An Interactive Parallel Multiprocessor Level-Set Solver with Dynamic Load Balancing

Suyash P. Awate and Ross T. Whitaker

UUCS-05-002

School of Computing
University of Utah
Salt Lake City, UT 84112 USA

Abstract

Level-set methods, which deform implicitly defined surfaces using partial differential equations, have applications in a wide range of fields including vision, image processing, visualization, graphics, and computational physics. We propose a novel interactive parallel scalable level-set algorithm, based on a narrow band method, which incorporates dynamic load balancing. We show results on a distributed-shared-memory SGI multiprocessor. The interactive update rates combined with real-time visualization allow users to dynamically control the motion of the level set surface.

An Interactive Parallel Multiprocessor Level-Set Solver with Dynamic Load Balancing

Suyash P. Awate, Ross T. Whitaker
Scientific Computing and Imaging Institute,
School of Computing
University of Utah

March 8, 2004

Abstract

Level-set methods, which deform implicitly defined surfaces using partial differential equations, have applications in a wide range of fields including vision, image processing, visualization, graphics, and computational physics. We propose a novel interactive parallel scalable level-set algorithm, based on a narrow band method, which incorporates dynamic load balancing. We show results on a distributed-shared-memory SGI multiprocessor. The interactive update rates combined with real-time visualization allow users to dynamically control the motion of the level set surface.

1 Introduction

Level-set methods [17, 14], which deform implicitly defined surfaces using partial differential equations, have applications in wide ranging fields including computer vision [8, 22, 11], image processing [24, 13, 20, 3], visualization [23, 19], graphics [4, 18], and computational physics [17, 14]. Level-set methods form a powerful tool in modeling surface deformations because they avoid many problems associated with deformations using parametric surfaces. For instance, the deformation of parametric surfaces often requires frequent regularization of surface elements without which the deterioration of the surface can lead to numerical inaccuracies and instabilities [14]. Also handling topological changes like merging and splitting of parametrically represented surfaces can be difficult.

The level-set method represents the deforming surface using a scalar function, which we will call the *embedding*, whose domain is the same as the range of the associated surface. The motion of the surface is computed by solving a corresponding partial differential equation (PDE) on the embedding. A straightforward strategy for computing the surface deformation is to solve the level-set PDE on the entire embedding, and thus the nested family of level sets evolve simultaneously. If one is interested only in a single level set (i.e. a single curve or surface), this strategy is inefficient, because each level set evolves independently from the others. The *narrow band approach* [1] exploits this fact and solves the level-set PDE in a band of grid points around the level set of interest, generating a computational speedup of an order of magnitude. Whitaker

[22] proposed the *sparse-field method*, which restricts the computational domain to several layers around the designated level set. The layers are visited via a linked-list data structure, and the domain is updated as the surface moves. This approach, and related approach of [15], have a computational complexity like that of parametric surfaces, which is proportional to the area of the surface rather than the size of the space in which the surface is embedded. In spite of these advances, most level-set applications do *not* run at interactive rates which limits their application to real world problems.

A common technique of speeding up the computation of PDEs is to exploit the inherent, fine-grained, parallelism in finite difference schemes. Parallel versions of such algorithms can be implemented on clusters of computers, multiprocessor/shared-memory systems, or specialized hardware such as graphics processing units [10, 9].

In this paper we introduce a parallel algorithm for the sparse-field method for a distributed-shared-memory multiprocessor. The dynamic nature of the level set and the corresponding computational domain calls for careful dynamic load balancing. We propose a method which is simple yet effective. The algorithm is demonstrated on an SGI Origin 3000. The code is written within the Insight Toolkit (ITK) [6], which is an open-source software library for multidimensional image processing, segmentation, and registration. The algorithm is implemented as a *solver* within ITK as part of a general, modular framework that the toolkit provides for solving PDEs. This paper also shows results from an interactive application, using ITK and the proposed solver, that allows users to perform level-set segmentations of volume data sets.

The organization of the remainder of paper is as follows. In Section 2 we describe some recent papers on the computational aspects of level sets. Section 3 describes the sparse-field algorithm, which is the basis for this paper, in some detail. Later in Section 4 we talk about the various issues in parallelizing the algorithm on multiprocessors, such as information transfer and synchronization between threads, and then give the details of the parallel version of the sparse-field algorithm. Section 5 gives some details about the SGI Origin 3000 multiprocessor and describes several important aspects of the implementation relating to this particular architecture. Section 6 describes results of experiments with our multiprocessor implementation, and Section 7 summarizes the paper and our conclusions about this work.

2 Related Work

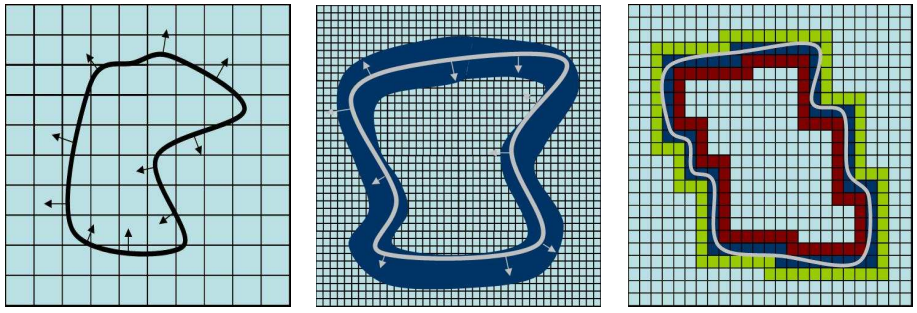
The related work to this paper covers several different areas. The first is the development of the *narrow-band algorithm* [1] and related approaches [22, 15]. The narrow-band algorithm restricts the computation of the level-set PDE to a band of 10–20 grid points around the level set of interest. Grid points outside the band are set to a constant value and therefore do not get updated, as shown in Figure 1. All grid points within the band undergo the same level-set update equation. Where appropriate, the velocity is extended from the position of the zero-level set (designated curve or surface) to the rest of the computational domain using nearest distance. Special care must be taken at the edge of the domain to ensure that the boundary conditions are appropriate. As the level set approaches the edge of the computational domain, the solver

stops the Euler updates, computes the current position of the level set (to subgrid accuracy), and rebuilds the embedding within a new computational domain centered around the current surface. A trade-off exists between the size of the band and frequency at which the evolving surface requires a reinitialization of the narrow band.

The sparse-field approach [22] performs computations on a narrow domain that is *one cell wide*. A narrow band, precisely of the width needed to compute the derivatives, is maintained, via a distance transform as the surface evolves. Thus, it is essentially a narrow band method in which the domain is reinitialized after every Euler update. This reinitialization is made computationally feasible via an approximation to the signed distance transform. Keeping track of this narrow band at each iteration requires maintaining lists of neighborhood points in layers. The sparse-field method computes the velocity of the zero-level set using a first-order approximation to its position from the updated grid points. Thus, there is neither an extension of velocity required nor any boundary stability issues, unlike the narrow-band method. Another related work is that of Peng et al. [15]. They also compute solutions of the PDE on a very narrow band. They maintain an explicit list of points in the domain but update the distance transform, at each iteration, with another PDE.

The proposed algorithm addresses a computational issue common to all the above algorithms. That is, in the context of parallel implementations the irregular and dynamic nature of the computational domain presents some challenges for effective load balancing. It also addresses some particular aspects of the sparse-field method (and that of [15]), which deal with the maintaining of distributed versions of the dynamic data structures that describe the computational domain.

Another area of related work is that of adaptive methods for computing level-set solutions [2, 16]. These methods achieve some computational advantage by solving for coarse resolutions in regions of the domain where there is a low density of level sets or where the level sets are relatively flat. While these approaches are promising, they have yet to achieve the level of speedup associated with multiprocessor solutions combined with narrow-band methods. In this paper we demonstrate the proposed algorithm for solutions at fixed resolu-



(a) (b) (c)

Figure 1: (a) A deforming 1D surface (b) The narrow band around the level set of interest (c) A minimal-width band around the level set in the sparse-field approach

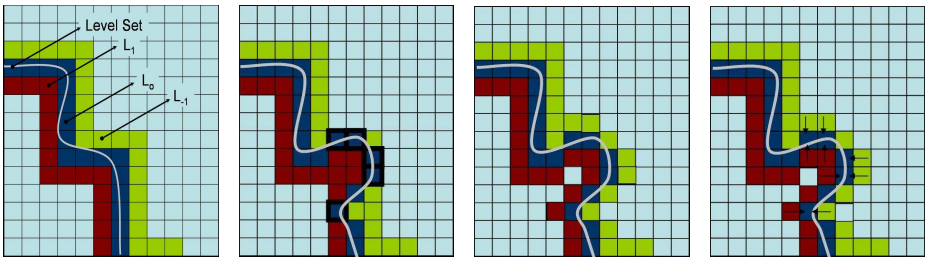
tion, but it has implications for adaptive methods, which could result in further gains. The use of parallelism with adaptive resolution schemes is beyond the scope of this paper and is a subject of ongoing research.

Finally, the work of Lefohn et. al. [9, 10] is also relevant. They use a graphics processing unit (GPU) to achieve 10–15× speedups over the single-processor, sparse-field algorithm. They treat the graphics processor as a parallel vector computer (with some special limitations). They decompose the computational domain into *tiles* which are active or inactive, depending on whether or not they contain the level set of interest. The texture-rendering engine of the GPU processes only active tiles. Tiles are dynamically added and removed from the active list through a hierarchical mechanism with support from the specialized graphics hardware. The computational domain is updated, as in [15], with an additional PDE that pushes solutions toward a distance transform. There are, however, inherent limitations to the GPU approach. First, the size of the problem is restricted by size of the texture memory in state-of-the-art GPUs. This capacity is increasing, but it is still a fraction of the RAM available in a modern multiprocessor machine. Also, the computational speedup comes at the cost of a very restricted set of capabilities that must be very carefully hand-crafted for the GPU. Thus, the GPU solution does not extend easily to problems such as surface reconstruction [22, 21], which require more complex, heterogeneous operations for each grid point and each Euler update. Furthermore, the computational gain of the GPU solution is limited by the progress of GPU hardware, which is improving rapidly, but will offer only about 20× improvements over CPUs in the near future. Thus, the generality of a scalable multiprocessor solution is still quite an important development.

3 The Sparse-Field Algorithm

For this work we represent a deformable surface, L, as a level set of a 3D scalar function, $\phi(x,t)\colon \Re^3 \times \Re \mapsto \Re$. The surface, corresponding to some level set k, is implicitly defined as $L = \{x|\phi(x) = k\}$, where we assume $k = 0$ without a loss of generality. Define u_p to be a discrete sampling of the embedding ϕ, where p is the grid point at location $\{x, y, z\}$, defined on a rectilinear grid. The minimal connected set of grid points that are closest to the level set is referred to as the *active set*, and the individual elements in this set are the *active points*. The neighborhood of the active set in the embedding is defined in layers, L_i for $i = \pm 1, \ldots, \pm N$. Here i indicates the city block distance of a neighborhood point from the nearest active point. The positive (negative) subscripts denote layers inside (outside) the active set. The active set could be considered as the layer L_0. We now give the sparse-field algorithm, with relevant illustrations in Figure 2.

1. Initialize.
   (a) Construct the active set L_0 by determining the level set of interest.
   (b) Populate the layers L_i and initialize the embedding values for points in the layers.
2. Solve the PDE on the level set.



(a) (b) (c) (d)

Figure 2: (a) An example using the sparse-field approach with 3 layers. The active set (layer L_0) is shown in blue color. (b) Updating the active set location and its embedding values (c) Updating the sparse band location (layers L_i) based on the new active set location (d) Updating the embedding values for points in the sparse band based on the embedding values of the active points

   (a) For each point p in L_0, compute the net force on p by summing up the internal and external forces. The internal forces arise from the local geometry of the level set e.g. regularization constraints. The external forces arise from the application.

   (b) Compute the time step for the PDE evolution.

3. Update the level set (and active set) location.

   (a) For each active point, p, do the following:

      i. Compute the net change in u_p and update u_p.
      We maintain lists of grid points, called the *status lists*, for points that are changing distances from the level set. These are named S_i for $i = \pm 1, \ldots, \pm(N+2)$.
      If $(u_p > 0.5)$ then insert p in S_{+1}. Similarly, if $(u_p < -0.5)$ insert p in S_{-1}.

4. Update the sparse band location in the embedding.

   (a) For each status list S_i, in the order $i = \pm 1, \ldots, \pm(N+1)$, do the following:

      i. For each point p on the status list S_i, remove p from the layer $L_{i\mp 1}$ and add p to the layer $L_{i\mp 2}$.

      ii. If $i \neq (N+1)$ then add all neighbors of p belonging to layer L_i on the $S_{i\pm 1}$ list. If $i = (N+1)$ then add neighbors of p not in any of the layers (outside the band of layers) to $S_{i\pm 1}$.

   (b) For each point p on the status list $S_{\pm(N+2)}$ add p to layer $L_{\pm N}$.

   (c) Empty all status lists.

5. Update the embedding values for points in the new band.

   (a) For each layer L_i, in the order $i = \pm 1, \ldots, \pm N$, do the following:

      i. For each point p in layer L_i two events can happen. Either p has some neighbors in the next inner layer $L_{i\mp 1}$ or it has none. If

neighbors exist then update u_p based on the neighbor closest to the active set (by approximating the distance transform). If no neighbors exist for p then remove p from L_i and if $i \neq N$ add p to $L_{i\pm1}$, the next level away from the active set.

6. To continue the surface deformations go to step 2, otherwise terminate.

4 A Parallel Sparse-Field Algorithm

In this section we first discuss the various issues in parallelizing the sparse-field algorithm on multiprocessors, concerning information transfer and synchronization between threads, and then give the details of the parallel version of the algorithm.

4.1 Load Balancing Issues

Maintaining a proper load balance is critical for parallel algorithms to achieve efficient speedups. Load balancing requires distributing the level-set surface processing among threads. We do this by partitioning the volume among the threads by forming *slabs*. A slab is a piece of a rectilinear 3D grid which includes a contiguous subset of one of the indices (e.g. we use z slabs), as shown in Figure 3. The dynamic load balancing scheme makes sure that positions and heights of the slabs maintain a relatively equal partitioning of the work throughout the execution of the program. Because the work done per Euler iteration is approximately proportional to the number of points in the active set, we use that as a metric for determining the slabs.

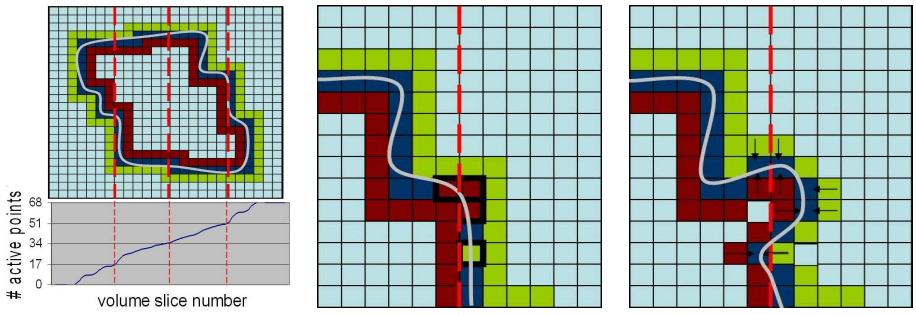


Figure 3: (a) Dynamic load balancing scheme using the active points' distribution. The red dashed lines denote the slab boundaries. The graph represents the cumulative distribution function of the active points lying in planes along a chosen axis (the horizontal axis in this example). (b) Inter thread interactions near slab boundaries during the sparse band update. Here the active set has been updated and we see that sparse-band updates (based on the updated active set) could originate from across the boundary. (c) Inter thread interactions near slab boundaries during the embedding update Here we see that computing embedding values for some points, needs values at other points across the boundary.

The number of active points in each x–y slice gives rise to a histogram over z, denoted by H. The z axis is defined so that each slab resides in contiguous memory locations (i.e. z is the slowest moving index). We discuss more about this later in Section 5. The integral of the histogram H gives a cumulative distribution function. We then create as many partitions in the cumulative distribution function as the number of threads, each partition with roughly the same number of points, as illustrated in Figure 3. The level set continuously deforms as the algorithm proceeds, possibly creating a load imbalance. We can detect this by maintaining a dynamic histogram data structure $H_{dynamic}$, which reflects the latest distribution of active points in the embedding.

Based on this $H_{dynamic}$ we are in a position to perform load balancing after every Euler iteration. However, doing this would incur a significant overhead. We observe that to maintain numerical stability the level set is not allowed to evolve at a high speed by limiting the PDE time step [17, 14]. In fact the level set moves at most 1 grid unit per iteration, typically moving at the rate of about 0.2 grid unit per iteration. This suggests that the load balance does not shift significantly in one iteration, allowing us to perform load redistribution after a few tens of iterations.

4.2 Thread Synchronization Issues

For the parallel algorithm, data dependencies are generated between threads at various stages in the algorithm. These call for synchronization among the threads as the algorithm proceeds. Every thread maintains and operates on lists containing points in its own slab. At times the threads need to transfer data among themselves and remain synchronized. Handling these issues properly is important for the efficiency of the implementation.

The notation in the following sections involves the use of a subscript r for entities. This denotes the fact that the entity is local to thread r that processes points in slab r. An entity without the subscript r may lie in any slab. Thus p_r denotes a point p belonging to slab r. $L_{i,r}$ denotes the layer i for thread r. Similarly, $S_{i,r}$ denotes the status list i for thread r.

4.2.1 While Computing the PDE Time Step

Every thread r computes (every iteration) the local changes in u_{p_r} and a local time step over its slab. However the numerical algorithm needs a common global time step over the entire level set. We choose the least local time step as the global time step. This task is done by a single thread, after all threads have computed the local time steps. During this calculation the other threads are idle.

4.2.2 While Processing the Status Lists

Two threads are said to be *neighbors* if their slabs share a boundary. Thus every thread has exactly two neighbors, excepting the threads that process the first and the last slabs which have one neighbor only.

When thread r processes point p_r in the status list $S_{i,r}$, it needs to find neighbors of p_r in the layer L_i (the next layer farther from the active set). Also at this stage the thread (and neighbor threads) removes points from layer $L_{i\mp1,r}$

and inserts them in layer $L_{i\mp2,r}$. Further, for points p_r near the boundary of the slab r the neighbors of p_r may lie in the neighbor thread's slab, and hence may be processed by the neighbor thread. All these properties together induce a data dependency between neighbor threads. Hence, before each thread processes $S_{i,r}$ it must ensure that its neighbors r' have completely processed $S_{i\mp1,r'}$. We illustrate this boundary phenomenon in Figure 3.

Another issue arises in this context. If thread r, while processing points in its status list $S_{i,r}$, finds neighbor points not in its own slab, what should it do with them ? A thread must not add such points in its status list $S_{i\pm1,r}$, for three reasons. First, a neighbor thread may also add the same points to its status lists leading to redundant processing. Second, the addition of such points could lead to further addition of points belonging to another thread's region in a recursive manner, potentially creating a load imbalance. Third, aspects relating to data placement on shared memory systems makes this practice inefficient. The details of data placement are discussed in Section 5.

We address this issue of processing neighbor points across a slab boundary in the following manner. At points p_r near the boundary of slab r the neighbors of p_r may lie outside the slab, in another slab, say r'. In this case the neighbors are temporarily inserted in a separate list, called the *transfer list* $T_{r',i\pm1,r}$, instead of the status list $S_{i\pm1,r}$. $T_{r',i,r}$ denotes the transfer list i used to transfer points from thread r to thread r' and is a local entity for thread r. When the neighbor thread r' starts to process its status lists $S_{i\pm1,r'}$ it will include the points from the list $T_{r',i\pm1,r}$ that its neighbor possesses and add them to its own status list. Points duplicated in the list are removed as the list is serviced in the next iteration.

4.2.3 While Processing the Layers

When thread r processes grid points p_r in layer $L_{i,r}$, it needs information about the neighbors of p_r in layer $L_{i\mp1}$ (belonging to a layer closer to the active set) to compute u_{p_r}. This creates a data dependency between a thread and its neighbors because when p_r lies near the common boundary its neighbors may lie outside the slab r. Hence, before a thread processes layer $L_{i,r}$ it must ensure that its neighbors have completely processed layer $L_{i\mp1,r}$. We illustrate this boundary phenomenon in Figure 3.

4.2.4 While Load Balancing

Once every few iterations we need to check if the load is still balanced by checking if all threads have roughly the same number of active points to process. A tolerance measure for load imbalance can be quantified in terms of the total number of active points in the level set at that point in time. A single thread performs this check while the others wait. If the load imbalance is severe then the load is redistributed among the threads.

The load redistribution involves recreation of the slabs and correspondingly updating the local layers $L_{i,r}$ as reflected by the new slabs. In other words every thread must ensure that its layers contain all the points in its newly assigned slab and only those. Thus threads need to exchange points between themselves. This is performed efficiently, in parallel, in a synchronized two-stage process. In the first stage, the threads remove those points from their layers that no longer

belong to their new slab and copy them to specific transfer lists (described earlier) $T_{r',i,r}$. Here r' corresponds to the new slab that the point now belongs to. Once all the threads have completed the first stage, each thread moves on to the second stage. Here every thread r examines the transfer lists $T_{r,i,r'}$ (local to other threads) and transfers the points they contain (if any) to its local layers $L_{i,r}$.

4.3 Algorithm Summary

We call the axis along which the volume is divided into slabs (for load balancing) as the z axis. Thread r processes slab r of the volume. The histogram $H_{dynamic,r}$ denotes the local dynamic histogram of active points over z planes in slab r. The histogram $H_{dynamic}$ denotes the global dynamic histogram which reflects the histogram of active points over the entire embedding volume. p_r denotes a point p belonging to slab r. $L_{i,r}$ denotes the local layer i for thread r containing points only in slab r. Similarly, $S_{i,r}$ denotes the status list i for thread r. $T_{r',i,r}$ denotes the local transfer list i used to transfer points from thread r to thread r' and is local to thread r.

1. Initialize: For now we assume that the data structures have been initialized suitably and the load is well balanced. We discuss more about this later in Section 5. Also assume that all the status lists and the transfer lists are empty.

2. Solve the PDE on the level set.

   (a) For each active point p_r in $L_{0,r}$ the thread r computes the net force on p_r by summing up the internal and external forces.

   (b) Thread r computes the local time step for PDE evolution (over the thread's slab).

   (c) Wait until all the threads have computed the time step.

3. A single thread computes the global time step as the minimum among the local time steps computed by each thread (while all other threads wait).

4. Update the level set (and active set) location.

   (a) For each active point p_r in $L_{0,r}$ the thread r does the following:

      i. Compute the net change in u_{p_r} and update u_{p_r}. If $(u_{p_r} > 0.5)$ then insert p_r in $S_{+1,r}$. Similarly if $(u_{p_r} < -0.5)$ insert p_r in $S_{-1,r}$.

      ii. Update the histogram $H_{dynamic,r}$ when p_r moves out of the active set.

      iii. Signal neighbors and wait for similar signals in return.

5. Update the sparse band location in the embedding.

   (a) For each status list $S_{i,r}$, in the order $i = \pm 1, \ldots, \pm(N+1)$, thread r does the following:

      i. For all neighbors r' of thread r, copy points from list $T_{r,i,r'}$ to the status list $S_{i,r}$.

ii. For each point p_r on the status list $S_{i,r}$, remove p_r from the layer $L_{i\mp1,r}$ and add p_r to the layer $L_{i\mp2,r}$. Update the histogram $H_{dynamic,r}$ when p_r moves into the active set.

iii. If $i \neq (N+1)$ then add all neighbors n_r of p_r belonging to layer $L_{i,r}$ on the $S_{i\pm1,r}$ list. If $i = (N+1)$ then add neighbors n_r of p_r not in any of the layers (outside the band of layers) to $S_{i\pm1,r}$. If neighbors $n_{r'}$ of p_r belong to slab r' then add them to the transfer list $T_{r',i\pm1,r}$.

iv. Signal neighbors and wait for similar signals in return.

(b) For all neighbors r' of thread r, copy points from list $T_{r,\pm(N+2),r'}$ to the status list $S_{\pm(N+2),r}$. For each point p_r on the status list $S_{\pm(N+2),r}$ add p_r to layer $L_{\pm N,r}$. Empty all status lists and transfer lists.

6. Update the embedding values for points in the new band.

(a) For each layer $L_{i,r}$, in the order $i = \pm1, \ldots, \pm N$, thread r does the following:

i. For each point p_r in layer $L_{i,r}$ two events can happen. Either p_r has some neighbors in the next inner layer $L_{i\mp1,r}$ or it has no such neighbors. If such neighbors exist then update u_{p_r} based on the neighbor closest to the active set (by approximating the distance transform). If no such neighbors exist for p_r then remove p_r from L_{i_r} and if $i \neq N$ add p_r to $L_{i\pm1,r}$, the next level away from the active set.

ii. Signal neighbors and wait for similar signals in return.

7. To continue the surface deformations go to the next step, otherwise terminate.

8. Dynamic load balancing.

(a) Once every few tens of iterations, do the following:

i. Suspend all threads except one which reconstructs the global histogram $H_{dynamic}$ from all local histograms $H_{dynamic,r}$, computes the cumulative distribution function and checks for load imbalance.

ii. If the load is not balanced perform dynamic load redistribution, in parallel, as follows:

A. For each point p in the layers $L_{i,r}$ for $i = \pm1, \ldots, \pm N$: if p belongs to slab r' then remove p from $L_{i,r}$ and insert it in the transfer list $T_{r',i,r}$.

B. Wait until all threads have completed the previous step.

C. For all threads r', copy all points in the transfer lists $T_{r,i,r'}$ into layer $L_{i,r}$.

9. Go to step 2.

5 Implementation

This Section gives some details about the SGI Origin 3000 multiprocessor and describes several important aspects of the implementation, which are specific to the particular architecture.

5.1 The SGI Origin 3000 Multiprocessor System

The SGI Origin series [5] comprises a family of multiprocessor distributed-shared-memory computer systems. It uses a global address space multiprocessor using a cache-coherent non uniform memory access (ccNUMA) architecture. All memory and IO is globally accessible. Because the Origin uses physically distributed memory, the cost of accessing memory depends on the distance between the processor and the memory location.

The memory is distributed in *nodes* which are arranged as the vertices of a *hypercube*. Every memory node contains four processors and for those processors it is the fastest memory. A router hub forms the interface between the processors and the remote memory nodes via the interconnect network. The hypercube connectivity avoids the bandwidth limitations experienced by bus-based architectures and plays a crucial role in enhancing scalability.

We use the Chapman machine in the Origin 3000 series. It consists of R14000 MIPS 64-bit CPUs running at 600 MHz with four processors per memory node and 2GB of local memory per node. The processors are equipped with four-way super-scalar architecture, out-of-order execution and branch prediction. The configuration contains 64 processors in 16 memory nodes arranged in a 4D hypercube. The processors consist of separate instruction and data L1 caches (on-chip) of size 32 KB each and an external L2 cache (instruction and data combined) of size 8 MB. The default page size is 16 KB.

The operating system on the machine is IRIX 6.5. Threads are created using *sproc* system calls. Thread synchronization support is via semaphores and barriers. The *fetchop* library, which allows atomic fetch and increment operations on variables in memory via use of hardware, can be used to build efficient barriers.

5.2 Implementation Issues on the SGI Origin 3000

The largest data structures used are two volumes; one to maintain the embedding values and the other to keep track of the location of the sparse band layers. These volumes also provide information about neighborhoods of points, which is required during the processing of the layers/lists as described earlier. Each thread keeps local linked lists for the layers, status lists, and transfer lists to transfer data between neighbors. We choose the highest axis/dimension for constructing the slabs in the volume because the data is stored in memory in a row major format and doing this makes points in a slab reside in contiguous memory locations. This aids in effective data placement, which is discussed shortly.

We use *sproc* calls to create threads. We use semaphores for the signaling and waiting mechanism between neighbor threads. We use barriers for synchronization between all threads implemented using the native *fetchop* library. We use the IRIX command, namely *dplace*, to increase the page size of the code,

data and stack to 64 KB from the default 16 KB. This can reduce translation look-aside buffer (TLB) misses while accessing large amounts of data. We also use *dplace* to constrain specific threads to run on specific processors and use memory on specific memory nodes. The set of the nodes used is a subset of the smallest hypercube needed to contain all the threads. This keeps the data in memory as near as possible to the processors running the threads, thus minimizing memory latency.

The implementation is highly memory intensive due to the operations on large linked lists and grid points in the two volumes. As a consequence, achieving effective speedup and scalability requires careful data placement. Suboptimal data placements can substantially increase memory access times and create memory bandwidth bottlenecks when many memory access requests get directed towards the same memory node. This problem is compounded as the number of processors increase.

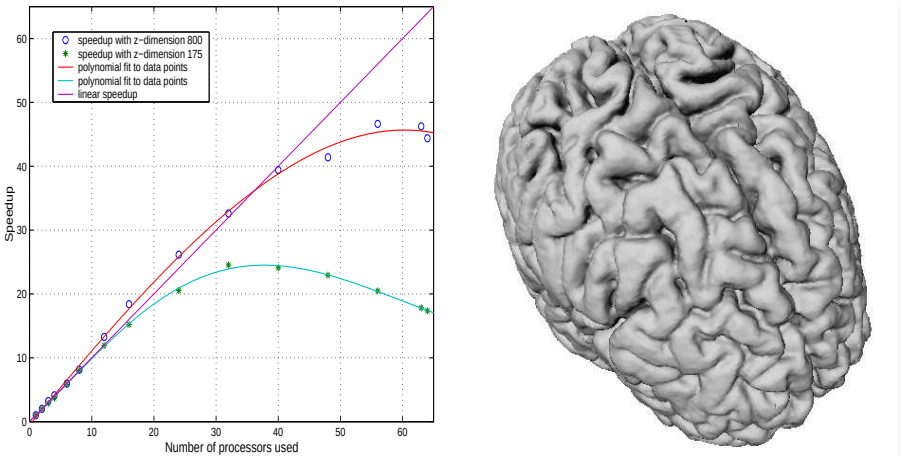
We perform data placement as follows. The default page allocation policy on the SGI Origin 3000 is the *first touch policy* [5]. Under this policy the process which first touches a page of memory causes that page to be allocated on the same node on which the process is running. The allocation and initialization of the two volume data structures (mentioned at the beginning of this section), hence, should not be done by one thread alone. Instead we allocate new copies of the two volumes and let each thread r initialize (write to) the points in slab r, and then discard the old copies. This task is performed in parallel to decrease overhead and maintain scalability. Thus different contiguous parts of the volumes physically reside on different nodes. This helps bring memory lines that the threads would use in the future into the cache and also reduces TLB misses. The extra copy overhead is negligible as compared to the total cost of the initialization.

6 Experiments and Results

The literature shows a variety of applications of level sets to image segmentation, where the level sets are attracted to image based features such as intensity, gradients and edges in the surface deformation process. Typically, the user initializes a contour which evolves until it fits the form of the surface of interest in the image. In our example with an MRI volume dataset, using an intensity based approach, the goal is to define a range of intensity values that classify the tissue type of interest and then construct the level-set deformation PDE on that intensity range. Using the level-set approach, the smoothness of the evolving surface can be enforced to prevent the kind of leaking common in connected-component schemes, as described in [9].

We use a 3D MRI dataset of the head having dimensions of $256 \times 256 \times 175$ for segmentation. A slice from the volume is shown in Figure 5. The slabs, for load distribution, are constructed on the axis with the length of 175 (the z axis). The goal is to segment the brain cortex. Figure 4 shows the scalability of the algorithm on the above mentioned dataset and an example of the segmented brain cortex surface.

We perform experiments to test the scalability of the algorithm using upto 64 processors. We choose a suitable program topology which decreases memory latencies for the threads. For timing the experiments, we use the SGI *perfex*



(a) (b)

Figure 4: (a) Scalability graphs (b) Segmented brain cortex from the 256×256×175 MRI head dataset

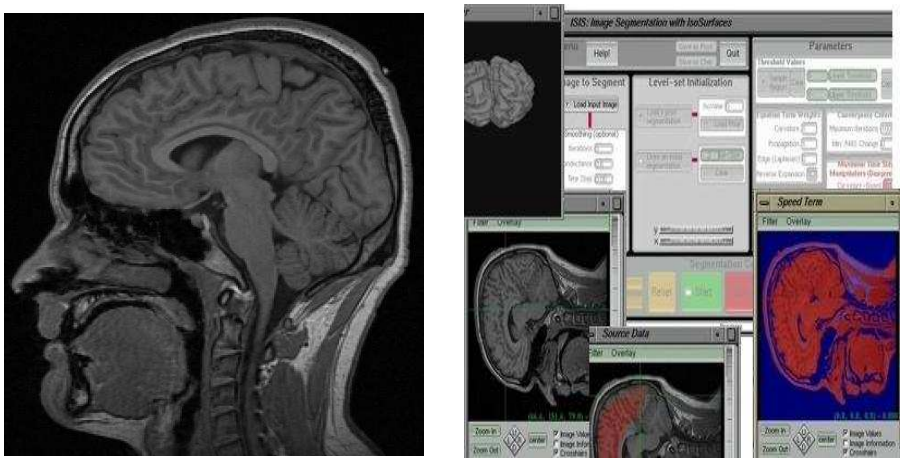
tool which is a command line interface to process hardware event counters [5] on the SGI Origin. We count the number of cycles taken by each thread during the execution. This includes the time to initialize all the thread data structures. We also enable counting at the exception level using the *-x* flag for the *perfex* command, which accounts for cycles spent on behalf of the user during, for example, TLB refill exceptions [5]. We plot graphs using the numbers for slowest thread because the slowest thread represents the time taken for the entire execution.

Using the z dimension length of 175 does not yield linear speedup when using more than 32 threads. This is expected because every thread incurs a cost for communication with its neighbors which is proportional to the number of points on the slab boundaries. As the number of threads increase (and the slabs become smaller), the ratio of the points on the slab boundary to all the points in the slab increases, thus increasing the overhead. This kind of a phenomenon, where the relative communication cost increases as number of threads increase, is typical with virtually any parallel algorithm. Another experiment, with a volume 800 units long, sheds some light on the scalability. With a grid of 256×256×800 and slabs constructed along the z axis, the performance continues to improve with as many as 50 processors. The results are shown in Figure 4.

Number of processors	1	2	3	4	6	8	12
Iterations per second	0.72	1.46	2.08	2.67	4.26	5.84	8.65
Number of processors	16	24	32	40	48	56	64
Iterations per second	11.03	14.90	17.82	17.50	16.65	14.87	12.61

Table 1: Level-set solver speeds for the 256×256×175 MRI head dataset

Table 1 shows the number of level-set updates performed per second for the brain cortex segmentation seen in Figure 4 when the surface is near the final segmentation and deforming considerably. We see that this surface evolves at a rate of about 18 iterations per second using 32 processors. We also have an



(a) (b)

Figure 5: (a) A slice from the MRI head dataset of size 256×256×175 (b) Some snapshots of the user interface with real-time visualization

application (built in ITK) that dynamically extracts the iso-surface (from the embedding), corresponding to the deforming surface, using a marching cubes [12] algorithm and renders the resulting mesh using OpenGl [7]. We parallelize the iso-surface extraction using the same threads used for the parallel level-set solver. The iso-surface extraction and rendering need not be done every iteration because the level set evolves very slowly, as mentioned earlier in Section 4. In addition, we present a graphical user interface to the user to aid in this entire process of segmentation using intensity ranges. It allows users to select intensity ranges, initialize the surface, and control surface smoothness parameters while the surface deforms to obtain a segmentation. Some snapshots from the application are shown in Figure 5. Thus the user can dynamically see and control the motion of the level-set surface. This process of deformation of the initial surface into the brain cortex (about 700–800 iterations), along with real-time visualization (with the deforming surface rendered every 5 iterations), takes less than a minute using 32 processors and we obtain level-set update rates of about 14 iterations per second (on average).

7 Conclusion

We have presented a novel parallel algorithm for the sparse-field level-set solver incorporating dynamic load balancing. We have implemented the parallel algorithm on the distributed-shared-memory SGI Origin 3000 multiprocessor. The implementation performs level-set surface deformation updates at interactive rates and has good scalability. The interactive update rates combined with real-time visualization allow users to dynamically control the motion of the level-set surface.

Further experiments may be performed with the dynamic load balancing model using a more advanced scheme based on a space partitioning technique e.g. k-d trees. In general, domain decomposition schemes which try to minimize

the ratio of the number of active points near the boundary to the total number of active points in the region, may reduce inter-thread-communication overhead. However, because of the dynamic nature of the level set and the corresponding computational domain, such schemes need to be dynamic too. This may result in an algorithm of much higher complexity and in turn might result in a higher load-balancing overhead.

References

[1] David Adalsteinsson and James A. Sethian. A fast level set method for propagating interfaces. *J. Comput. Phys.*, 118(2):269–277, 1995.

[2] D. L. Chopp and J. A. Sethian. Motion by intrinsic laplacian of curvature. *Interfaces and Free Boundaries*, 1:107–123, 1999.

[3] Vidya Elangovan and Ross T. Whitaker. From sinograms to surfaces: A direct approach to the segmentation of tomographic data. In *Proceedings of the 4th International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 213–223. Springer-Verlag, 2001.

[4] Nick Foster and Ronald Fedkiw. Practical animations of liquids. In Eugene Fiume, editor, *SIGGRAPH 2001, Computer Graphics Proceedings*, pages 23–30. ACM Press / ACM SIGGRAPH, 2001.

[5] http://techpubs.sgi.com/library. SGI IRIX 6.5 Man Pages.

[6] http://www.itk.org. National library of medicine insight segmentation and registration toolkit.

[7] http://www.opengl.org. OpenGl.

[8] B. Kimia, A. Tannebaum, and S. Zucker. Shapes, shocks, and deformations I: the components of two-dimensional shape and the reaction-diffusion space. *Int. J. Comput. Vision*, 15:189–224, 1995.

[9] A. E. Lefohn, J. E. Cates, and R. T. Whitaker. Interactive, gpu-based level sets for 3d segmentation. *Medical Image Computing andComputer Assisted Intervention (Miccai)*, pages 564–572, 2003.

[10] A. E. Lefohn, J. M. Kniss, C. D. Hansen, and R. Whitaker. Interactive deformation and visualization of level set surfaces using graphics hardware. *IEEE Visualization 2003*, pages 75–82, October 2003.

[11] M. Leventon, E. Grimson, and O. Faugeras. Statistical shape influence in geodesic active contours. In *CVP'00*, 2000.

[12] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169. ACM Press, 1987.

[13] R. Malladi and J. A. Sethian. A unified approach to noise removal, image enhancement, and shape recovery. *IEEE Trans. on Image Processing*, 1996.

[14] S. Osher and R. Fedkiw. *The Level Set Method and Dynamic Implicit Surfaces*. Springer-Verlag, New York, 2003.

[15] D. Peng, B. Merriman, H. Zhao, S. Osher, and M. Kang. A pde based fast local level set method, 1999.

[16] M. Rumpf and R. Strzodka. Level set segmentation in graphics hardware. In *IEEE International Conference on Image Processing*, pages 1103–1106, 2001.

[17] J. A. Sethian. *Level Set Methods and Fast Marching Methods*. Cambridge University Press, 1999.

[18] T. Tasdizen, R. Whitaker, P. Burchard, and S. Osher. Geometric surface processing via normal maps. *ACM Transactions on Graphics*, 2003.

[19] R. Westermann, C. R. Johnson, and T. Ertl. Topology preserving smoothing of vector fields. *IEEE Transactions on Visualization and Computer Graphics*, 2001.

[20] R. Whitaker. Reconstructing terrain maps from dense range data. In *IEEE International Conference on Image Processing*, pages 165–168, October 2001.

[21] R. Whitaker and V. Elangovan. A direct approach to estimating surfaces in tomographic data. *Journal of Medical Image Analysis*, 6(3):235–249, 2002.

[22] Ross T. Whitaker. A level-set approach to 3d reconstruction from range data. *Int. J. Comput. Vision*, 29(3):203–231, 1998.

[23] Ross T. Whitaker. Reducing aliasing artifacts in iso-surfaces of binary volumes. In *Proceedings of the 2000 IEEE symposium on Volume visualization*, pages 23–32. ACM Press, 2000.

[24] Ross T. Whitaker and David T. Chen. Embedded active surfaces for volume visualization. In *SPIE Medical Imaging 94*, 1994.