*June 2019*

# GraphiX: A Fast Human-Computer Interaction Symmetric Multiprocessing Parallel Scientific Visualization Tool[1]

Re'em Harel [a,b] and Gal Oren [c,d]
[a] *Department of Physics, Bar-Ilan University, IL52900, Ramat-Gan, Israel*
[b] *Israel Atomic Energy Commission, P.O.B. 7061, Tel Aviv 61070, Israel*
[c] *Department of Computer Science, Ben-Gurion University of the Negev, P.O.B. 653, Be'er Sheva, Israel*
[d] *Department of Physics, Nuclear Research Center-Negev, P.O.B. 9001, Be'er-Sheva, Israel*

**Abstract.** Scientific visualization tools are essential for the understanding of physical simulation, as it gives a visualization aspect of the simulated phenomena. In the past years, data produced by simulations join the big-data trend. In order to maintain a reasonable reaction time of the user's commands many scientific tools tend to introduce parallelism schemes to their software. As the number of cores in any given architecture increase, the need for software to utilize the archutecture is inevitable. Thus, GraphiX - a scientific visualization tool parallelized in a shared-memory fashion via OpenMP version 4.5 was created. We chose Gnuplot as the graphical utility for GraphiX due to its speed as it is written in C. GraphiX parallelism scheme's work-balance is nearly perfect, and scales well both in terms of memory and amount of cores. We achieved a maximum of 560% speedup with 16 cores while visualizing approx 3 million cells.

**Keywords.** Visualization, HCI, OpenMP, GUI, NUMA, SMP, ParaView, VisIt, MATLAB, multi-core

## 1. Introduction

Scientific visualization tools play an important role in the understanding of simulated physical data [2], exploring the data produced and debugging the simulation itself. This data is produced by various scientific simulations and is analyzed by placing the data in some visual context. Among these scientific simulations are computational fluid dynamics, molecular dynamics and so forth. Nowadays, many scientific visualization tools can be used in a variety of ways to visualize data as heat maps, contours, isosurfaces, three-dimensional and unstructured meshes. One important aspect these tools must take into account is how *fast* the tool can process the user's command or data and produce a visual image [3].

As the demand for simulation resolution increases, the amount of data produced by simulation also increases [4], i.e the data that needs to be processed by the visualization

[1]This work was supported by the Lynn and William Frankel Center for Computer Science. Computational support was provided by the NegevHPC project [1].

tool, combined with the complicated ways to represent the data, leads to long response time, thus harming the user's experience. One approach visualization tools may take to shorten the response time is introducing the distributed-memory parallelism schemes [5] to the tool, such as done in *VisIt* [6], *ParaView* [7], *Tecplot* [8] and many more.

The distributed-memory model consists of multiple processes with different address space, that may coordinate in some manner to perform a task. These processing units may reside on completely different computer nodes using MPI (Message Passing Interface) [9] to communicate with one another. With this approach, data is automatically read, processed, and if needed rendered, in a distributed manner. Thus, dividing the workload and the data between the processing units and decreasing the response time results in an improvement of the HCI (Human-Computer Interaction).

## 2. Previous Work

Many current scientific visualization tools ease the work of the scientist. Some of these visualization tools provide the graphical aspect and it is in the scientists duty to write the data-parsing and plotting methodology. These tools are fast in response time and flexible in their options (as the scientist has full command on how to plot). However, these tools are less scalable, even at the presence of multi-core hardware, as they are work in a serial fashion. For example, *MATLAB* [10], which is a numerical computing environment and programming language developed by MathWorks. Among *MATLAB*'s various features are multi-dimension plotting, contour generation, histograms, vector fields and more. However, besides the parallel computing toolbox [11], *MATLAB* is a serial software. In the similar well-known open-source mimic, *Matplotlib* [12] is a Python plotting library with a similar syntax to *MATLAB*'s plotting commands. *Matplotlib* is capable of two-dimension plots with different options such as color-maps, histograms and more. Nevertheless, is still not essentially parallel.

There are many more tools with the same rationale - focusing on providing a fast-response graphic visualization of data such as *Gnuplot* [13] , *GNU Octave* [14] etc. However, as previously mentioned, these tools require manual parsing of the data, and specifically producing (via command-line or code) the wanted plot. To further ease the job of the scientist, some scientific visualization tools provide the processing and parsing of the data, and already include built-in scientific-relevant options such as contours, iso-surfaces, color-maps, mesh generation and more. In contrary to the previous tools, these introduced with scalable parallelization schemes to the parsing and rendering stages. For example, *VisIt* is an open-source, scalable, interactive, parallel up-to three-dimensional visualization tool developed by Lawrence Livermore National Laboratory (LLNL). *VisIt* supports multiple operating systems such as Unix, Windows and Mac and multiple scientific data formats. Users can manipulate and change their data by applying different operators and mathematical expressions on the data, save their results and images and even produce animations. Moreover, users can use the tool to have a better understanding of their data and even use it to debug their code. *VisIt*'s parallelism scheme [15] is based on the distributed-memory model. The most frequent parallel mode in *VisIt* is where data is partitioned and distributed over the different processing units - the MPI tasks. Each MPI task is responsible for the analysis of the data, i.e on the different operators applied to the data. Additionally, the MPI tasks are responsible for the rendering of its chunk of data and the resulting images from each task are composite together. In most of the cases, the parallelism behind *VisIt* is in an embarrassingly parallel fashion, meaning there is no need for communication between the processing units. However, in cases where the data needs to be shared among the processing units due to streamlining calculation or volume rendering, the processing units will coordinate and communicate via the MPI API. Another scalable, parallel visualization tool that is based on *VisIt* and extends its parallelization scheme is *VisIt-OSPRay* [16]. The rationale behind this system is to visualize hundreds of gigabytes and even terabytes of data efficiently on regular processors (Intel Xeon [17]) and co-processors (Intel Xeon-Phi [18]). *VisIt-OSPRay* implements a hybrid parallelization scheme, that includes both a distributed-memory model (processes) and a shared-memory model (threads). In the final stage of the visualization - the final image composition, the composition in the same node will be done by using threads thus, minimizing communication overhead and using the shared memory between the threads while the composition between different nodes will be done via MPI. A similar tool to *VisIt* is *ParaView* [19]. This tool is an open-source, multi-platform data analysis, parallel up-to three-dimensional visualization tool developed by Los Alamos National Laboratory. *ParaView*, similarly to *VisIt*, was developed to visualize both small datasets which are suited for laptops, personal computers, etc. and large scale datasets that are suited for HPCs. *ParaView* was designed in layers: The most basic layer is the visualization toolkit (VTK), which provides the data representation, algorithms and the connection between the two. The second layer of *ParaView*'s design is the parallel extension to the first layer (VTK). The parallel layer allows the execution of the algorithms on shared and distributed memory machines. The third layer is the graphical user interface (GUI) itself which provides the transparency of the visualization and the rendering. *ParaView* supports many options such as contours and isosurfaces, vector fields and more. *ParaView*'s parallelsim scheme [7] is based on the distributed-memory model, and works in the same work fashion as *VisIt*. *ParaView* implements its parallelism scheme the same way *VisIt* uses MPI. Each MPI task reads a portion of the data, processes it, and if needed will render the data in a parallel manner. The communication between the MPI tasks is handled by the internal modules, i.e every algorithm is implemented in a parallel manner and contains the communication schemes. Figure 1 presents a common way to implement a visualization tool with MPI.

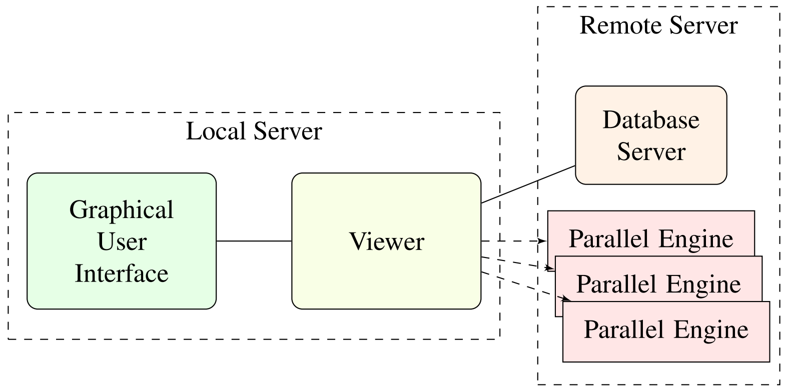


**Figure 1.** Visualization tool scheme with distributed memory parallelism in *ParaView* and *VisIt*.

## 3. Problem Formulation

As seen in section 2, the distributed-memory model is a common visualization approach tools take to enhance the user's experience. However, the use of distributed computing and the work-fashion in those tools has some major disadvantages [20]:

- **Distributed Computing Overhead:** When executing commands on the whole dataset, the processing units need to communicate and synchronize with one another - a situation which creates communication overhead. Furthermore, executing a parallel visualization on a Symmetric multiprocessing (SMP) or Non-uniform memory access (NUMA) architecture leads to unnecessary communication overhead, as there is no need for such communication as they can all share the same physical memory.
- **Slow HCI:** In many cases when there is a need for *immediate* visualization, even of relatively large amount of data [$\sim$ second], the data can actually fit on a single socket [21] - both in terms of memory and computational power - and thus the distribution of the domain on many nodes in order to utilize more cores is counter-effective. Furthermore, the creation of MPI processes even on the same socket is longer than spawning threads, which can be done in the shared-memory model.
- **Non-Optimized Resource Utilization:** As the current distributed visualization tools use a parallel engine that launches the processes to the nodes throughout all of the tool usages - regardless of actual service - it also implies that the computational resources are in many cases idle but still allocated.

However, in the past years, multi-core architectures become more and more common in desktops, laptops, mobile, etc [22]. The multi-core architecture provides a way for software developers to introduce parallel schemes such as the shared-memory parallelization [23], thus decreasing the software's response time and allowing more complicated operations. The shared-memory model consists of processing units that share the same address space, allowing the processing units to exchange data and communicate with minimal overhead.

As the number of those cores and the amount of their RAM increase [24], the need for distributing the data on different processing units decreases. Thus, introducing shared-memory model parallel schemes can lead to faster response time and optimized resource management [25]. In regard to scientific visualization, distributed tools were created first and foremost for complicated and extreme high-resolution simulations and are suited for the HPC arena. On the contrary, simplified visualization tools were created for visualization of lower to medium resolution than the latter and are suited for desktops computers or single socket of NUMA computers. This distinction is very common in the sciences work fashion, and as such of great interest to be improved in both cases. Due to the increased usage of multi-core architectures in all computational architectures since the year of 2005 [22], the gap between these two types of visualization tools can be reduced by introducing shared-memory parallel schemes to the tools, and most urgently to the desktop and single socket NUMA suited ones. Therefore, we created *GraphiX* a Fast HCI SMP scientific visualization tool.

## 4. GraphiX

*GraphiX* is a fast HCI-suited scientific visualization tool for both SMP and NUMA up to three-dimension. GraphiX supports several ways to visualize the data such as volume mesh representation, color-maps, contours, x-axis/y-axis mirroring, presenting data related to the mesh and more.

*GraphiX* graphical utility is based on Gnuplot [26]. We chose *Gnuplot* because it is open-source and written in C and C++ while *MATLAB* is a proprietary software and not open-source. Additionally *MATLAB* cannot be parallelized in shared or distributed memory thus, choosing this tool as the graphical utility is impractical. Although *Matplotlib* is open-source and can be parallelized, *Gnuplot* is more suited for OpenMP.

*Gnuplot* [13] is an interactive, multi-platform up-to three-dimensional visualization tool. Unlike *VisIt* and *ParaView*, *Gnuplot* is not parallel in any way and is a command-line driven visualization tool, rather than GUI driven. Nevertheless, it is a fast visualization tool written in C/C++ and was created to allow scientists to visualize different functions and data interactively and non-interactively. It also supports different interactive screen displays such as Qt, wxWidgets, x11 or system-specific. Users can also direct their plots to different file types such as pdf, eps, gif, jpeg, LaTeX, svg and more.

*GraphiX* GUI is written in Python and the heavy computational operations such as mesh creation, contour line calculations, and color-map interpolations are written in *Cython* [25]. *Cython* is a programming language designed to give C-like performances while maintaining the simplicity of Python syntax. In cases of large data and many operations, we used OpenMP under *Cython*.

Parallelism schemes were introduced to two main modules inside *GraphiX*. The first module, as discussed above, is responsible for reading and parsing the initial data (creating the polygon's coordinates, contours calculations etc.), and creating the *Gnuplot*'s commands that will later produce the visual image. For the second module, Gnuplot's source code was partially introduced with shared-memory parallelism via OpenMP [23], specifically the source code that creates the polygons (the mesh and color-map) which is the most time-consuming part as will be discussed in section 5. The rationale behind introducing *Gnuplot* with OpenMP is explained in section 3, which is minimizing the overhead and optimizing the use of the common modern architecture - multi-core processing.

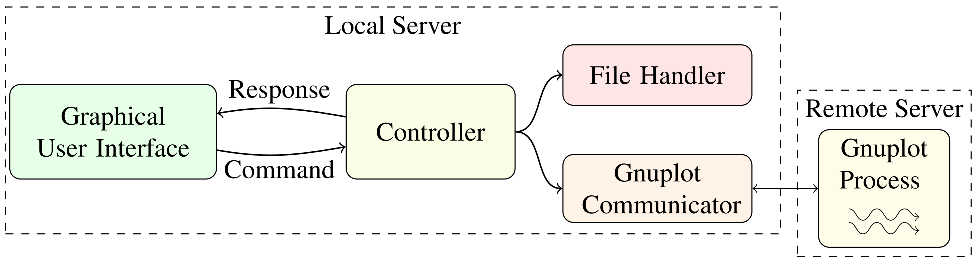
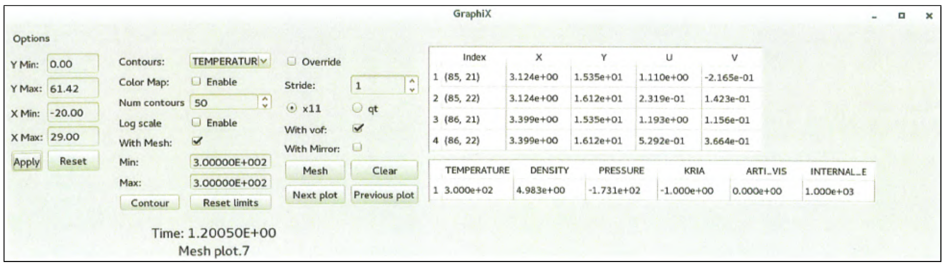


**Figure 2.** GraphiX's work-flow.

*GraphiX* workflow consists of four main modules: GUI, Controller, File Handler and Gnuplot communicator along with *Gnuplot*'s source code. Figure 2 illustrates the main modules and work-flow.

## *4.1. Graphic User Interface*

The Graphic User Interface module handles the user's requests and interaction. As seen in figure 3 and in section 4, *GraphiX* can produce contours, color-maps, axis mirroring, along with providing the cell's physical data when clicked. The GUI is written in Python with PyQt [27] as the visualization kit. The GUI contains a window (the *Viewer*) that is connected automatically to a *Gnuplot* process. This means that when *Gnuplot* displays the plot it is automatically drawn on to the *Viewer*. Figure 3 presents the GUI part, which includes all the different options such as contours, skipping to the next plot, showing the physical data of a cell and more. In figure 4 different simulations are plotted on the GUI, with *Gnuplot* as the graphical utility. Among the plots are Sedov-Taylor simulation (blast wave) mesh presentation and pressure color-map, and Rayleigh-Taylor instability mesh presentation and density color-map.



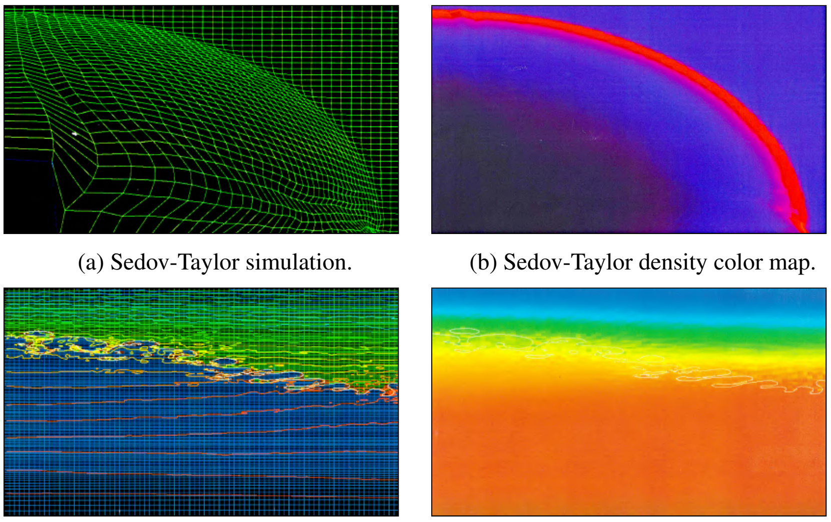
| | |
|---|---|
| Y Min: | 0.00 |
| Y Max: | 61.42 |
| X Min: | -20.00 |
| X Max: | 29.00 |
| Contours: | TEMPERATUR |
| Color Map: | ☐ Enable |
| Num contours | 50 |
| Log scale | ☐ Enable |
| With Mesh: | ✓ |
| Min: | 3.00000E+002 |
| Max: | 3.00000E+002 |
| Override | ☐ |
| Stride: | 1 |
| ● x11 | ○ qt |
| With vof: | ✓ |
| With Mirror: | ☐ |

Time: 1.20050E+00
Mesh plot.7

| | Index | X | Y | U | V |
|---|---|---|---|---|---|
| 1 | (85, 21) | 3.124e+00 | 1.535e+01 | 1.110e+00 | -2.165e-01 |
| 2 | (85, 22) | 3.124e+00 | 1.612e+01 | 2.319e-01 | 1.423e-01 |
| 3 | (86, 21) | 3.399e+00 | 1.535e+01 | 1.193e+00 | 1.156e-01 |
| 4 | (86, 22) | 3.399e+00 | 1.612e+01 | 5.292e-01 | 3.664e-01 |

| | TEMPERATURE | DENSITY | PRESSURE | KRIA | ARTI_VIS | INTERNAL_E |
|---|---|---|---|---|---|---|
| 1 | 3.000e+02 | 4.983e+00 | -1.731e+02 | -1.000e+00 | 0.000e+00 | 1.000e+03 |

**Figure 3.** Graphix Graphic User Interface.



(a) Sedov-Taylor simulation.

(b) Sedov-Taylor density color map.

(c) Rayleigh-Taylor pressure contours.

(d) Rayleigh-Taylor pressure color map.

**Figure 4.** GraphiX visualization on different simulations.

## 4.2. *Controller*

The Controller module is the main module that connects all the other modules. The module stores all the data (coordinates, physical data such as pressure and temperature and more) of the plot. Additionally, the creation of contour lines, color-map and the mesh is done in this module. As mentioned, *GraphiX* is written in Python. Due to this, operations that require heavy calculations such as contours, mesh and color-map creation may lead to long response time (compared to low-level programming languages such as C). To cope with this problem and speedup this process, these modules were written in *Cython*.

When a user executes a command the GUI sends the request to the Controller. The Controller then executes the appropriate action via the File Handling module. Finally, the Controller will send the file name, and if needed more data, to *Gnuplot*'s communicator (see below 4.4) that will later send the appropriate execution command to *Gnuplot*.

### *4.3. File Handling*

The File Handling module is the module that handles two main parts. The first part reads and parses (if needed) the user's data. The second part creates temporary files that contain the commands *Gnuplot* will later execute. For example, producing a mesh in *GraphiX* is usually done via *Gnuplot*'s polygon objects. To create the appropriate polygon objects the coordinates are parsed and connected in some manner. Afterward, a temporary file is created where each line defines a *Gnuplot* polygon object. Finally, *Gnuplot*'s *load* command is executed on the temporary file and the mesh is presented on the *Viewer*. Currently *GraphiX* supports only the VTK input file format. However, it is possible to extend this part and support additional formats.

### *4.4. Gnuplot Communicator and Parallel Source Code*

*Gnuplot Communicator* and Source Code module consists of two separate modules that are strongly connected. Because *Gnuplot* is a command-line based visualization tool, the first module, the communicator, opens a *Gnuplot* process shell (command-line) and is in charge of sending the *Gnuplot* commands such as the *load* command, the *plot* command etc. Furthermore, the module receives messages back from the *Gnuplot* process. For example, clicked coordinates that provides the cell physical data.

The second module consists of *Gnuplot*'s modified, OpenMP parallel source code. It was found that producing color-maps is the most computational intensive and time-consuming part in *Gnuplot*. Producing color-maps is done by creating polygons with some value that represents its' color. *Gnuplot* creates polygons by creating a linked-list of objects (objects can be polygons, rectangles etc.). Each time a new polygon is created it is added in some manner to the linked list. Thus, to speed-up the process of creating the polygons the function *load_file*, which in fact creates the linked-list of polygons, was introduced with shared-memory parallelism - OpenMP. As the *Gnuplot load* command is executed to produce the color-map, the parallelism scheme was introduced to the function *load_file* that parses each line of the file and creates the linked-list of polygons accordingly. The parallelism scheme is based on dividing the linked-list to the working threads, i.e each thread that is spawned by the OpenMP run-time environment is responsible on parsing the specific file line and eventually creating a polygon that is part of its own *private* and independent linked-list. Finally, once all the threads finish creating their linked-lists, the *master-thread* links them together. In NUMA architectures the OpenMP run-time environment may execute the threads on processing units (cores) that reside on different sockets, resulting in more frequent false-sharing [28], thus reducing the speedup gained. To overcome this [29], thread affinity and placement were included in the parallelization schemes using OpenMP 4.5 [30].

## 5. Parallelisem and Performances Evaluation

Evaluating *GraphiX*'s parallelism scheme (or in other words *Gnuplot*'s modified parallel source code) in terms of speed and memory scaling was done by creating a file that contains many polygon-objects and executing *GraphiX* color-map command, which as mentioned is the most time consuming operation (in *Gnuplot*'s source code it is the *load_file* function). First, evaluating *GraphiX*'s thread-scaling capability was tested with {1, 2, 4, 8, 16, 32} threads with a 500MByte file which is roughly 3,850,000 polygons. *GraphiX* was executed on the NUMA architecture with two different options of a new OpenMP 4.5 feature, the thread affinity with the options of *close* and *spread*. The thread affinity *close* option spawns new threads as close as possible to the master thread, thus utilizing the cache-usage and local memory, while the *spread* option spreads the threads across the machine (on different sockets) as much as possible, thus utilizing the memory-bandwidth. As one can see from figure 5, spawning threads close to the master thread yields better speedup, as the algorithm behind the creation of the polygons is better utilized with cache-sharing. The parallelism scheme (with *close*) scales well until 8 cores. Although there is a slight speedup with 16 threads compared to 8, it was found that 8 threads on the NUMA architecture yields the optimal results in terms of performance per dollar. It is also notable that nowadays most desktops and laptops have 8 to 16 cores in a SMP architecture. This further indicates that the parallelism scheme is suited for SMP architecture as *close* ensures the threads are created within the same socket. Additionally, the two trends intersect at 32 threads as this is the number of cores on the machine, thus there is no meaning for *close* or *spread* as they perform in the the same way. Furthermore, we included the well-known theoretical Amdahl's law graph to demonstrate that the speedup of *Gnuplot* total execution is almost the same, as the function *load_file* takes approximately 99% of the total execution time.

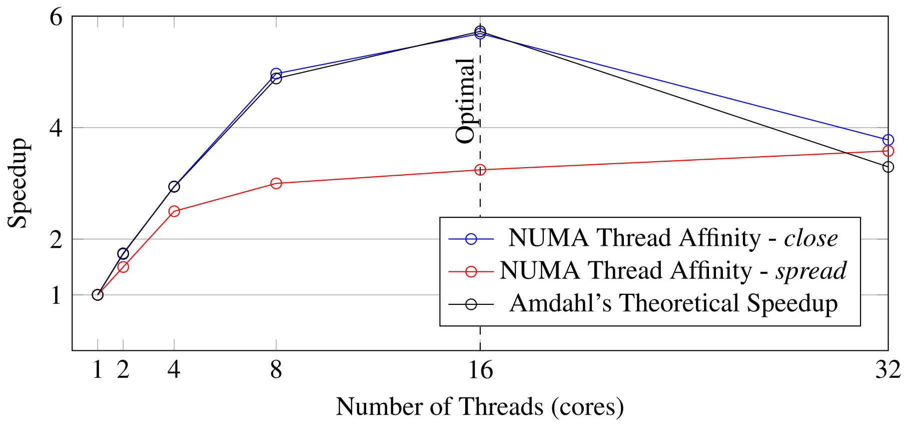


**Figure 5.** Speedup of the *GraphiX* color-map creation with 500MByte file compared to serial *GraphiX*.

To evaluate *GraphiX*'s memory-scaling capabilities, similarly to evaluating the thread-scaling capability, *GraphiX* color-map option was tested with files of sizes 100MByte (∼770,000 polygons), 500MByte (∼3,850,000 polygons), 1GByte (∼7,700,000 polygons), 2GByte (∼15,200,000 polygons) and 4GByte (∼30,000,000 polygons).

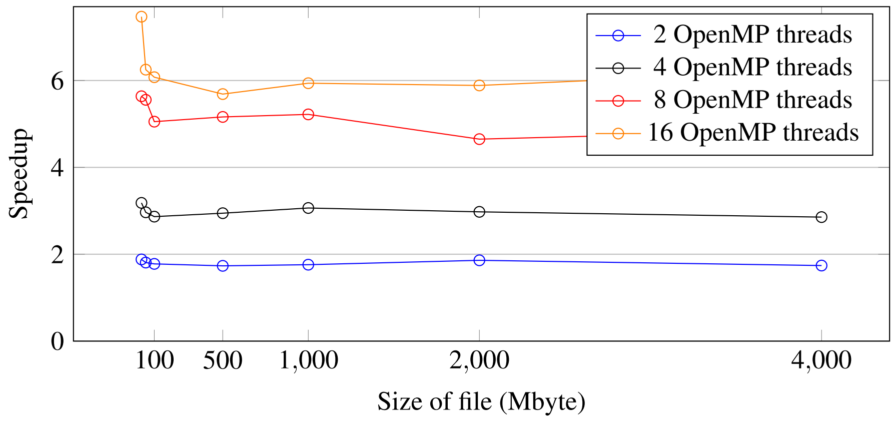
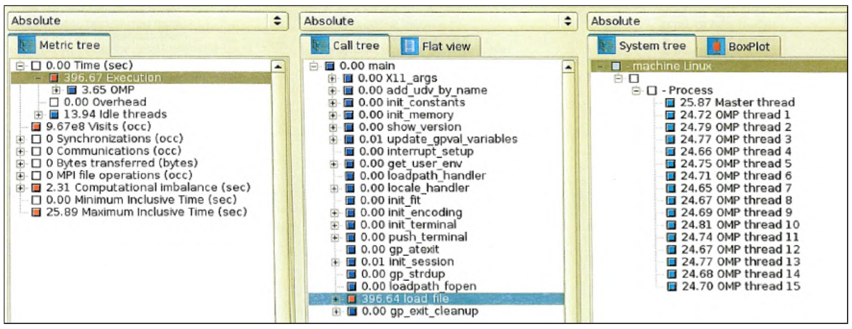


**Figure 6.** Speedup of the *GraphiX* color-map creation over different file sizes compared to serial *GraphiX* execution with thread affinity *close*.

To evaluate *GraphiX* workload between threads, it was compiled with *Scalasca* [31] - a tool that analyzes and measures a programs runtime behavior. One of the features in *Scalasca* is to identify performance bottlenecks - specifically in our case, the work-balance between the threads - and to verify that there is no bottlenecks in Gnuplot's modified source code. As seen in figure 7 on the third column, the execution time of each thread in the OpenMP region is nearly the same, pointing out that the work-balance between all 16 threads is the same.



```
0.00 Time (sec)
  396.67 Execution
    3.65 OMP
  0.00 Overhead
  13.94 Idle threads
9.67e8 Visits (occ)
0 Synchronizations (occ)
0 Communications (occ)
0 Bytes transferred (bytes)
0 MPI file operations (occ)
2.31 Computational imbalance (sec)
0.00 Minimum Inclusive Time (sec)
25.89 Maximum Inclusive Time (sec)
```

```
0.00 main
  0.00 X11_args
  0.00 add_udv_by_name
  0.00 init_constants
  0.00 init_memory
  0.00 show_version
  0.01 update_gpval_variables
  0.00 interrupt_setup
  0.00 get_user_env
  0.00 loadpath_handler
  0.00 locale_handler
  0.00 init_fit
  0.00 init_encoding
  0.00 init_terminal
  0.00 push_terminal
  0.00 gp_atexit
  0.01 init_session
  0.00 gp_strdup
  0.00 loadpath_fopen
  396.64 load_file
  0.00 gp_exit_cleanup
```

```
- machine Linux
  
    - Process
      25.87 Master thread
      24.72 OMP thread 1
      24.79 OMP thread 2
      24.77 OMP thread 3
      24.66 OMP thread 4
      24.75 OMP thread 5
      24.71 OMP thread 6
      24.65 OMP thread 7
      24.67 OMP thread 8
      24.69 OMP thread 9
      24.81 OMP thread 10
      24.74 OMP thread 11
      24.67 OMP thread 12
      24.77 OMP thread 13
      24.68 OMP thread 14
      24.70 OMP thread 15
```

**Figure 7.** *GraphiX* Parallelisem with 16 threads, exhibiting near perfect load-balancing.

## 6. Conclusions

As the trend of multi-core architectures is getting more and more popular, the introduction of shared-memory parallelism scheme to software is necessary in order to utilize this architecture. Scientific visualization tools are no exception to this introduction, thus *GraphiX*, a fast two/three dimension visualization tool suited for every multi-core architecture, was created. The most time consuming option found in GraphiX was the color-map, thus OpenMP directives were introduced to the tool. As shown in section 5 the parallelism scheme's work-balance is perfect and scales well with both the problem size and the number of threads, and achieve speedup of ∼5.6 at peak with 16 cores.

## References

[1] NegevHPC Project. `www.negevhpc.com`. [Online].

[2] Johannes Kehrer and Helwig Hauser. Visualization and visual analysis of multifaceted scientific data: A survey. *IEEE transactions on visualization and computer graphics*, 19(3):495–513, 2013.

[3] Melanie Tory and Torsten Moller. Human factors in visualization research. *IEEE transactions on visualization and computer graphics*, 10(1):72–84, 2004.

[4] Leading examples of using VisIt at scale. `visitusers.org@VisIt_top_50`. [Online].

[5] Ewa et al. Deelman. The future of scientific workflows. *The International Journal of High Performance Computing Applications*, 32(1):159–175, 2018.

[6] VisIt homepage. `wci.llnl.gov/simulation/computer-codes/visit`. [Online].

[7] James Ahrens, Berk Geveci, and Charles Law. Paraview: An end-user tool for large data visualization. *The visualization handbook*, 717, 2005.

[8] WA Bellevue. Tecplot. 360 2018 users manual, 2018.

[9] Marc Snir, Steve Otto, Steven Huss-Lederman, Jack Dongarra, and David Walker. *MPI–the Complete Reference: The MPI core*, volume 1. MIT press, 1998.

[10] MATLAB. `https://www.mathworks.com/products/matlab.html`. [Online].

[11] Parallel Computing Toolbox. `www.mathworks.com/products/parallel-computing.html`. [Online].

[12] John D Hunter. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(3):90, 2007.

[13] Gnuplot homepage. `www.gnuplot.info`. [Online].

[14] John W Eaton, David Bateman, and Soren Hauberg. *GNU Octave manual*. Network Theory Ltd. Bristol, UK, 2002.

[15] Hank Childs. Visit: An end-user tool for visualizing and analyzing very large data. 2012.

[16] Qi Wu, Will Usher, Steve Petruzza, Sidharth Kumar, Feng Wang, Ingo Wald, Valerio Pascucci, and Charles D Hansen. Visit-ospray: Toward an exascale volume visualization system. In *EGPGV*, pages 13–23, 2018.

[17] Intel Xeon website. `https://www.intel.com/content/www/us/en/products/processors/xeon.html`. [Online].

[18] George Chrysos. Intel® xeon phiâĎć coprocessor-the architecture. *Intel Whitepaper*, 176, 2014.

[19] ParaView website. `https://www.paraview.org/`. [Online].

[20] Marc Buffat, Anne Cadiou, Lionel Le Penven, and Christophe Pera. In situ analysis and visualization of massively parallel computations. *The International Journal of High Performance Computing Applications*, 31(1):83–90, 2017.

[21] Christoph Lameter et al. Numa (non-uniform memory access): An overview. *Acm Queue*, 11(7):40, 2013.

[22] David Geer. Chip makers turn to multicore processors. *Computer*, 38(5):11–13, 2005.

[23] Barbara Chapman, Gabriele Jost, and Ruud Van Der Pas. *Using OpenMP: portable shared memory parallel programming*, volume 10. MIT press, 2008.

[24] Comparison Charts for Intel Core Desktop Processor Family. `www.intel.com/content/www/us/en/support/articles/000005505/processors.html`. [Online].

[25] E Wes Bethel, Hank Childs, and Charles Hansen. *High Performance Visualization: Enabling Extreme-Scale Scientific Insight*. CRC Press, 2012.

[26] Thomas et al. Williams. gnuplot 5.3. 2017.

[27] Mark Summerfield. *Rapid GUI programming with Python and Qt: the definitive guide to PyQt programming*. Pearson Education, 2007.

[28] Josep Torrellas, HS Lam, and John L. Hennessy. False sharing and spatial locality in multiprocessor caches. *IEEE Transactions on Computers*, 43(6):651–663, 1994.

[29] Christian Terboven, Dirk Schmidl, Henry Jin, Thomas Reichstein, et al. Data and thread affinity in openmp programs. In *Proceedings of the 2008 workshop on Memory access on future processors: a solved problem?*, pages 377–384. ACM, 2008.

[30] Ruud Van der Pas, Eric Stotzer, and Christian Terboven. *Using OpenMP The Next Step: Affinity, Accelerators, Tasking, and SIMD*. MIT Press, 2017.

[31] Markus Geimer, Felix Wolf, Brian JN Wylie, Erika Ábrahám, Daniel Becker, and Bernd Mohr. The scalasca performance toolset architecture. *Concurrency and Computation: Practice and Experience*, 22(6):702–719, 2010.