point its position in the heap will be updated.

The heap data structure guarantees a strict order of increasing function values for solving the Eikonal equation in the fast marching method. In this study, a binary heap data structure similar to what described in [6] is used. The standard heap operations in an implementation of the indexed priority queue algorithm, i.e., Insert Heap, Locate Min, Remove Min, Up Heap (also used in Insert Heap), and Down Heap (used in Remove Min), are available in textbooks for algorithms (e.g., [5]).

**Algorithm 3** Update the neighbors of a KNOWN point:

**UPDATE_NEIGHBORS**(i, j, k).

1: for all \((l, m, n)\) such that \((|l - i| + |m - j| + |n - k|) = 1\) do
2:     if \((l, m, n) \in \mathcal{Z}\) then
3:         if \(G_{l,m,n} \neq \text{KNOWN}\) then
4:             \(\psi_{\text{temp}} \leftarrow \text{SOLVE_QUADRATIC}(l, m, n)\)
5:             if \(\psi_{\text{temp}} < \psi_{l,m,n}\) then
6:                 \(\psi_{l,m,n} \leftarrow \psi_{\text{temp}}\)
7:                 \(G_{l,m,n} \leftarrow \text{BAND}\)
8:                 if \((l, m, n) \not\in \mathcal{H}\) then
9:                     \text{INSERT_HEAP}(l, m, n)
10:                else
11:                    \text{UP_HEAP}(l, m, n)
12:             end if
13:         end if
14:     end if
15: end for

Figure 2 gives a 2D example of updating neighbors of a newly added KNOWN point \((i, j)\), which shows a close-up view of the procedure illustrated in Fig. 1. Point \((i+1, j)\) is an existing BAND point, but its function value was calculated using Eq. (2) solely from KNOWN point \((i+1, j+1)\). Now it has two KNOWN neighbors, and Eq. (2) is re-solved to possibly update its function value. For point \((i, j-1)\), it was a FAR point, now with a KNOWN neighbor, Eq. (2) is solved to update its function value and its status is changed to
The solution procedure of the quadratic equation, Eq. (2), is described in Algorithm 4. Only the operation for the $x$ direction is detailed as other two directions are very similar. For simplicity, assume that the only upwind point is $(l-1, m, n)$ in the quadratic equation, then Eq. (2) will become the following form

$$(D_{l,m,n}^{-x} \psi)^2 = \frac{1}{F_{l,m,n}^2},$$  \hspace{1cm} (4)

or,

$$\left(\frac{\psi_{l,m,n} - \psi_{l-1,m,n}}{\Delta x}\right)^2 = \frac{1}{F_{l,m,n}^2},$$  \hspace{1cm} (5)

since $\psi_{l+1,m,n} = +\infty$ in Eq. (2). Then a standard quadratic equation $a\psi_{l,m,n}^2 + b\psi_{l,m,n} + c = 0$ can be obtained with the following coefficients:

$$a = \frac{1}{\Delta x^2}, \quad b = -\frac{2\psi_{l-1,m,n}}{\Delta x^2}, \quad c = \frac{\psi_{l-1,m,n}^2}{\Delta x^2} - \frac{1}{F_{l,m,n}^2}. \hspace{1cm} (6)$$

And only the solution

$$\psi_{\text{temp}} = \frac{-b + \sqrt{b^2 - 4ac}}{2a} \hspace{1cm} (7)$$
is acceptable if available, since \( \psi_{\text{temp}} \geq \psi_{t-1,m,n} = -b/2a \). The above example shows the case in which the left neighbor in the \( x \) direction is the only upwind point. In practice, all directions are checked for possible upwind points. The whole algorithm only uses upwind points in set \( \text{KNOWN} \) to advance the front, which guarantees that the final result is the correct, unique viscosity solution to the Eikonal equation.

Algorithm 5 shows the loop for propagating the front with a narrow band defined by its width \( \text{width}_{\text{band}} \). It is evident that the full field algorithm can be obtained by removing the band width related termination condition in the above algorithm or simply setting the band width to \(+\infty\). An empty heap is the other loop termination condition, i.e., in the given region, all \( \text{BAND} \) points have been given \( \text{KNOWN} \) status and there are no more \( \text{FAR} \) points can be added to the \( \text{BAND} \) category. In each loop, this termination condition is checked first; if not satisfied then the \( \text{BAND} \) point on the top of the heap is located and its value is checked against the band width termination condition; if still not satisfied then this point is tagged as \( \text{KNOWN} \) and removed from the heap. For this newly added \( \text{KNOWN} \) points, its neighboring points will be checked and updated if possible. Fig. 5 shows such a step in the loop of the narrow band fast marching method, in which the \( \text{BAND} \) point with the minimum function value in the heap is upgraded to \( \text{KNOWN} \) status; then among its neighbors, a \( \text{FAR} \) point is upgraded to \( \text{BAND} \) status with an updated function value, and the function value of a \( \text{BAND} \) point is also updated.

Algorithm 6 gives the overall solution procedure for the sequential fast marching method. It simply consists of two initialization steps and the narrow band marching procedure. For further details, the reader is referred to [6] and the references therein.

3. Data structures for two-sided interface problems

As discussed in the previous section, for interface problems with both positive and negative regions, the sequential fast marching method can be applied in a region-by-region manner; and the data structures required in the algorithm is exactly the same as those for single region problems. Here is a simple approach to apply the min-heap data structure introduced above to two-sided interface problems: first, the signs of the \( \text{KNOWN} \) points from the interface initialization step are inverted, i.e., the \( \text{KNOWN} \) points in the negative region become positive and vice versa for the positive region; then, the fast marching method with the min-heap data structure is applied to the negative
Algorithm 4 Solve the quadratic equation:

SOLVE_QUADRATIC\((l, m, n)\).

1: \(\psi_{\text{temp}} \leftarrow +\infty\)
2: \(a \leftarrow 0\)
3: \(b \leftarrow 0\)
4: \(c \leftarrow -F_{l,m,n}^{-2}\)
5: Accumulate \(a, b,\) and \(c\) for the \(x\) direction:
6: \(d \leftarrow 0\)
7: if \((l - 1, m, n) \in \Xi\) then
8: if \(G_{l-1,m,n} = \text{KNOWN}\) then
9: \(d \leftarrow -1\)
10: end if
11: end if
12: if \((l + 1, m, n) \in \Xi\) then
13: if \(G_{l+1,m,n} = \text{KNOWN}\) then
14: if \(d = 0\) then
15: \(d \leftarrow +1\)
16: else \(\{d = -1\}\)
17: if \(\psi_{l+1,m,n} < \psi_{l-1,m,n}\) then
18: \(d \leftarrow +1\)
19: end if
20: end if
21: end if
22: end if
23: if \(d \neq 0\) then
24: \(a \leftarrow a + \Delta x^{-2}\)
25: \(b \leftarrow b + \Delta x^{-2} \psi_{l+d,m,n}\)
26: \(c \leftarrow c + \Delta x^{-2} \psi_{l+d,m,n}^2\)
27: end if
28: Accumulate \(a, b,\) and \(c\) for the \(y\) direction
29: Accumulate \(a, b,\) and \(c\) for the \(z\) direction
30: if \(b^2 - ac \geq 0\) then
31: \(\psi_{\text{temp}} \leftarrow \frac{b + \sqrt{b^2 - ac}}{a}\)
32: end if
Algorithm 5 Front propagation within the narrow band:

\textsc{March\_Narrow\_Band}.

1: \textbf{loop}
2: \hspace{1em} if $\text{size}_N = 0$ then
3: \hspace{2em} exit loop
4: \hspace{1em} end if
5: \hspace{1em} $(i,j,k) \leftarrow \text{Locate\_Min}$
6: \hspace{1em} if $\psi_{i,j,k} > \text{width}_{\text{band}}$ then
7: \hspace{2em} exit loop
8: \hspace{1em} end if
9: \hspace{1em} $G_{i,j,k} \leftarrow \text{KNOWN}$
10: \hspace{1em} $\text{Remove\_Min}$
11: \hspace{1em} $\text{Update\_Neighbors}(i,j,k)$
12: \hspace{1em} end loop

Figure 5: One step of the narrow band fast marching method.
Algorithm 6 Sequential narrow band fast marching method:
\texttt{NARROW\_BAND\_FAST\_MARCHING}.
\begin{algorithmic}
\STATE {\bf Initialize} Interface
\STATE {\bf Initialize} Heap
\STATE {March Narrow Band}
\end{algorithmic}

region; after that, the signs of all points in the negative region and the \texttt{KNOWN} points in the positive region (carry a negative sign from the operation in the first step) are inverted again to return to the correct signs; and finally, the fast marching method is applied to the positive region. It is evident in the above procedure for two-sided interface problems, a strict order of increasing (positive) function values for updating the solution is guaranteed.

A simple parallelization can be implemented by following the same philosophy such that the parallel algorithm is applied to a single region at a time and the information propagation in this region has to be completed before moving to the region of an opposite sign. This approach essentially further decomposes the computational domain into positive and negative subdomains. A major problem of this approach is that all communications between neighboring subdomains and synchronizations among all subdomains have to be performed separately for both the positive and negative regions, which practically doubles the number of communication calls. On the other hand, it greatly deteriorates the load imbalance inherently rooted in solving the Eikonal equation on distributed memory parallel computers, as the information only propagates away from an interface.

In this study, novel data structures are designed to update both the positive and negative regions concurrently within our parallel algorithm. One particularly attractive property of the new data structure is that the fast marching algorithm given in the previous sections is barely changed. As shown in Algorithm 7, an outer \texttt{for} loop, in which the counter \texttt{s} has a value of $-1$ for the negative region or $1$ for the positive region, is added to the \texttt{MARCH\_NARROW\_BAND} Algorithm. Correspondingly, the scalar variables for the size of the binary heap and the width of the narrow band are changed into three-element arrays, i.e., \texttt{size}$_N(-1:1)$ and \texttt{width}$_\text{band}(-1:1)$, to be used inside the loop. However, the heap still keeps its one-dimensional array-backed structure through the incorporation of negative indices. That is, the heap for a two-sided interface problem will range from \texttt{size}$_N(-1)$ (a non-positive integer) to \texttt{size}$_N(1)$ (a non-negative integer). The side infor-
mation $s$ is added to all the functions in the priority queue algorithm, i.e., 
\texttt{Insert\_Heap}, \texttt{Locate\_Min}, \texttt{Remove\_Min}, \texttt{Up\_Heap}, and \texttt{Down\_Heap}. For the 
increment operations to an index $p$ in the heap, instead of $p+1$ or $p-1$ in 
the original algorithm, now they are simply $p+s$ or $p-s$. For the comparison 
operations of indices and function values, $p$ will be replaced by $s \cdot p$ or $|p|$ 
and $\psi$ becomes $s \cdot \psi$ or $|\psi|$. With this treatment, the side information for a 
\texttt{KNOWN} or \texttt{BAND} point is always available from the sign of the function value 
at the given point.

It is evident that the algorithm works for one-sided boundary value problems 
without any issues, as the heap size for the other side should be zero and 
the algorithm will not be executed for that side at all. On the other hand, the 
present approach is quite straightforward for two-sided interface problems. 
It gets rid of the positive-negative domain decomposition, which has some 
significant impacts on the parallelization of the fast marching method. For 
example, the number of communications for data exchanges and reductions 
among processes is simply halved with the new data structures as two 
communication calls with one for each side of the interface can be combined into 
one now. In many applications, the load balance can be greatly improved as 
both sides are treated in one loop without involving any data communications 
within the loop. Because the processes spend more time in computations 
before data communications are required, this is very beneficial for reducing 
network congestion and improving parallel performance.

4. Parallel fast marching method

4.1. Overlapping domain decompositions

The computational domain is divided into $npx \times npy \times npz = nprocs$ sub-
domains using a Cartesian process topology and mapped to $nprocs$ processes. 
Each process $p = 0, \cdots, nprocs - 1$ works on a subdomain identified by its 
process coordinates $(ip, jp, kp)$ ($ip = 0, \cdots, npx - 1; jp = 0, \cdots, npy - 1; 
and kp = 0, \cdots, npz - 1$) in the Cartesian process grid. For simplicity, the 
domain is divided evenly in each direction. As shown in Fig. 6 for a 2D case, 
just like the case in a sequential computation, for each subdomain $\Omega_p$ all 
subdomain boundaries, including those generated from the domain decom-
position, are patched with one layer of ghost points to obtain a ghost point 
zone $\Theta_p$ that encloses $\Omega_p$. Similarly, with the parallel algorithm the basic 
scheme is executed in $\Xi_p$ without major changes.
**Algorithm 7** Front propagation within the narrow band for two-sided interface problems:

\textbf{MARCH\_NARROW\_BAND\_TWO\_SIDED.}

1: for $s \leftarrow -1$ to 1 step 2 do
2: \hspace{1em} \textbf{loop}
3: \hspace{2em} if $\text{size}_N(s) = 0$ then
4: \hspace{3em} exit loop
5: \hspace{2em} end if
6: \hspace{2em} $(i, j, k) \leftarrow \text{LOCATE\_MIN}(s)$
7: \hspace{2em} if $|\psi_{i,j,k}| > \text{width}_{\text{band}}$ then
8: \hspace{3em} exit loop
9: \hspace{2em} end if
10: $G_{i,j,k} \leftarrow \text{KNOWN}$
11: \text{REMOVE\_MIN}(s)
12: \text{UPDATE\_NEIGHBORS}(i, j, k, s)
13: \hspace{1em} \textbf{end loop}
14: \hspace{1em} \textbf{end for}

![Diagram](image-url)

Figure 6: Domain decomposition in the parallel computation: (a) Sequential computational domain; (b) parallel computational domain.
When a boundary value problem is solved using a discretization method, usually function values at ghost points are obtained from boundary conditions for physical domain boundaries or from neighboring processes for virtual boundaries generated in the domain decomposition procedure. For the Eikonal equation, however, it makes more sense to treat the ghost points in the same way as the internal points; because the interface, which contains the boundary values, is commonly embedded in the computational domain and the information propagates away from it does not depend on the domain boundary conditions at all. Even for the case when a domain boundary is the source of information (i.e., a Dirichlet boundary condition), it still can be considered as an interface embedded in the enlarged domain with the ghost points counted in. Therefore, in the present work, the Eikonal equation is solved everywhere without distinguishing the ghost points from the others. In both the sequential and parallel algorithms, it is only necessary to make sure that a discretization stencil does not involve an inaccessible point for the current process (i.e., to avoid array out-of-bounds errors) and boundary conditions are not implemented at all. It should be noted that an interface intersects with a domain boundary should be extended into the ghost points with appropriate values before the equation is being solved.

It is obvious that the present strategy is an overlapping domain decomposition approach, since each process solves the equation everywhere including its ghost points. And the ghost points may contain better approximations to the solution than their corresponding physical domain points residing in a different process. Therefore, besides sending out values at physical domain points that are ghost points of neighboring processes, it is also necessary for a process to transfer its lastest ghost point values to its neighboring processes that share these points with the specific process. As shown in Fig. 7, a process has to exchange information with all neighboring processes that share subdomain faces, edges, and corners with it.

At a first glimpse, it may seem there are unnecessary additional computations and extra communications by including the ghost points in the equation solution and data exchange procedures, respectively. Actually, in the fast marching method, only the points in the downwind direction will be updated, which means that very likely a grid point shared by two or more processes will be visited only in one of them and the updated value of this point will be sent from this process to others. Also, only the updated points, whose size is usually much less than that of the entire shared points, will be involved in the data exchange. Moreover, for the exchange of small size data
such as ghost point updates, usually the number of message passing function calls determines the communication overhead instead of the actual data size of each exchange. On the other hand, the present overlapping domain decomposition strategy and the corresponding data exchange mechanism can greatly simplify the code structure by unifying the treatment of ghost points and physical domain points.

4.2. Augmented status tags

Algorithm 8 Interface initialization:

\begin{algorithm}
\textbf{INITIALIZE\_INTERFACE\_PARALLEL.}
\begin{algorithmic}
  \State $\psi \leftarrow +\infty$
  \State $G \leftarrow \text{FAR}$
  \ForAll{$(i, j, k) \in \Xi p$ such that $\psi_{i,j,k} \in \psi^0$ is adjacent to $\Gamma$}
    \State $\psi_{i,j,k} \leftarrow \psi^0$
    \State $G_{i,j,k} \leftarrow \text{KNOWN\_FIX}$
  \EndFor
\end{algorithmic}
\end{algorithm}
Algorithm 9 Heap initialization:
INITIALIZE_HEAP_PARALLEL.

1: $\text{size}_p(-1:1) \leftarrow 0$
2: for all $(i, j, k) \in \Xi_p$ such that $G_{i,j,k} = \text{KNOWN\_FIX}$ do
3: $s \leftarrow \text{sgn}(\psi_{i,j,k})$
4: Update_Neighbors_Parallel$(i, j, k, s)$
5: end for

Algorithm 10 Update the values of neighbors of a point newly added to set KNOWN:
UPDATE_Neighbors_Parallel$(i, j, k, s)$.

1: for all $(l, m, n)$ such that $(|l-i| + |m-j| + |n-k|) = 1$ do
2: if $(l, m, n) \in \Xi_p$ then
3: if $G_{l,m,n} \neq \text{KNOWN\_FIX}$ and $|\psi_{l,m,n}| > |\psi_{i,j,k}|$ then
4: $\psi_{\text{temp}} \leftarrow \text{SOLVE\_QUADRATIC}(l, m, n, s)$
5: if $\psi_{\text{temp}} < |\psi_{l,m,n}|$ then
6: $\psi_{l,m,n} \leftarrow s \cdot \psi_{\text{temp}}$
7: $G_{l,m,n} \leftarrow \text{BAND\_NEW}$
8: if $(l, m, n) \notin \mathcal{H}(s)$ then
9: Insert_HEAP$(l, m, n, s)$
10: else
11: Up_HEAP$(l, m, n, s)$
12: end if
13: end if
14: end if
15: end if
16: end for
Algorithm 11 Solve the quadratic equation:
SOLVE_QUADRATIC_PARALLEL(l, m, n, s).

1: $\psi_{\text{temp}} \leftarrow +\infty$
2: $a \leftarrow 0$
3: $b \leftarrow 0$
4: $c \leftarrow -F_{l,m,n}^{-2}$
5: Accumulate $a, b, c$ for the $x$ direction:
6: $d \leftarrow 0$
7: if $(l - 1, m, n) \in \Xi_p$ then
8:  if $G_{l-1,m,n} \in \text{KNOWN}$ and $|\psi_{l-1,m,n}| < |\psi_{l,m,n}|$ then
9:  $d \leftarrow -1$
10: end if
11: end if
12: if $(l + 1, m, n) \in \Xi_p$ then
13:  if $(l + 1, m, n) \in \text{KNOWN}$ and $|\psi_{l+1,m,n}| < |\psi_{l,m,n}|$ then
14:   if $d = 0$ then
15:    $d \leftarrow +1$
16:   else
17:    if $|\psi_{l+1,m,n}| < |\psi_{l-1,m,n}|$ then
18:     $d \leftarrow +1$
19:   end if
20: end if
21: end if
22: end if
23: if $d \neq 0$ then
24:  $a \leftarrow a + \Delta x^{-2}$
25:  $b \leftarrow b + \Delta x^{-2}s \cdot \psi_{l+d,m,n}$
26:  $c \leftarrow c + \Delta x^{-2}\psi_{l+d,m,n}^2$
27: end if
28: Accumulate $a, b, c$ for the $y$ direction
29: Accumulate $a, b, c$ for the $z$ direction
30: if $b^2 - ac \geq 0$ then
31:  $\psi_{\text{temp}} \leftarrow \frac{b + \sqrt{b^2 - ac}}{a}$
32: end if
As mentioned above, only portion of the shared points are involved in the data exchanges in the parallel algorithm. This is realized by further distinguish the status tag of a grid point in the BAND and KNOWN categories. In the present parallel algorithm, the FAR category remains unchanged from its definition in the sequential algorithm. The BAND category is divided into two sub-categories: a) BAND\_NEW, which is the tag for a new BAND point elevated from a FAR status; and b) BAND\_OLD, which is the tag assigned to a BAND\_NEW point in the shared regions after the position and value of this point are collected for data exchanges. The KNOWN category is divided into three sub-categories: a) KNOWN\_FIX, which is the tag for those grid points that obtain their functions during the interface initialization procedure and their values are fixed during the solution process; b) KNOWN\_NEW, which is the tag assigned to the point on top of the heap with a BAND\_NEW tag when it is to be removed from the heap; and c) KNOWN\_OLD, which is the tag assigned to a KNOWN\_NEW point in the shared regions after the position and value of this point are collected for data exchange. KNOWN\_OLD is also assigned to the point on top of the heap with a BAND\_OLD tag when it is to be removed from the heap.

The introduction of these new status tags enables minimizing the point refreshing computations and the data exchanged between processes, but barely changes the main elements of the sequential algorithm. Algorithm 8 shows the parallel version of the interface initialization procedure. Compared with the sequential version, the only differences are the replacements of $\Xi$ and KNOWN with $\Xi_p$ and KNOWN\_FIX, respectively. Likewise, the parallel version of the heap initialization procedure, i.e., Algorithm 9, follows the same modifications. As discussed in the previous section, the sign of a point value is obtained to determine which side of the heap a BAND\_NEW point should be inserted to in the Update\_Neighbors\_Parallel procedure given in Algorithm 10. In this part the condition $G_{l,m,n} \neq$ KNOWN\_FIX is of significance for the present parallel algorithm. It allows the function value at point $(l, m, n)$ with a tag KNOWN\_OLD or KNOWN\_NEW to be updated, just like a BAND or FAR point, as long as $|\psi_{l,m,n}| > |\psi_{i,j,k}|$. Also if its function value does be updated, then its tag will be reset to BAND\_NEW no matter what tag it has previously. This strong resemblance is also seen in the sequential and parallel versions of the solution procedure of the quadratic equation. As shown in Algorithm 11, the KNOWN tag in the parallel version is used to include a neighboring point with a KNOWN\_FIX, KNOWN\_OLD, or KNOWN\_NEW tag in the stencil. However, it is still necessary to make sure that the specific neighboring point is an upwind
Algorithm 12 Front propagation within the narrow band:

\texttt{March\_Narrow\_Band\_Parallel.}

1: \textbf{for } $s \leftarrow -1$ \textbf{to } 1 \textbf{ step } 2 \textbf{ do}
2: \hphantom{1:} \textbf{loop}
3: \hphantom{1:} if \text{size}_H(s) = 0 \textbf{ then}
4: \hphantom{1:} \hphantom{3:} \textbf{exit loop}
5: \hphantom{1:} end if
6: \hphantom{1:} $(i,j,k) \leftarrow \text{Locate\_Min}(s)$
7: \hphantom{1:} if $|\psi_{i,j,k}| > \text{bound}_{\text{band}}(s)$ \textbf{ then}
8: \hphantom{1:} \hphantom{3:} \textbf{exit loop}
9: \hphantom{1:} end if
10: \hphantom{1:} if $G_{i,j,k} = \text{BAND\_NEW}$ \textbf{ then}
11: \hphantom{1:} \hphantom{3:} $G_{i,j,k} \leftarrow \text{KNOWN\_NEW}$
12: \hphantom{1:} \hphantom{2:} else if $G_{i,j,k} = \text{BAND\_OLD}$ \textbf{ then}
13: \hphantom{1:} \hphantom{3:} $G_{i,j,k} \leftarrow \text{KNOWN\_OLD}$
14: \hphantom{1:} \hphantom{2:} end if
15: \hphantom{1:} \text{Remove\_Min}(s)
16: \hphantom{1:} \text{Update\_Neighbors}(i,j,k,s)
17: \hphantom{1:} \textbf{end loop}
18: \textbf{end for}
point by a comparison of its value with that of point \((l, m, n)\). This is because that, as discussed above, point \((l, m, n)\) could be a KNOWN\_OLD or KNOWN\_NEW point and might carry a function value lower than that of its neighbor with a KNOWN\_OLD or KNOWN\_NEW tag. The parallel procedure for the front propagation within a narrow band is given in Algorithm 12. Here, a BAND point has to be elevated to the corresponding KNOWN tag because of the augmented tag sets. Also \(width_{\text{band}}\) is replaced by \(\text{bound}_{\text{band}}\), which is not a constant specified beforehand any more and will be discussed later. Other than these two small differences, this procedure is almost the same as the sequential version given in Algorithm 7.

Up to this point, it should be evident that the major components of the sequential fast marching method are barely modified in the present parallel method. Actually, in a single-process setting, the augmented tag sets work exactly in the same way as the original BAND and KNOWN tags; and the additional upwind direction checks for two neighboring points are not something unexpected (they are implied by the one-way conversion of BAND to KNOWN status) in the sequential algorithm. This is essential for keeping all the desirable properties of the fast marching method as well as achieving a straightforward parallel implementation based on a sequential algorithm.

4.3. Synchronized data exchanges

The data exchanges at the boundaries of subdomains play a central role in a domain decomposition parallelization. Usually the ghost points for one subdomain are filled with solutions computed at the corresponding physical domain points from a neighboring subdomain. But in a fast marching algorithm, it is very likely that function values are only updated at a portion of physical domain points that coincide the ghost points of a neighboring subdomain. Apparently just this portion of points with updated values is to be conveyed to the neighboring subdomain. For a process \(p\), it may have at most 26 neighbors in a 3D case. The set of processes that are neighbors of \(p\) is labelled as \(\mathcal{N}_p\). For the overlapping domain decomposition approach adopted in the present algorithm, the shared region between process \(p\) and its neighbor \(q\) is \(\Xi_p \cap \Xi_q\). As shown in Algorithm 13, the status of every point in the shared regions, i.e., \(\sum_{q \in \mathcal{N}_p} \Xi_p \cap \Xi_q\), is checked to single out points with updated function values. It should be noted that this step includes both KNOWN\_NEW and BAND\_NEW points. The inclusion of the latter serves the purpose of propagating the latest information away from the upwind direction in a timely manner. The importance of this cannot be over-emphasized for
Algorithm 13 Collect data in the overlapping region:

\[ \text{Collect Overlapping Data.} \]

1: \textbf{for all } \((i, j, k) \in \sum_{q \in \mathcal{N}_p} \Xi_p \cap \Xi_q\) \textbf{ do} \\
2: \quad \textbf{if } G_{i,j,k} = \text{BAND\_NEW} \textbf{ or } G_{i,j,k} = \text{KNOWN\_NEW} \textbf{ then} \\
3: \quad \quad \textbf{if } G_{i,j,k} = \text{BAND\_NEW} \textbf{ then} \\
4: \quad \quad \quad G_{i,j,k} \leftarrow \text{BAND\_OLD} \\
5: \quad \quad \textbf{else} \\
6: \quad \quad \quad G_{i,j,k} \leftarrow \text{KNOWN\_OLD} \\
7: \quad \textbf{end if} \\
8: \quad \textbf{for all } \text{process } q \in \mathcal{N}_p \textbf{ do} \\
9: \quad \quad \textbf{if } \text{point } (i, j, k) \in \Xi_q \textbf{ then} \\
10: \quad \quad \quad \text{Add point } (i, j, k) \text{ to outgoing buffer: } \mathcal{S}_q^p(i, j, k) \leftarrow \psi_{i,j,k} \\
11: \quad \quad \textbf{end if} \\
12: \quad \textbf{end for} \\
13: \textbf{end if} \\
14: \textbf{end for} \\

Algorithm 14 Exchange data in the overlapping region:

\[ \text{Exchange Overlapping Data.} \]

1: \textbf{for all } \text{process } q \in \mathcal{N}_p \textbf{ do} \\
2: \quad \text{Send outgoing buffer } \mathcal{S}_q^p \text{ to } q \\
3: \textbf{end for} \\
4: \textbf{for all } \text{process } q \in \mathcal{N}_p \textbf{ do} \\
5: \quad \text{Receive incoming buffer } \mathcal{R}_q^p \text{ from } q \\
6: \textbf{end for}
the parallelization of the fast marching method as a sequential algorithm in nature. The size of the communicated data may be slightly increased because of it; but the associated penalty in message passing communication overhead should be negligible as explained earlier. After such a point is identified, its tag should be changed from a NEW suffix to a OLD one. This can avoid the inclusion of the same point in the next round of communication, unless its value is renewed again. (Also the reasoning behind the tag conversion from BAND to KNOWN in Algorithm 12 should be apparent at this point.) An updated point may be shared by more than one neighboring processes. Therefore, it is necessary to check against all neighbors and add it to the corresponding outgoing data buffers.

Algorithm 15 Integrate data received from neighboring processes:

**INTEGRATE_OVERLAPPING_DATA.**

1: count$_{\text{refresh}}$ ← 0
2: for all process $q \in \mathcal{M}_p$ do
3:     for all $(l, m, n) \in \mathcal{R}_p^q$ do
4:         $\psi_{\text{new}} \leftarrow \mathcal{R}_p^q(l, m, n)$
5:         if $|\psi_{\text{new}}| < |\psi_{l,m,n}|$ then
6:             if $G_{l,m,n} = \text{KNOWN\_NEW}$ or $T_{l,m,n} = \text{KNOWN\_OLD}$ then
7:                 count$_{\text{refresh}}$ ← count$_{\text{refresh}} + 1$
8:             end if
9:             $s \leftarrow \text{sgn}(\psi_{\text{new}})$
10:            $\psi_{l,m,n} \leftarrow \psi_{\text{new}}$
11:            if $|\psi_{l,m,n}| > \text{bound}_{\text{band}}(s)$ then
12:                $G_{l,m,n} \leftarrow \text{BAND\_OLD}$
13:            else
14:                $G_{l,m,n} \leftarrow \text{KNOWN\_OLD}$
15:            end if
16:            if $(l, m, n) \not\in \mathcal{H}(s)$ then
17:                INSERT\_HEAP($l, m, n, s$)
18:            else
19:                UP\_HEAP($l, m, n, s$)
20:            end if
21:        end if
22:     end for
23: end for
In the present work, only synchronized local communications are used for data exchanges between neighboring processes. It is possible to deliver an updated point (or even only a newly added KNOWN point) to the neighbors immediately after the specific point is identified. And each process constantly checks updates from neighboring processes. Then the parallel algorithm would become quite complicated with very high communication overhead because of many small-size asynchronous data exchanges. With a synchronized data exchange procedure as shown in Algorithm 14 following the data collection procedure, the present parallel algorithm retains the main elements of the sequential fast marching method and, additionally, has the benefit of a simple and straightforward implementation. The inclusion of BAND\_NEW points in the data exchange also further makes the synchronized communications consistent with the present parallelization.

As shown in Algorithm 15, the updated information from the shared regions is incorporated into the current solution field after the incoming data buffers from all neighboring processes are received. This procedure resembles Algorithm 10 for updating the neighboring points of a newly added KNOWN point in several aspects except that the updated value is obtained from a neighboring process instead of from solving the quadratic equation. If the incoming value has a smaller magnitude than that of the local one, the function value at the local point will be replaced by the incoming one. Its tag should be updated consistently according to the tag of incoming point in its residing process as determined in Algorithms 10, 12, and 13. It is worth noting here that an updated local point with a newly assigned KNOWN\_OLD tag will be inserted into the heap (or moved up in the heap if its previous status was BAND) for further treatment. In the spirit of the sequential fast marching method, Algorithm 10 could be directly used instead of the current heap operations. However, updating neighbors here might result in redundant computations as the neighboring points to be computed could also be updated by the incoming data. In addition, it could not utilize the full set of updated upwind points that is only available after the data integration step is completed. In the next part it will become clear that the treatment in the present algorithm only slightly delays the neighbor updating operations. And the seemingly unnecessary heap operations on these KNOWN\_OLD points can actually reduce the amount of neighbor updating operations by keeping a strict order of these points in heap, since all KNOWN\_OLD and KNOWN\_NEW as well as BAND points are to be considered in Algorithm 10. Here a counter count\_refresh is used to determine the number of points that were elevated.
to KNOWN_OLD or KNOWN_NEW status during the updating computations within their own processes, but are refreshed by the incoming data from the neighboring processes. If this counter is not zero after the data integration procedure, obviously the fast marching algorithm has to be carried on to update those points with stale values derived from these previous KNOWN_OLD and KNOWN_NEW points. Therefore, it is necessary to achieve a zero count here for considering the termination of the whole parallel algorithm later.

4.4. Restarted narrow band approach

Algorithm 16 shows the main procedure for the present parallel fast marching method with a novel restarted narrow band approach proposed in this work. Compared with the sequential version given in Algorithm 6, the interface and heap initialization procedures remain the same, but the narrow band marching procedure (the first instance corresponds to that in the sequential version) is placed in a loop for a restarted scheme. Within this loop, the termination criterion is determined first. Two global minimum values (one for each side of the interface) of the BAND points at the tops of the local heaps from all processes are obtained from a synchronized global reduction. Just like the case in the sequential version, these two values have to surpass width_band before the fast marching procedure can be stopped. In addition, the global maximum count of the refreshed KNOWN points during the data integration procedure is also obtained from the same global reduction. A zero value of the global maximum count, together with the former condition, means all KNOWN points in one process won’t be updated by its neighboring processes through data exchanges and their function values can be considered as final.

In this restarted narrow band approach, the restart frequency is determined by the parameter stride: one run-through can further advance the front by an amount of stride. Actually stride is the only free parameter required in the present parallel algorithm. If it takes a zero value, the current parallel algorithm will be running in almost the same sequence as the sequential algorithm. The only exception will be the parallelism that could exist in those processes whose heap top BAND points share the same global minimum function values. On the other hand, with stride = +∞ each process will be running the sequential fast marching algorithm until the heaps are empty in each restart (assume width_band = +∞ for a whole field computation). And the second instance of the parallel narrow band marching algorithm inside the loop can be omitted. However, in a restarted narrow
Algorithm 16 Parallel narrow band fast marching method:

```
PARALLEL_NARROW_BAND_FAST_MARCHING.
1: INITIALIZE_INTERFACE_PARALLEL
2: INITIALIZE_HEAP_PARALLEL
3: loop
4:   for s ← −1 to 1 step 2 do
5:     (i, j, k) ← LOCATE_MIN(s)
6:     minval_{local}(s) ← |ψ_{i,j,k}|
7:   end for
8:   minval_{global} ← ALLREDUCE_{MIN}(minval_{local})
9:   count_{global} ← ALLREDUCE_{MAX}(count_{refresh})
10:   if minval_{global} (±1) > width_{band} and count_{global} = 0 then
11:     exit loop
12:   end if
13:   bound_{band} (±1) ← min(minval_{global} (±1) + stride, width_{band})
14:   MARCH_NARROW_BAND_PARALLEL
15:   COLLECT_OVERLAPPING_DATA
16:   EXCHANGE_OVERLAPPING_DATA
17:   INTEGRATE_OVERLAPPING_DATA
18:   MARCH_NARROW_BAND_PARALLEL
19: end loop
```
band framework with a proper stride, the second marching step brings the local fronts residing in different processes to the same bound. This is essential for a restarted scheme with a synchronized global upper bound for each run-through. Without this step, the neighbor updating procedure will be required for each point that is updated by a neighboring process or an updated neighboring point and consequently receives a KNOWN tag in Algorithm 15. As discussed in the previous part, the resulting algorithm will be less efficient and far more complicated. It is also worth noting that the augmented tags proposed in this work facilitates the restarted narrow band approach with minimized data communications and makes the overall algorithm quite compact.

5. Results

5.1. Test cases

Although the Eikonal equation is encountered in many fields, here we are mainly concerned about obtaining a signed distance function in computational fluid dynamics applications, such as the reinitialization of the level set function in two-phase flow simulations and the calculation of the wall distance function in turbulence modeling. For this purpose, the speed function $F$ is set to be unity in the present study. However, the algorithm can be applied to problems with variable speed functions in a straightforward manner.

For simplicity, a unit cube $[-0.5, 0.5] \times [-0.5, 0.5] \times [-0.5, 0.5]$ was used as the computational domain. As shown in Fig. 8, three test cases were performed. In the first case, a sphere of radius 0.25 was centered in the domain. This case was used by Herrmann in his study [2]. In the second case, eight spheres of radius 0.23 were evenly placed in the eight octants of the domain. The centers of these spheres are $(\pm 0.25, \pm 0.25, \pm 0.25)$. In the third case, some offsets were added to the sphere centers in the second case, such that the centers of the eight spheres are $(-0.24, -0.26, -0.24), (0.24, -0.26, -0.26), (-0.24, 0.24, -0.26), (0.24, 0.24, -0.24), (-0.26, -0.24, 0.26), (0.26, -0.24, 0.24), (-0.26, 0.26, 0.24)$, and $(0.26, 0.26, 0.26)$. In these cases, the distance function inside a sphere is negative and no boundary conditions are required as the interfaces are totally embedded in the domain. In the numerical implementation, the interfaces were initialized by assigning analytical function values to the two sets of grid points immediately adjacent to the interface from the positive and negative sides, respectively. Six uniform grids were
used in the computations and the total numbers of grid points (excluding the ghost points) were $N = nh^3 = 32^3, 64^3, 128^3, 256^3, 512^3, \text{ and } 1024^3$ (more than 1 billion).

Figure 8: Test cases: a) single sphere; b) eight spheres of uniform distribution; and c) eight spheres of non-uniform distribution.

All computations were performed on Garnet, a Cray XE6 supercomputer located at the U.S. Army Engineer Research and Development Center (ERDC) in Vicksburg, Mississippi, one of the five U.S. Department of Defense (DoD) Supercomputing Resource Centers (DSRCs) that are operated by the U.S. DoD High Performance Computing Modernization Program (HPCMP). Garnet has 150912 compute cores (4716 compute nodes each with 32 cores) and is rated at 1.5 peak PFLOPS. The compute nodes are populated by 2.5 GHz AMD Interlagos Opteron (6200 series) processors with two processors per node, each with sixteen cores. Each node contains 64 GBytes of DDR3 memory shared by the 32 cores. Computer nodes are connected by the Cray Gemini Interconnect network. Garnet supports different parallel programming models. In this study, the algorithm was implemented in Fortran 2003 with the Message Passing Interface (MPI) for communications among processes. The MPI library on Garnet derives from the Argonne National Laboratory MPICH, which implements the MPI-3.0 standard. The code was compiled in double precision using the Intel Fortran Compiler XE version 14.0.2.144 with the -fast optimization level. The code performance is affected by many different factors, especially the network throughput, since the system is shared by many users. In the present study, all computations were repeated five times and the CPU times reported here are the averaged values.
5.2. Single-process runs

Both the sequential and parallel algorithms were first run with a single process. \texttt{stride} = 2\Delta h was used with the parallel algorithm. To show that
the present implementations achieve the expected first-order accuracy of a first-order fast marching method, a grid convergence study was carried out. The error at a grid point is defined as the difference between the computed and the analytical function values. The $L_2$ error norm is defined for the whole computational domain as

$$L_2 = \sqrt{\sum_{i,j,k} \left( \psi^{\text{exact}}_{i,j,k} - \psi^{\text{computed}}_{i,j,k} \right)^2 \over N - N_0},$$

where $N_0$ is the total number of $\text{FAR}$, $\text{BAND}$, and $\text{KNOWN\_FIX}$ points, which are not included in the error norm calculation (in the present whole field computations, $\text{FAR}$ or $\text{BAND}$ point does not exist at the end). Fig. 9(a) shows the $L_2$ error norm as a function of the grid size $nh$. The first-order accuracy of the present algorithms is evident. The first case gives higher errors because the interface information propagates longer distances than those in the other two cases. The error norm results from the parallel algorithm are exactly the same as those from the sequential one and they are not labelled separately in the figure.

Fig. 9(b) shows the CPU time as a function of $nh$. Although there are four interfaces in the second and third cases, the total CPU times are not very different from that for the first case. For the fast marching algorithm, the $O(\log N)$ in its theoretical algorithm complexity, $O(N \log N)$, comes from the worst-case scenario of reordering of a heap of length $N$. In actual applications, the heap lengths are usually much smaller than $N$. In the figure, a nonlinear curve fitting given by $a + b N \log_2(c N)$ with $a = 0.05$, $b = 4.5 \times 10^{-7}$, and $c = 1.5 \times 10^{-5}$. On one hand, this fitting verifies the correctness of the present implementation of the fast marching algorithm. On the other hand, with such a small constant $c$ it also demonstrates the efficiency of the fast marching algorithm as a single-pass approach. Interestingly, a different fitting of $d + e N^{1.2}$, with $d = 0.01$ and $e = 1.2 \times 10^{-7}$, matches the computational results slightly better than the $N \log(N)$ one. This shows for the current test cases that the algorithm complexity of the fast marching method is only slightly higher than linear.

The number of restarts, i.e., $nr$, is a major parameter in the present restarted narrow band approach for the parallel algorithm. Fig. 9(c) shows a linear relationship between $nr$ and the grid size $nh$ for all three test cases. For the second and the third cases, the results are almost the same. As mentioned above, the interface information propagates longs distances in the first case,
correspondingly, \( nr \) is higher than those in other two cases because of the same size of stride. Fig. 9(d) shows the parallelization overhead as a function of the grid size \( nh \) for all three cases. Parallelization overhead is the relative difference between the CPU time of the sequential algorithm \( T_S \) and that of the parallel algorithm running with a single process, \( T_1 \), i.e., \( \frac{T_1 - T_S}{T_S} \times 100\% \). It measures the extra work required to parallelize a sequential algorithm. For the present parallel algorithm, the sequential marching algorithm has only been slightly modified. Apparently, the introduction of the restarting loop has the major impact on the overhead. Besides a second marching step, this loop includes several new modules: global data reduction, and collection, local exchange, and integration of the data from the overlapping regions. In the single-process runs of the parallel algorithm, these modules incur additional computational cost, although data communication does not occur at all. In general, as the grid size increases, the parallelization overhead decreases due to the decreasing ratios of the overhead from these additional modules and the cost of the first marching step. For the present test cases, the parallelization overheads are generally insignificant and well below 4\% on finer grids. Case three gives higher overheads than the other two cases, one possible reason is that the special interface configuration trends to trigger much more data collection operations at the domain boundaries.

5.3. Parallel speedup and efficiency

A systematical study of the parallel performance of the present parallel fast marching algorithm was carried out with \( \text{stride} = 2\Delta h \). As mentioned above, the three cases were executed on six different grids ranging from less than 40 \( k \) to more than 1 billion points using different number of processes ranging from \( np = 1 \) to 65536. The total number of domain decomposition configurations is 17, i.e., \( p_i \times p_j \times p_k = 1 \times 1 \times 1 (np = 1), 1 \times 1 \times 2 (np = 2), 1 \times 2 \times 2 (np = 4), 2 \times 2 \times 2 (np = 8), 2 \times 2 \times 4 (np = 16), 2 \times 4 \times 4 (np = 32), 4 \times 4 \times 4 (np = 64), 4 \times 4 \times 8 (np = 128), 4 \times 8 \times 8 (np = 256), 8 \times 8 \times 8 (np = 512), 8 \times 8 \times 16 (np = 1024), 8 \times 16 \times 16 (np = 2048), 16 \times 16 \times 16 (np = 4096), 16 \times 16 \times 32 (np = 8192), 16 \times 32 \times 32 (np = 16384), 32 \times 32 \times 32 (np = 32768), \) and \( 32 \times 32 \times 64 (np = 65536) \).

The overall results are shown in Fig. 10, which is organized such that rows present different performance parameters or metrics and columns give different test cases. The first row gives the number of restarts \( nr \) as a function of the number of processes \( np \) and the grid size \( nh \). One finer grids (\( nh \geq 256 \)), \( nr \) keeps the same or only increases by one as \( np \) increases. This implies
Figure 10: Parallel fast marching algorithm: the number of restarts $nr$, CPU time, parallel speedup, and parallel efficiency as functions of number of processes $np$ and grid size $nh$. 

35
some overheads such as loop structures and function/subroutine calls remain
more or less the same no matter how many processes are used. But for coarser
grids ($nh \leq 128$), as $np$ increases the total number of points for each process
decreases; the process exhausts its heaps (both negative and positive sides)
befor $\text{bound}_{\text{band}}$ could be reached. This causes $nr$ eventually doubled or
even tripled after the block size assigned to each process reduces to less than
$4 \times 4 \times 4$. Of course, grid blocks of this tiny size are not efficient and should not
be adopted in actual applications. $nr$ does not change much when the grid
block assigned to each process is of a reasonable size. Its variation along with
the grid size $nh$ shows a linear relationship and has already been reported in
Fig. 9(c). This verifies that the number of restarts is totally different from
the number of iterations in iterative algorithms and the present restarted
narrow band approach retains the single-pass, non-iterative property of the
sequential fast marching method.

The second row of Fig. 10 shows the CPU time as a function of the
number of processes $np$ and the grid size $nh$. On coarser grids ($nh \leq 64$),
as $np$ increases the overhead due to the increased number of restarts takes
over and the total CPU time begins to increase with a similar pattern as
that of $nr$ shown in the first row of the figure. On medium grids ($nh = 256$
and $nh = 128$), a grid block of size around $16 \times 16 \times 16$ appears to the
threshold for achieving a positive gain from the increased $np$. Nevertheless,
a sustained trend of decreasing CPU time as $np$ increases is observed on finer
grids ($nh \geq 512$).

The discussions above mainly focus on the fixed problem analysis, which
shows the strong scalability of the parallel algorithm. Since problems of
different sizes are also studied, it is interesting to check the weak scalability,
i.e., each process computes the same number of grid points. In the first and
second rows of Fig. 10, each dashed line gives the number of restarts and
the CPU time for a grid block of a constant size, respectively. Let $nb$ be the
number of blocks in each grid direction, it is evident that $np = nb^3$ for this
study. In general, when the number of grid blocks $nb$ doubles, $nr$ also doubles.
It should be noted that for the Eikonal problems considered here, it is not
expected the number of restarts or the CPU time will remain the same as the
global grid size increases (but with a grid block of a constant size for each
process) even under the ideal circumstance of negligible communication and
other overheads. That is, for the same problem but on a refined global grid
that has doubled number of points in each direction, in the computational
space the information from the interface has to propagate through a doubled
distance to reach the same location in the physical space. The variation of the CPU time along with increasing \( nb \) largely depends on the interface properties of the problem. For the present test cases, from \( nb = 1 \) to \( nb = 2 \), the CPU time only slightly increases; and the CPU time even decreases when \( nb \) increases from 2 to 4 for the second and third cases with a grid block of size between \( 128^3 \) and \( 256^3 \) (for the second case with the \( 64^3 \) block too). This comes from the combined effects of an increased but still balanced computational load and reduced heap sizes on each process. For general scenarios (\( nb > 2 \) for case 1, and \( nb > 4 \) for cases 2 and 3), the CPU time increases following a slope of around \( nb^{1.3} \). Load imbalance is the major reason for this super-linear behavior.

The third row of Fig. 10 shows the parallel speedup of the present parallel fast marching algorithm for the three test cases. Here the relative speedup defined as \( S = \frac{T_1}{T_{np}} \) is used with \( T_1 \) and \( T_{np} \) the CPU times of the parallel algorithm on one process and \( np \) processes, respectively. The absolute speedup defined using \( S_S \) instead of \( T_1 \) will be slightly lower than the relative one, as demonstrated by the low parallelization overheads for grid sizes of practical interest. For the first case, the first three domain decomposition configurations, i.e., \( 1 \times 1 \times 2 \), \( 1 \times 2 \times 2 \), and \( 2 \times 2 \times 2 \), give symmetrical decompositions and thus optimal balanced computational loads. Therefore, the algorithm complexity for each process without considering communication and other overheads will be \( O(\frac{N}{np} \log \frac{N}{np}) = O(\frac{N}{np} \log N - \frac{N}{np} \log np) \), which explains the super-linear speedup behavior at \( np = 4 \) and 8 for the finest two grids. For the second case, the domain decomposition remains symmetrical with optimal load balance up to \( np = 64 \) (\( p_i \times p_j \times p_k = 4 \times 4 \times 4 \)); whereas for the third case, the domain decomposition is always non-symmetrical due to the special arrangement of the eight spheres, although an optimal load balance can be achieved up to \( np = 8 \). For these two cases, sustained super-linear speedup is observed up to \( np = 2048 \) for the grid \( nh = 1024 \). The threshold of a \( 16^3 \) block per process for retaining an overall positive gain from increasing \( np \) is also highlighted in the plot for each case, although it can be even lower for coarser grids due to smaller counts of restarts.

The relative parallel efficiency, which is defined as \( E = \frac{T_1}{np T_{np}} \), is shown in the fourth row of Fig. 10. The super-linear speedup behavior discussed above is more evident here showing up as data above the \( E = 1 \) line. The performance highly depends on the interface properties of each problem. It is difficult to obtain a general grid block size per process for guaranteeing a
certain given parallel efficiency for all cases.

In order to understand the parallel performance shown in Fig. 10, the CPU time spent by each major module of the present parallel fast marching algorithm has been recorded for each restart: $t_r$ for the global reduction and termination check (lines 4 to 13 in Algorithm 16); $t_{m1}$ for the first marching step; $t_c$ for the collection of updated data in the overlapping regions; $t_e$ for the local data exchange between neighboring processes; $t_i$ for the integration of new data received from neighboring processes; and $t_{m2}$ for the second marching step. Because of the two synchronization steps in each restart, i.e., the global data reduction and local exchange of updated data in the overlapping regions, processes with lower loads have to be in an idle status. And the processes with longer CPU times are very likely to have more updated data in the overlapping regions to send to neighbors. Therefore, computation and communication are hardly overlapped, although non-blocking communication mode is used for the local data exchanges. For each module, the difference between the maximum and the minimum CPU times measures the load imbalance during one restart. And the difference between their sums in the whole course of computation is a good indicator the overall load imbalance. Of course, this does not apply to the data communications, since the idle time due to imbalanced computational loads and message sizes cannot be simply separated from the communication time. However, the sums of the minimum CPU times for the data communication modules can still give an idea of the overall trend of communication cost with varying $np$.

Figure 11 shows the load imbalance as a function of the number of processes $np$ on the finest grid $nh = 1024$ for the four computational modules in the present parallel fast marching method. Again, columns give different test cases and rows present different modules. The total CPU time for each case is also given in the sub-figures to illustrate the cost of each module in proportion. The first row shows $\sum_{ir} \max(t_{m1})$ and $\sum_{ir} \min(t_{m1})$ with $ir$ the counting index of the restarts and the min/max functions are global operations over all $np$ processes during each restart. It is clear that the first marching step, as represented by $\sum_{ir} \max(t_{m1})$, accounts for most of the total CPU time. This also verifies that the parallel algorithm does not alter the essential characteristics of the sequential fast marching algorithm. For the first case, a very good load balance is observed for $np$ up to 8, as expected from the interface configuration. Then starting from $np = 128$, some processes became completely idle. For the second case, an excellent load balance is evident for $np$ up to 64. Then completely idle processes appeared after
Figure 11: Load imbalance as a function of $np$ on grid $nh = 1024$ for the four modules in the parallel fast marching algorithm.
\( np \geq 2048 \). The third case shows a slightly lower level of load balance than that of the second case due to its non-symmetrical interface configuration. On the other hand, these data also demonstrate that it is not required to have a (nearly) perfect balanced computational load for achieving a good parallel efficiency for the present parallel fast marching algorithm, thanks to the much reduced cost of resorting much shortened heaps.

The second row of Fig. 11 gives the load imbalance in the collection of updated data in the overlapping regions. It should be noted that, for simplicity, the domain boundaries were treated like subdomain interfaces here as if there were neighboring processes sharing these boundaries. \( \sum_{ir} \min(t_c) \) largely represents the fixed cost of checking all grid points for updated values in the overlapping regions. And \( \sum_{ir} \min(t_c) \) decreases following nearly a constant slope as \( np \) increases, since the overlapping regions shrink with increasing \( np \) and for this grid \( nr \) barely changes with \( np \). The difference between \( \sum_{ir} \min(t_c) \) and \( \sum_{ir} \max(t_c) \) indicates the cost of storing the updated data in the overlapping regions, especially, when some processes are completely idle during the first marching step. As \( np \) increases, the total count of grid points in the overlapping regions increases, and so does the number of points that have been updated. This is the reason that the difference between \( \sum_{ir} \min(t_c) \) and \( \sum_{ir} \max(t_c) \) increases, and eventually \( \sum_{ir} \max(t_c) \) increases along with \( np \).

The collected data are exchanged between neighboring processes, and each process integrates the received data into its own heaps. The load imbalance in this integration procedure is shown in the third row of Fig. 11, which follows a pattern similar to that of the first marching step in terms of the relative scale between \( \sum_{ir} \min(t_i) \) and \( \sum_{ir} \max(t_i) \). Overall, \( \sum_{ir} \max(t_i) \) only slightly decreases as \( np \) increases; and, just like \( \sum_{ir} \max(t_c) \), it becomes a significant portion in the total CPU time for a very large \( np \). As \( np \) increases, apparently, there will be more grid points in the overlapping regions and these points, once updated, will be redundantly stored by up to three neighboring processes. The additional cost required for dealing with these data largely offsets the cost saving due to shortened heaps in a smaller grid block.

As shown in the fourth row of Fig. 11, the second marching step only accounts for a small portion of the total CPU time for most of the domain decompositions. Like the data integration module, its share becomes significant only after the grid block size per process reduces to less than about 64\(^3\). In the present restarted narrow band approach, the interface information only
propagates a limited distance in one restart. This feature greatly confines the load imbalance in the second marching step and guarantees the overall parallel performance. For the first case, the load imbalance prevails among all \( np \), which illustrates the largely unilateral propagation of information due to the interface configuration. For the third case, however, relative mild load imbalance is observed up to \( np = 512 \), which comes from the more frequent bilateral data exchanges between subdomains due to the non-symmetrical arrangement of the eight spheres in the interface initialization. This also explains the abrupt raise of \( \sum_{ir} \min(t_m) \) for \( np = 128 \) to 512 in the second case, since the domain decomposition is no longer balanced and symmetrical after \( np > 64 \).

Figure 12: Minimum CPU time as a function of \( np \) on grid \( nh = 1024 \) for data communications in the parallel fast marching algorithm: local data exchanges and global data reduction.

Compared with the load imbalance in each computational module, the cost of data communications seems to be minor as shown in Fig. 12. As \( np \) increases, \( \sum_{ir} \min(t_e) \) slightly decreases, which indicates that less data are involved in the local data exchanges among neighboring processes; whereas \( \sum_{ir} \min(t_r) \) shows an opposite trend, because of the increased processes in the global reduction operation. The amount of updated data in the overlapping regions may vary from two layers of grid points to zero. Instead of exchanging the full overlapping regions, the set of augmented tags in the present parallel
algorithm is designed to minimize data sizes involved in local data exchanges. On the other hand, the global data reduction only involves three elements as shown in Algorithm 16. Therefore, the data communication cost is not substantial in the present parallel algorithm. Of course, the computations were performed on a supercomputer equipped with the Gemini interconnect, which is well known for its extremely low latency and super-high network bandwidth. This also contributes to the low communication cost indicated here.

5.4. Effect of memory bandwidths

![Graphs showing CPU time, parallel speedup, and parallel efficiency as functions of the number of processes np on grid nh = 1024 with different memory bandwidths.]

Figure 13: Comparison of the CPU time, parallel speedup, and parallel efficiency as functions of the number of processes np on grid nh = 1024 with different memory bandwidths.

For the supercomputer used in this study, the 32 CPU cores on each compute node are organized into compute units of core pairs. The 16 compute units are divided into four NUMAs (non-uniform memory access). There are 4 compute units, 8 MBytes L3 cache and 16 GBytes NUMA memory on each NUMA. A compute node consists of two processors and each processor contains two NUMAs. All four NUMAs are connected so each core can access other NUMA memory besides its own. A compute node has a memory hierarchy of six levels with increasing data capacity and decreasing access time: 16 KBytes L1 cache accessible to each core, 2 MBytes L2 cache shared by each core pair, 8 MBytes L3 cache shared by 8 cores, 16 GBytes NUMA memory shared by the same 8 cores, 16 GBytes NUMA memory on the other NUMA within the same processor, and 32 GBytes NUMA memory on the other processor. It should be noted here that a portion of the L3 cache and NUMA memory is reserved and not accessible to the user. Because of the
limited data capacity of cache memory, data have to move up through the memory hierarchy, even be transferred from a different compute node using MPI (slower) or read from the disk (much slower) first, to reach a core’s L1 cache for any computation to be performed on them. And the computation results have to move down through the memory hierarchy so the L1 cache can be used for the next set of data. The core will be in an idle status before the required data reach its L1 cache. Therefore, frequent data movements in the memory hierarchy can badly affect the performance.

One way to improve cache performance is to reduce the number of cores sharing the L2 and L3 cache memory. In the previous section, all tests were performed with the full computational capacity of each compute node, unless the number of processes $np$ was smaller than the number of cores available on one compute node. This is the regular way of CPU core utilization, in which all CPU cores share the memory bandwidth available to the compute node. In order to examine the effect of different memory bandwidths on the parallel performance, a set of tests was carried out using 2048 compute nodes, which contain a total of 65536 cores. For $np = 2$ to 2048 only a single process was assigned to one compute node. Thus all the accessible memory bandwidth was available to a single core instead of shared by 32 cores. For $np = 4096, 8192, 16384, \text{and } 32768$, two, four, eight, and sixteen processes were assigned to each compute node, respectively. For all the tests, just like those with $np < 32$ in the previous section, no effort was made for optimizing core affinity, i.e., assignment of processes to designated cores, to possibly improve the parallel performance.

Figure 13 shows the comparison of the CPU time, parallel speedup and efficiency as functions of $np$ on grid $nh = 1024$ between the current tests (multiplied memory bandwidth) and the tests from the previous section (regular). Substantial improvements of parallel performance were achieved for $np$ up to 4096. In the previous section, the results of case 1 does not demonstrate a super-linear speedup at all. But with the multiplied memory bandwidths here, case 1 shows a similar high parallel efficiency close to 2 at $np = 4$ as the other two cases. And this super-linear speedup sustains up to $np = 64$. For cases 2 and 3, the maximum parallel efficiencies are more than 3 for $np = 128$. Especially, the super-linear speedup of case 2 sustains up to $np = 4096$, which is highly remarkable considering the increasing load imbalance after $np = 64$ for this case. For $np = 8192$ with 4 processes per compute node, the results are moderately improved. But for $np = 16384$ and 32768 with 8 and 16 processes per node respectively, the parallel efficiencies of cases 2 and 3 become
slightly lower than those obtained from the regular memory bandwidth. Apparently, in addition to the fact that the increase of memory bandwidths was not significant at all, the block sizes per process were small enough to make the algorithm less memory-hungry. On the other hand, the communication cost will be higher due to the involvement of more compute nodes.

5.5. Effect of domain decomposition strategies

Figure 14: Comparison of the CPU time, parallel speedup, and parallel efficiency as functions of the number of processes $np$ on grid $nh = 1024$ between the 2D and 3D domain decomposition configurations.

For 3D problems, there are three possible domain decompositions: 1D decomposition that partitions the 3D grid only in one grid direction, either $i$, $j$, or $k$; 2D decomposition that partitions the 3D grid in two grid directions, either $i$ and $j$, $j$ and $k$, or $k$ and $i$; and 3D decomposition that partitions the 3D grid in all three grid directions. Apparently, the total number of processes is severely limited in a 1D decomposition, i.e., $np \leq nh$, which make the 1D decomposition a sub-optimal choice in most applications. On the other hand, a 3D decomposition is very flexible in terms of the range and the maximum number of applicable processes. For some applications, however, a 2D decomposition can be very useful since highly efficient sequential algorithms such as FFT (fast Fourier transform) or TDMA (Tridiagonal matrix algorithm) can be utilized in the non-decomposed direction and may result in a better overall performance. Therefore, 2D decompositions are tested and the results are compared with those from the 3D decompositions in this part to show the effect of domain decomposition strategies.
In addition to the $p_i \times p_j \times p_k = 1 \times 2 \times 2$ decomposition that has already been shown above, seven 2D decompositions, i.e., $p_i \times p_j \times p_k = 1 \times 4 \times 4$ ($np = 16$), $1 \times 8 \times 8$ ($np = 64$), $1 \times 16 \times 16$ ($np = 256$), $1 \times 32 \times 32$ ($np = 1024$), $1 \times 64 \times 64$ ($np = 4096$), $1 \times 128 \times 128$ ($np = 16384$), and $1 \times 256 \times 256$ ($np = 65536$), were tested. Clearly, unlike the 3D decompositions, 2D decompositions is only applicable to grids with $nh \geq p_j$ ($p_i = 1$ and $p_k = p_j$ here). One advantage of the 2D decompositions is that the number of neighboring subdomains (and the number of data communications between neighbors correspondingly) is 8 for an inner subdomain instead of 26 in the 3D decompositions. Nonetheless, the surface area of a subdomain is always larger than its counterpart in a 3D decomposition with the present domain decomposition configurations.

Figure 14 shows the comparison of the CPU time, parallel speedup, and parallel efficiency as functions of $np$ on grid $nh = 1024$ between the 2D and 3D domain decomposition configurations. There are several noticeable features in this comparison. For instance, for cases 2 and 3, the super-linearity of the parallel speedup at $np = 64$, 256, and 1024 in the 2D decompositions are far less striking than that in their 3D counterparts, as evident by the obvious differences in their the parallel efficiencies. In addition, the performance of the 2D decompositions in case 1 is better than that of the 3D decompositions for the four runs from $np = 64$ to $np = 16384$. But the performance of the 2D decomposition at $np = 32768$ is lower than that of the 3D decomposition for all three cases. Interestingly, the 2D decomposition configurations seem less sensitive to the source (interface) properties within the domain, as shown by the highly similar variations of the CPU time and speedup for all three cases. The overall performance from the 2D domain decompositions is fairly close to that from the 3D decompositions. This evidences that the 2D domain decomposition configuration is a viable choice for the present parallel fast marching method. It also demonstrates the applicability of the present approach in a wide range of practical problems, in many of them the Eikonal equations are just components of much large systems and sometimes the major portions of the systems might be tackled more efficiently with the 2D domain decomposition configuration.

5.6. Effect of stride sizes

2D or 3D domain decomposition configurations are typical options that need to be determined by the user in many parallel applications. Therefore, they are not some special variables merely required here. On the other hand,
Figure 15: Comparison of the number of restarts $nr$, CPU time, and relative change in the CPU time as functions of the number of processes $np$ on grid $nh = 1024$ between different stride sizes.
just like the narrow band width in the sequential fast marching method, the stride size in the restarted narrow band approach is the only essential free parameter in the present parallel fast marching algorithm. A constant value \( \text{stride} = 2 \Delta h \) was used in all of the parallel runs above. This value was simply chosen according to the unity speed function and the thickness of the overlapping regions. But this does not mean that \( \text{stride} = 2 \Delta h \) is always the optimal choice and the performance of the present parallel algorithm becomes less satisfactory for any other values. Apparently, the optimal stride size is affected by many factors including the interface (source) properties, domain decomposition configurations, grid sizes, and computing platforms. Nevertheless, the efficiency of a versatile algorithm should not be dramatically affected by a sensible choice of a free parameter.

A parametric study was conducted to illustrate the effect of different stride sizes on the performance of the present parallel algorithm. The 3D domain decompositions used in the previous sections were adopted for the tests of all three cases on the finest grid \((nh = 1024)\). Besides \( \text{stride} = 2 \Delta h \), a series of stride sizes from \( \text{stride} = 0.5 \Delta h \) to \( 3.5 \Delta h \) with an increment of \( \Delta h \) were examined. A special case was also investigated, in which \( \text{stride} \) was set to \(+\infty\) (a huge positive value in the actual implementation) such that only the termination condition \( \text{size}_0(s) = 0 \) in Algorithm 12 was to be met. In this case, therefore, all points will be computed once the computation is triggered. This seems to be a resemblance to many iterative algorithms for the Eikonal equation. But a salient feature distinguishes this case from those iterative algorithm: the completion of computation does not rely on a convergence check at all.

Figure 15 shows the comparison of parallel performance between different stride sizes on the finest grid \((nh = 1024)\). The first row presents the number of restarts \( nr \) as a function of the number of processes \( np \). In general, \( nr \) remains as a constant for a limited stride size among \( 0.5 \Delta h \) to \( 3.5 \Delta h \). In addition, \( nr \) decreases linearly as the stride size increases. On the other hand, \( nr = 1 \) at \( np = 1 \) for all three cases since the parallel algorithm becomes equivalent to the sequential version (as expected). \( nr \) remains at two for case 2 up to \( np = 64 \). Under these circumstances, apparently, each subdomain can be updated independently because of the symmetrical interface configuration. Interestingly, \( nr = 2 \) sustains up to \( np = 16 \) for case 1, although the optimal decompositions only sustain up to \( np = 8 \). For case 3 \( nr \) stays at six for several decompositions up to \( np = 64 \), although the interface configuration is not symmetrical at all. It is worth noting that for all three cases \( nr \)
grows rapidly as \( np \) further increases and eventually reaches the same order of magnitude as that of \( \text{stride} = 3.5\Delta h \).

The second row of Fig. 15 presents the CPU times as functions of \( np \) at different stride sizes, and the relative changes of the CPU times with regard to that of \( \text{stride} = 2\Delta h \) are shown in the third row. For the CPU times, it is difficult to distinguish the results of different strides sizes when a domain decomposition gives an optimal load balance, which is the scenario in case 1 for \( np \) up to 8, in case 2 for \( np \) up to 64, and in case 3 for \( np \) up to 8. It is interesting to note that generally \( \text{stride} = +\infty \) gives a slightly better performance for these decompositions with an optimal load balance in cases 1 and 2, as evident by the negative relative changes from it in the figure. Of course, usually it is not a good choice in practical applications the computational cost could be several times higher than the benchmark.

On the other hand, as \( np \) increases, the grid block size per process becomes smaller and eventually the computational cost with \( \text{stride} = +\infty \) decreases to the same order of magnitude as those from a stride size around \( 2\Delta h \). This is quite reasonable for a large \( np \) because of the relative scale variations between the computation and communication costs. For instance, at \( np = 32768 \) the subdomain grid size is \( 32^3 \) for grid \( nh = 1024 \): with such a small computational load the communication cost becomes a substantial part; and with \( \text{stride} = +\infty \) the computation takes less number of restarts although each restart covers more grid points.

The other extreme case is \( \text{stride} = 0 \), in which the parallel algorithm runs exactly like the sequential version without activating the second marching step at all. This scenario was not tried in the present study due to the astronomic number of restarts required (\( \sim N \)). Instead, \( \text{stride} = 0.5\Delta h \) was tested to give a hint on this extreme scenario. Surprisingly, it performed fairly well for most decompositions with penalties of a few percent of the benchmark. Obviously the small stride size greatly reduces the CPU time spent on the second marching step, although the communication cost is substantially increased. As the opposite of the case at \( \text{stride} = +\infty \), with \( \text{stride} = 0.5\Delta h \) the computational cost dramatically increases as \( np \) changes from \( np = 16384 \) to 32768 and then 65536. Apparently, after the subdomain size becomes very small, the CPU time saved from a swift second marching step cannot offset the increased communication costs any more.

Stride sizes close to \( 2\Delta h \), such as \( 1.5\Delta h \) and \( 2.5\Delta h \), give results very comparable with or even better than those at \( \text{stride} = 2\Delta h \). As discussed above, there are many factors that affect the parallel performance. The
results in this part verify that the present parallel algorithm is not exceedingly sensitive to the choice of the stride size. In general, on supercomputers with very fast interconnect networks, a stride size reasonably larger than $2\Delta h$ can give a good performance for a large $np$ at which the subdomain size becomes rather small. Of course, it is always a worthwhile practice to experiment several different stride sizes to decide a good choice for new applications and/or new computing platforms.

6. Conclusions

A highly scalable massively parallel fast marching method has been developed in this paper. A domain decomposition technique is adopted for achieving an efficient parallel algorithm capable of tackling large-scale practical applications using billions of grid points and hundreds of thousands of processes. A novel restarted narrow band approach that profoundly resembles the sequential narrow band fast marching method has been established. The fronts are advanced using essentially the sequential algorithm by a specified stride in each restart until reaching the global narrow band width or no more points to be computed. Within each restart, simple synchronous local exchanges and global reductions are adopted for communicating updated data in the overlapping regions between neighboring subdomains and getting the latest front status, respectively. The restarted narrow band approach balances the cost associated with the number of restarts, i.e., the local data collection, communication, and integration as well as the global data reduction, and the cost of the fast marching computations extra to a sequential run. It greatly mitigates the adverse effects of the spatial-temporal load imbalance on the parallel performance. On the other hand, the independence of front characteristics is exploited to extract the masked parallelism within the fast marching method. First, special data structures are designed to advance both the positive and negative fronts concurrently in two-sided interface problems. In addition, for a newly accepted point received from a neighboring subdomain, grid points with larger function values will be refreshed only if they are influenced by its characteristics. This represents a great saving of computational cost compared with the rollback mechanism and is enabled by the augmented status tags introduced in this study. These tags are incorporated into the sequential fast marching algorithm with surprisingly few modifications. Detailed pseudo codes for both the sequential and parallel algorithms have been provided to illustrate the simplicity of im-
plementation and the similarity to the sequential narrow band fast marching algorithm.

Three examples with different interface configurations have been tested to demonstrate the efficiency, flexibility, and applicability of the present parallel algorithm. These problems are extensively tested on six uniform grids ranging from $32^3$ to $1024^3$ points using different numbers of processes ranging from 1 to 65536. The first-order accuracy and theoretical complexity of $O(N \log N)$ of the fast marching method have been verified through single-process runs on all grids with both the sequential and parallel algorithms. The parallel algorithm is slightly more expensive than the sequential version due to a parallelization overhead of a few percent. This overhead results from the restarted narrow band approach as extra operations on the data in the overlapping regions are required within each restart in the parallel algorithm.

A systematical performance study has been carried out on different grids using different 3D domain decompositions. It has been verified that the number of restarts has generally a linear relationship with the grid size in one direction and is totally different from the number of iterations in iterative algorithms. For computations on finer grids, sustained super-linear parallel speedups have been obtained using thousands of processes and remarkable parallel efficiencies are achieved using tens of thousands of processes. The overhead and load imbalance of each module on the finest grid have been analyzed to provide a better understanding of the performance data.

The effects of memory bandwidths, domain decomposition configurations, and stride sizes have been carefully studied. With a reasonable computational load, the parallel performance can be greatly enhanced by multiplied memory bandwidths and 2D domain decompositions can be viable choices as 3D decompositions. A stride size of $2\Delta h$ is suggested as a rule of thumb according to the unity speed function and the width of the overlapping regions. But other stride sizes in its neighborhood can also perform similarly well or slightly better for some cases.

In terms of future work, apparently, our parallel algorithm can be implemented for higher-order fast marching methods with the present restarted narrow band approach in a straightforward manner. Also, the extensions on unstructured meshes are possible as evident by the few modifications in the sequential algorithm on Cartesian grids. In the present work, one layer of ghost points for a first-order scheme results in overlapping regions of a two-cell width. It is of interest to study the effects of increased widths of overlapping regions on the parallel performance. On the other hand, a unity
speed function is very useful for applications relying on a distance function, but it is of great importance to test variable speed functions for applying the present parallel algorithm to many other fields with every confidence. Moreover, performance data on different supercomputers are very valuable for a deeper understanding and further improvements. Lastly, it is possible to enhance the present parallel algorithm with hybrid programming modes, e.g., using domain decompositions among compute nodes (distributed memory) and multiple threads within a compute node (shared memory), for solving problems of even larger scales and achieving even better performance with millions of cores.

Acknowledgment

This work was sponsored by the Office of Naval Research (ONR) under grant N00014-01-00-1-7, with Drs. Ki-Han Kim and Thomas C. Fu as the program managers. The simulations presented in this paper were performed at the Department of Defense (DoD) Supercomputing Resource Centers (DSRCs) through the High Performance Computing Modernization Program (HPCMP).

References

[1] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1:269–271, 1959.

[2] M. Herrmann. A domain decomposition parallelization of the fast marching method. In Annual Research Briefs, pages 213–225. Center for Turbulence Research, Stanford University, Stanford, CA, 2003.

[3] W.-K. Jeong and R. T. Whitaker. A fast iterative method for Eikonal equations. SIAM Journal on Scientific Computing, 30(5):2512–2534, 2008.

[4] E. Rouy and A. Tourin. A viscosity solutions approach to shape-from-shading. SIAM Journal on Numerical Analysis, 29(3):867–884, 1992.

[5] R. Sedgewick and K. Wayne. Algorithms. Addison-Wesley, Upper Saddle River, NJ, 4th edition, 2011.
[6] J. Sethian. *Level Set Methods and Fast Marching Methods: Evolving Interfaces in Computational Geometry, fluid Mechanics, Computer Vision, and Materials Science*. Cambridge University Press, Cambridge, 2nd edition, 1999.

[7] J. A. Sethian. A fast marching level set method for monotonically advancing fronts. *Proceedings of the National Academy of Sciences*, 93(4):1591–1595, 1996.

[8] J. Tsitsiklis. Efficient algorithms for globally optimal trajectories. *IEEE Transactions on Automatic Control*, 40(9):1528–1538, 1995.

[9] O. Weber, Y. S. Devir, A. M. Bronstein, M. M. Bronstein, and R. Kimmel. Parallel algorithms for approximation of distance maps on parametric surfaces. *ACM Transactions on Graphics*, 27(104):1–16, 2008.

[10] J. Yang, T. Michael, S. Bhusan, H. Akira, Z. Wang, and F. Stern. Motion prediction using wall-resolved and wall-modeled approaches on a Cartesian grid. In *Proceedings of the 28th Symposium on Naval Hydrodynamics*, September 2010. Pasadena, CA.

[11] H. Zhao. A fast sweeping method for Eikonal equations. *Mathematics of Computation*, 74(250):603–627, 2005.

[12] H. Zhao. Parallel implementations of the fast sweeping method. *Journal of Computational Mathematics*, 25(4):421–429, 2007.