Eurographics Symposium on Parallel Graphics and Visualization (2010)
J. Ahrens, K. Debattista, and R. Pajarola (Editors)

Self-Scheduled Parallel Isosurfacing using Distributed Span Space on Cell

Michael R. Caruso[1] and Timothy S. Newman[1]

[1]Department of Computer Science, Univ. of Alabama in Huntsville , USA

Abstract

A method designed for fast isosurfacing on Cell platforms is introduced. It well-utilizes limited amounts of local memory by exploiting a block-based span space. Exploitation goes beyond the usual steps of avoiding span space tiles whose range does not contain the isovalue. In particular, the method keeps resident in local memories most span space information in addition to the parts of the volume most likely to be examined if multiple isovalues are explored. The method also performs distributed self-scheduling of isosurfacing work among the Cell's Synergistic Processing Units (SPUs) without explicit centralized computation of workload or assignment of work. Results are also presented for trials on the Playstation-3, including comparison to another fast, parallel isosufacing method (which is faster than prior reported parallel methods on Cell).

Categories and Subject Descriptors (according to ACM CCS): I.3.1 [Computer Graphics]: Hardware Architecture—Parallel processing; I.3.6 [Computer Graphics]: Methodology and Techniques—Graphics data structures and data types; I.3.m [Computer Graphics]: Miscellaneous—Isosurfaces

1. Introduction

Volumetric datasets that are organized as collections of scalar values on rectilinear grids are produced by many scanners (e.g., CT) and simulations. One common means for information discovery from such datasets is the extraction and rendering of a surface (i.e., an *isosurface*) of a constant value (i.e., an *isovalue*). Isosurfacing often involves much exploration of the isosurface and of the isovalue space, usually via geometric transformations and trial-and-error experimentation, respectively. Fast isosurfacing can benefit the exploration process by keeping the user better-engaged with their exploration task as well as making it more possible to explore more of the isovalue space. Isosurfacing is also used to achieve certain special effects in graphics applications, and in such applications, the isosurfacing is usually just one part of a larger rendering; speed is of the essence.

In this paper, a new method for quickly performing the most popular sort of isosurfacing [NY06]—Marching Cubes (MC) isosurface extraction—on the Cell commodity processor is described. Versions of Cell are currently used in both the Sony Playstation-3 (PS-3) consumer graphics device and in servers. Cell is attractive for visualization tasks due to its high computational potential, especially considering that this potential is available in inexpensive systems such as the PS-3. The paper's new method, which uses a novel distributed self-scheduling mechanism well-suited to the Cell architecture, exhibits substantially better Cell performance than prior Cell-specific isosurfacing methods.

The rest of the paper is organized as follows. In Section 2, the Cell architecture is described. In Section 3, related work is discussed. The details of the new method are presented in Section 4. Results and comparisons against other methods are presented in Section 5. The paper is concluded in Section 6.

2. Cell Architecture

The work here is applied on the PS-3. The PS-3 architecture includes a Cell processor, Rambus-based main memory, and Reality Synthesizer (RSX) graphics processor. The RSX graphics processor is based on Nvidia's G70 architecture. The PS-3's Cell processor includes one PowerPC processor Element (PPE) (which has a two-level cache [IST08]) and eight Synergistic Processor Elements (SPEs). (Cells used in

DOI: 10.2312/EGPGV/EGPGV10/073-079

the PS-3 have just seven functioning SPEs, however, which allows for increased chip yield.) Each SPE has a Synergistic Processor Unit (SPU) and memory flow controller (MFC). The PPE and SPEs on our PS-3 run at 3.2GHz.

Our PS-3 runs linux. The linux hypervisor on PS-3 disables access to the RSX. It also claims one SPE, leaving six usable SPEs for our experiments.

A block diagram of the Cell layout is shown in Figure 1.

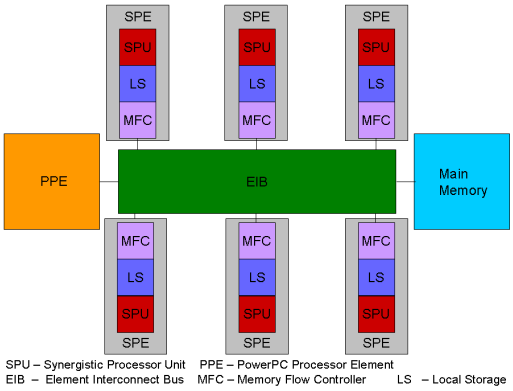

Figure 1: *Block Diagram of Usable Components of Linux-based PS-3 Cell, based on [IST08].*

Each SPU has 128 128-bit registers and runs a SIMD instruction set that allows small-scale vector operation. 256 KB SRAM of local storage is available on each SPE. SPEs have no local cache. The SPU code and the data stack are limited to the local storage space, although main memory can be indirectly accessed. This main memory access is via direct memory access (DMA) through the MFC. At most 16 main memory DMAs can be outstanding at once. Double buffering of DMAs can be used to hide main memory's latency. An overlay capability allows code sizes exceeding the local storage limit, when such sizes are needed. Our code's size was typically under 20KB per module; overlay was not needed.

SPUs execute in-order and treat all branches as not-taken. The SPUs are dual-issue, with each pipe dedicated to certain instruction types [IST08]. Pipe 0 is for most floating-point, integer, and logical operations. The other pipe handles loads, stores, data shuffles, branch resolutions, etc. In addition, dual issues happen only on clocks where an instruction that Pipe 0 can handle is at an even address and the next odd address contains an instruction that is assignable to Pipe 1.

Although the Cell offers great potential with its many SPEs and its PPE, methods developed for more conventional architectures are unlikely to be able to leverage much of the available computational power. In particular, the limited local storage spaces, pipeline issuance properties, and small-scale SIMD offered by the wide registers present challenges. Typically, taking advantage of the available power requires approaches that differ from the norm in conventional computing. The method we describe later in this paper includes steps that enable leverage of the available capabilities, however.

3. Related Work

Due to the number of application areas to which isosurfacing can be applied, many methods for achieving fast performance for Marching Cubes have been explored in the literature. The first direction that has been pursued includes the strategies that avoid unnecessary processing in regions of the volume that do not contain the isosurface. Methods of this type can be called space-skipping methods. Since Marching Cubes can be applied independently on the cubes that make up a rectlinear scalar volumetric dataset, data-parallel strategies for realizing it have been another popular direction pursued by performance-focused researchers [ZNZ04]. A number of methods that are both parallel and space-skipping have also been described.

Next, we describe some of the related work of these types. We also focus here on past efforts to develop parallel visualization methods geared toward the Cell processor.

The space-skipping methods avoid unnecessary computation in some parts of the volume not intersected by the isosurface. Since usually most cubes are not intersected (i.e., are not active), such methods can often be used to great effect. Many isosurfacing acceleration methods of this type are based on spatial data structures, such as various sorts of octrees [WvG92] or range-of-value data structures, such as span spaces [LSJ96]. Quantized span spaces (organized similar to the arrangement in Shen et al.'s [SHLJ96] ISSUE algorithm) have been found to be quite effective range-of-value data structures. As a result, they are popular space-skipping data structures. Quantized span spaces consist of a set of tiles. The tiles are used to organize the cubes of the dataset. Each tile has an associated span of minimum values and a span of maximum values. For each cube of the dataset, an entry is made in one tile (i.e., in the one tile whose span of minima includes the cube's minimum value and whose span of maxima includes the cube's maximum). Methods like ISSUE are fast since they easily identify dataset cubes whose range of values do not include the isovalue. With ISSUE, isosurface extraction is only considered for cubes stored in tiles whose range of values includes the isovalue. A diagram of a quantized span space with isovalue T is shown in Figure 2. The tiles containing possibly active cubes are shown highlighted in blue. Tiles shown in red do not contain active cubes; no isosurfacing computations are performed on these cubes. (N.B.: In quantized span spaces, for a given isovalue, some tiles are not active, others are active, and a few others are mixed—such tiles could store both active and inactive cubes.)

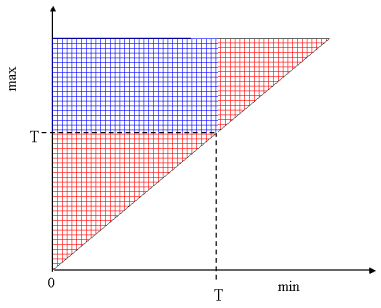

Figure 2: *Quantized span space (active tiles in blue, non-active tiles in red for isovalue T)*

Block-based versions [ZN04] of quantized span spaces have been found to offer the advantage of reduced memory consumption while still enabling very fast isosurfacing [ZNZ04]. In block-based span space, each tile stores information about a group of cubes of the data (rather than about just one cube as in a standard quantized span space). Each group has $m \times n \times p$ adjacent cubes, with each cube a member of exactly one group.

A variety of parallel approaches to MC-based isosurfacing have been described in the literature [NY06], including block-based span space approaches [ZNZ04]. Many of the parallel approaches have been designed for use in supercomputers or clusters (e.g., [LT04, MMD*05]) or in other server-class parallel computers (e.g., [SG99]). A few of the parallel approaches may also be suitable for the Cell. (We explore one such approach in this paper.) Previously, two parallel isosurfacing methods specific to Cell have also been described [JLZZ09, OOC06]. The earliest of these [OOC06] found the isosurface mesh in tetrahedral subdivisions of each cell after assigning an approximately equal number of cubes of the volume to each SPU. That work does not appear to utilize SIMD capabilities of the SPUs or use space-skipping mechanisms. The other work [JLZZ09] performs some components of MC using small-scale vector processing capabilities of the SPUs (which we also do in our method). It also processes the dataset in a block-based manner that is functionally similar to a block-based span space. Specifically, the PPE determines which blocks are active. It stores the active blocks in a buffer. The SPUs monitor the buffer, and whenever an SPU becomes idle, it grabs a block from the buffer and computes the portion of the isosurface passing through it.

Parallel methods for other problems have also been developed for Cell, including methods for other types of visualization (e.g,. [KJ09]). Parallel simulation schemes have also been described and implemented on PS-3 Cells (e.g., [LEV*08]).

4. Methods

Our new method is described next. It utilizes a $(4 \times 4 \times 4)$ block-based span space approach to skip processing in the parts of the volume that do not contain the isosurface.

We follow a MC processing regimen in which first the dataset data points are classified as being above the isovalue or not. Next, the triangle topology type for a cube is determined. We note that the topologies used guarantee formation of a water-tight isosurface since they follow the triangle facetization patterns described by Nielson et al. [NHS02]. Then, the location of the triangle vertices are determined using linear interpolation. Finally, the triangles are formed by linking vertices according to the facetization pattern for the topology. In the case of our block-based approach, these steps were applied using the small-scale vector (SIMD) parallelism operations of the SPUs for the cubes in each block.

Only the blocks that are potentially active are processed, according to the span space rules. Next, we describe the structure of the span space.

The span space tiles are of size two by two. Span space information is stored in a highly compact manner on the SPUs due to the limited amount of local storage on the SPUs. The method keeps as much of the span space resident on the SPUs as possible to reduce main memory accesses since main memory accesses are slower than local memory access and since only a limited number of main memory accesses can actually be outstanding at any instant in time. The tiles that are resident are kept resident for all the isosurface extractions. Due to the limited amount of storage space on the SPUs, it is typically not possible for all span space information to be kept resident.

A map of the block-based span space for two datasets used in our work are shown in Figures 3 and 4. In these figures, the axes are labelled by tile number. Since the datasets are byte formated datasets, there are a total of $(128 \times 128)/2$ tiles. The tiles are presented here in "hot" colors, where a red-to-yellow scale indicates the normalized count for the tile. Higher-count tiles are yellow and low count tiles are dark red, with oranges indicating mid-range values. A logarithmic scale is used in mapping the values to colors. There are no tiles below the diagonal or beyond the line marking tiles 128 on the vertical and horizontal, so these areas of the space should be ignored. Of particular interest here are the tiles in the upper left corner. These are tiles that are likely to be involved in nearly every isosurface extraction since most isovalue choices will include the tiles closest to the upper left corner as active tiles. In the case of the Engine data, there are many medium-gray tiles near the upper left corner. Keeping those tiles and associated blocks of the dataset resident in local storage on the SPUs reduces the memory traffic and improves the speed for iterative isosurfacing. Our approach exploits this characteristic of the span space.

A pre-processing step performed on the PPU steps

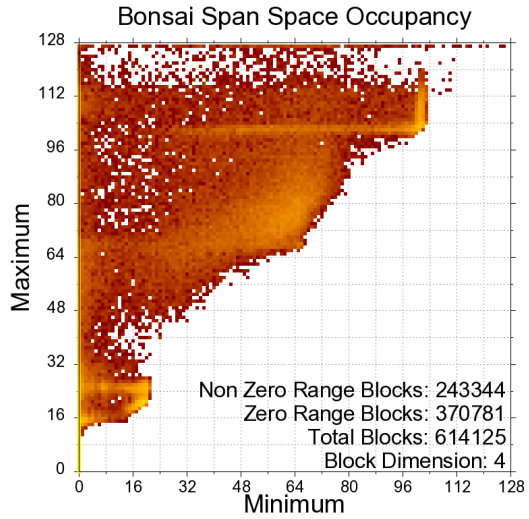

Figure 3: *"Hot color" coding of Span Space, Bonsai Data, axes labelled by bin number*

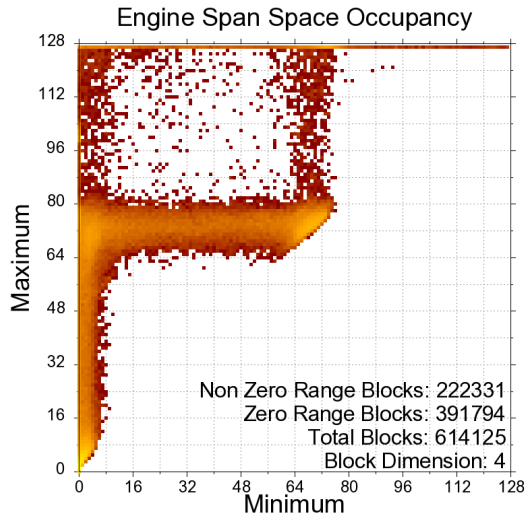

Figure 4: *"Hot color" coding of Span Space, Engine Data, axes labelled by bin number*

through the volume in a block-by-block manner to build the span space. Blocks are associated with the appropriate tiles of span space. Blocks with a range of 0 (i.e., blocks whose minimum and maximum are equal) are not stored in the span space since such blocks will never contain isosurface facets. For the Engine and Bonsai datasets, the counts of the blocks with range of 0 are indicated in the Figures 3 and 4 as "Zero Range Blocks." Each tile is organized as a collection of datums, each containing a small, compact representation of information about 4 blocks. Since main memory access on Cell is via DMA, with the access accomplished fairly efficiently on 16B-aligned addresses, the tile datums are each 16B in size.

The DMA accesses in our implementations were all performed asynchronously. They were also done using a strategy in which a new access request is sent on-the-fly while the results from a prior access result are processed. This strategy reduces idle cycles by overlapping processing of data with retrieval of data.

Once the span space is constructed, its tiles are assigned to the SPUs. Some of the tiles assigned to each SPU are physically distributed to the SPU, as described later. The other assigned tiles are physically located in main memory and retrieved as needed. There is available space to store only 130KB of span space information per SPU, although for 256^3 or smaller datasets, this is enough to hold the basic information about the span space entirely on the SPUs. Certain associated blocks are also distributed among the SPUs with their containing tile. The remainder of the blocks are stored in main memory and retrieved as needed. There is available space to store only 400 associated blocks per SPU.

The assignment of tiles to SPUs begins in the upper left corner of span space. Each tile is visited in order, moving outward from that base tile. As each tile is visited, the block *references* associated with the tile are evenly divided among the SPUs. As a result, each SPU has approximately the same number of pieces of the volume. Actual block information is then assigned to the SPUs in order until the SPUs run out of space. (As stated above, the remaining block data is kept in the main memory.)

In order to achieve a reasonable load balance among the SPUs, each SPU is responsible for approximately the same number of blocks in the span space.

Isosurface extraction is achieved by the SPUs, each of which self-schedules its necessary isosurface extraction activities in the tiles assigned to it. Processing considers only the active tiles assigned to the SPU, with data accessed as needed from either the SPU or the main memory.

4.1. Finer Weighted Method

For comparison with our method, we also experimented with a statically load-balanced work assignment mechanism that is very similar to the finer weighted load balancing mechanism described by Zhang et al. [ZNZ04]. We chose a finer weighted method because it has data-parallel properties that make it easily mappable to the Cell and because Zhang et al. reported that it was fairly fast with very low computational overhead. In finer weighted load balancing, the number of isosurface facets per slice is taken as an estimate of the amount of isosurfacing work for the slice. The facet count is determined by finding the active cubes and then the topological case of each active cube. This can be done quickly via a comparison of each dataset data point followed by a table

look-up for the topological case, as described by Zhang et al. [ZNZ04]. Once the slice-by-slice profile of the work has been determined, the PPU assigns an approximately equal amount of work per SPU, with work assigned on a slice-by-slice basis. In the Zhang et al. approach, all the work was divided at the beginning. Here, we first divided the first quarter of the work among the SPUs, with each SPU getting a set of slices with an approximately equal number of isosurface facets. Whenever an SPU completed one work assignment, another set of slices of approximately equal workload was assigned. The SPU's small-scale SIMD-based operations were used for the isosurfacing in each slice.

5. Results

Next we report results for several datasets. In all, 6 datasets were tested, with at least 5 isovalues per datasets. The presentation here focuses on the Bonsai and Engine datasets from Roettger's Volume Library since extractions for over 100 distinct isovalues have been completed for them. These two datasets are both 256^3 datasets. An example isosurface rendering from the Engine dataset is shown in Figure 5.

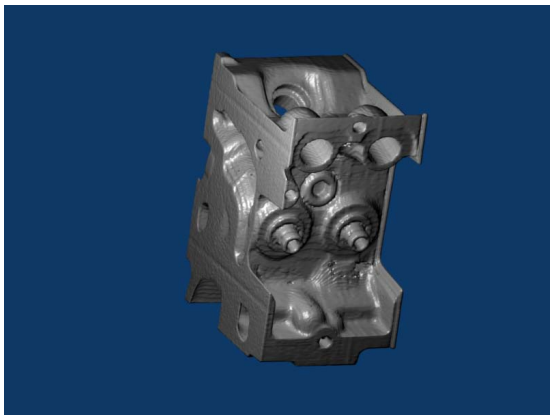

Figure 5: *Engine Dataset Isosurface Rendering*

Figure 6 exhibits isosurfacing times for the Bonsai dataset (=11) for the finer weighted and new approaches versus number of SPUs used. For comparison, extraction time for standard Marching Cubes on one core of an Intel Core 2 Quad Core is also shown. The new approach is faster on one SPU than is a standard Marching Cubes on the Core 2 core, and both approaches are much faster using all the Cell resources than is standard Marching Cubes on the Core 2.

A speedup chart for representative extractions from the Bonsai and Engine datasets are shown in Figures 7 and 8. Both the finer weighted and new distributed span space approaches exhibit nearly linear speedup as the number of SPUs utilized increases.

Performance is well-balanced among the SPUs for both of the techniques we discuss. A graphical picture of the SPU

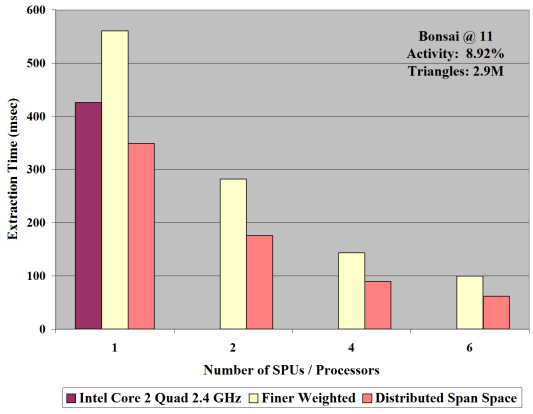

Figure 6: *Bonsai (= 11) Extraction Times vs. SPUs used.*

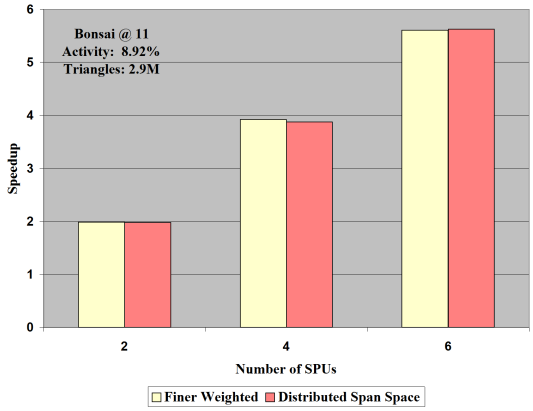

Figure 7: *Bonsai Speedup versus Number of SPUs*

assignments for the finer weighted method on Engine block is shown in Figure 9. This figure is color-coded by SPU; all of the facets for all the assignments to a given SPU are colored in the same color.

5.1. Pipeline Performance

Pipeline performance of the new approach's core isosurfacing component was examined using the IBM Cell SDK SPU Timing Tool. The key measure examined with the tool was the dual issue rate of the block processing actions (interpolation, topology determination, triangle construction, etc.). Of 739 issue slots for a typical block, 318 slots (43%) were dual-issued. Excluding the triangle construction, there were 281 issue slots, 186 of which (66%) were dual-issued. The triangle construction action has less dual issues—due to it being dominated by memory operations (i.e., for Pipe 1 only).

The triangle construction action's use of small-scale vec-

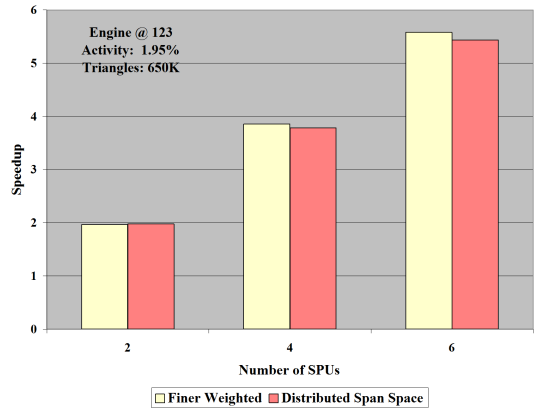

Figure 8: *Engine Speedup versus Number of SPUs*

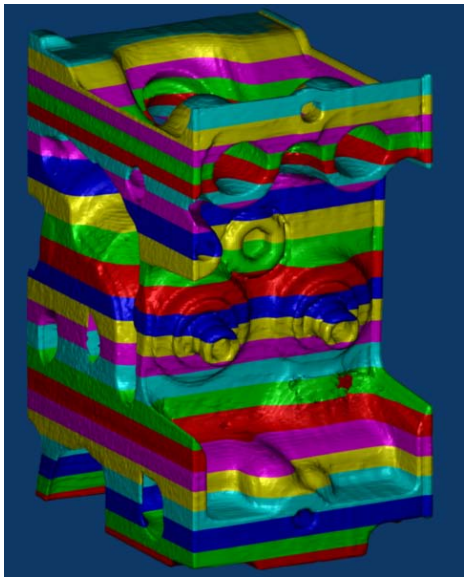

Figure 9: *Color Coding of SPU assignment, Finer Weighted Method, Engine Data*

tor parallel instructions was also found to provide a small (1.09) times performance improvement versus using scalar operations only.

5.2. Some More Analyses

Next, we report some additional analyses and characteristics of the methods. First, we determined (via benchmarking) that the self-scheduling aspect of the distributed span space exhibited a 1.08 times improvement in performance over centralized (PPU-driven) scheduling. A plot of the PPU-driven versus self-scheduled extraction times for several representative extractions on the Bonsai dataset is shown in Figure 10. In the figure, the times are presented versus percentage of active cells.

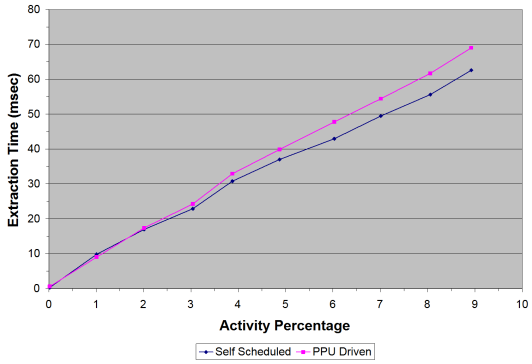

Figure 10: *Self-scheduled versus Centralized Scheduled Span Space Times, Bonsai Data*

Second, we determined the set-up overhead for the span space. This is a one-time (up front) activity, after which subsequent extractions proceed at full speed. For the Bonsai dataset this overhead was 416 msec. For reference, the distributed span space-based extraction took 62.5 msec using 6 SPUs with (i.e., isovalue) 11 for the Bonsai dataset. For this extraction, about 9% of the cells are active. For the Engine dataset at = 4, results are comparable: 7% of the cells are active and the distributed span space-based extraction took 51.2 msec using 6 SPUs. The overhead for Engine was 397 msec.

The finer weighted approach always requires work estimation. For the Bonsai dataset with 11, the total extraction time was 100 msec on 6 SPUs, although 5.3 msec of that was determining the work breakdown. Subsequent isosurfacings still need to do a new work calculation, though. If there will be little experimentation with isovalues, the finer weighted approach is thus preferrable.

5.3. Method Comparisons

Since the prior Cell-based isosurfacing method of O'Conor et al. [OOC06] reported performance for the Bonsai dataset, we next report our relative performance versus it for that dataset. Due to the fact that O'Conor et al. used a Cell clocked at a lower rate (2.1 GHz versus our 3.2 GHz), some extrapolation is necessary to complete the comparison. In addition, O'Conor et al. did not report the isovalue they used. For this comparison, we use an isovalue associated with a very high level of activity for the dataset (= 11, for which 9% of cubes are active). At this level, our extraction times are close to their longest; the comparison is conservative. O'Conor reported a processing rate of 4 million cubes (i.e., 20 million tets) per second on 2 SPUs, with

Method	Triangles	Cubes/Sec.	S
O'Conor [OOC06]	2.95 M	6.08 M	–
Finer Weighted	2.95 M	58.80 M	9.7
Dist. Span Space	2.95 M	94.21 M	15.5

Table 1: *Isosurface Extraction Time Comparison, 2 SPUs, Bonsai*

nearly linear speedup beyond 2 SPUs. (Finer weighted and distributed span space approaches also show nearly linear speedup.) Their results, normalized to a 3.2 GHz clock, are shown versus the finer weighted and distributed span space approaches (also on 2 SPUs) in Table 1. As shown in the table's last column ("S"), the finer weighted approach is about 10 times faster than O'Conor et al.'s method. The distributed span space approach is more than 15 times faster than that prior work.

The other Cell-based method of Jin et al. [JLZZ09] included a comparison study versus the O'Conor et al. method, but with datasets not available to us. Depending on the dataset, the Jin et al. method was between 1.2 and 3.4 times faster than the O'Conor et al. method. Thus, the finer weighted and distributed span space approaches are much faster than the fastest prior methods for Cell-based isosurfacing. In particular, finer weighted is about 3 to 8 times faster and the new distributed span space is 4.5 to 13 times faster than the Jin et al. method.

6. Conclusions and Future Work

A new self-scheduling distributed span space-based method for very fast isosurfacing on the Cell has been introduced. The method exploits properties of span space to keep resident in local storage the most critical parts of the data and supporting data structures, which enables the fast performance. Comparison studies versus the other reported methods for Cell-based isosurfacing suggest the new method is quite fast, offering in some cases an order of magnitude time improvement over the prior methods. The method exploits locality of reference properties of the span space and leverages the small-scale (SIMD) vector parallelism capabilities of the Cell's SPUs to achieve its very high level of performance.

For future work, we would like to explore the performance of the methods on Cell blade systems where more SPUs would be available. We would also like to explore if the distributed span space concept can be well-adapted for use in systems with multiple GPUs.

References

[IST08] IBM, SONY, TOSHIBA: *Cell Broadband Engine Programming Handbook Including the PowerXCell 8i Processor, Version 1.11*. 2008.

[JLZZ09] JIN J., LI B., ZHENG R., ZHANG Q.: Fast isosurface extraction for medical volume dataset on cell be. In *Proc., 2009 Int'l Conf. on Parallel Processing* (2009), pp. 100–107.

[KJ09] KIM J., JAJA J.: Streaming model based volume ray casting implementation for cell broadband engine. *Scientific Programming 17*, 1-2 (2009), 173–184.

[LEV*08] LUTTMANN E., ENSIGN D., VISHAL V., HOUSTON M., RIMON N., VOLAND J., JAYACHANDRAN G., FRIEDRICHS M., PANDE V.: Accelerating molecular dynamic simulation on the cell processor and playstation 3. *J. Computational Chem. 30* (2008), 268–274.

[LSJ96] LIVNAT Y., SHEN H.-W., JOHNSON C. R.: A near optimal isosurface extraction algorithm using span space. *IEEE Trans. on Vis. and Comp. Graphics 2* (1996), 73–84.

[LT04] LIVNAT Y., TRICOCHE X.: Interactive point-based isosurface extraction. In *Proc., Visualization '04* (2004), pp. 457–464.

[MMD*05] MERELLI I., MILANESI L., D'AGOSTINO D., CLEMATIS A., VANNESCHI M., DANELUTTO M.: Using parallel isosurface extraction in superficial molecular modeling. In *Proc., First Int'l Conf. on Dist. Frameworks for Multimedia Apps. (DFMA'05)* (2005), pp. 288–294.

[NHS02] NIELSON G., HUANG A., SYLVESTER S.: Approximating normals for marching cubes applied to locally supported isosurfaces. In *Proc., Vis. '02* (2002), pp. 459–466.

[NY06] NEWMAN T., YI H.: A survey of the marching cubes algorithm. *Computers & Graphics 30*, 5 (Oct. 2006), 854–879.

[OOC06] O'CONOR K., O'SULLIVAN C., COLLINS S.: Isosurface extraction on the cell processor. In *Proc., Seventh Irish Work. on Comp. Graphics* (2006).

[SG99] SULATYCKE P., GHOSE K.: A fast multithreaded out-of-core visualization technique. In *Proc., Int'l Symp. on Par. and Dist. Processing (IPPS/SPDP) '99* (1999), pp. 569–575.

[SHLJ96] SHEN H.-W., HANSEN C., LIVNAT Y., JOHNSON C.: Isosurfacing in span space with utmost efficiency (issue). In *Proc., Visualization '96* (1996), pp. 287–294.

[WvG92] WILHELMS J., VAN GELDER A.: Octrees for faster isosurface generation. *ACM TOG 11*, 3 (1992), 201–227.

[ZN04] ZHANG H., NEWMAN T.: Span space data structures for multithreaded isosurfacing. In *Proc., IEEE Southeastcon '04* (2004), pp. 290–296.

[ZNZ04] ZHANG H., NEWMAN T., ZHANG X.: Case study of multithreaded in-core isosurface extraction algorithms. In *Proc., Eurographics Symp. on Par. Graphics and Vis. '04* (2004), pp. 83–92.