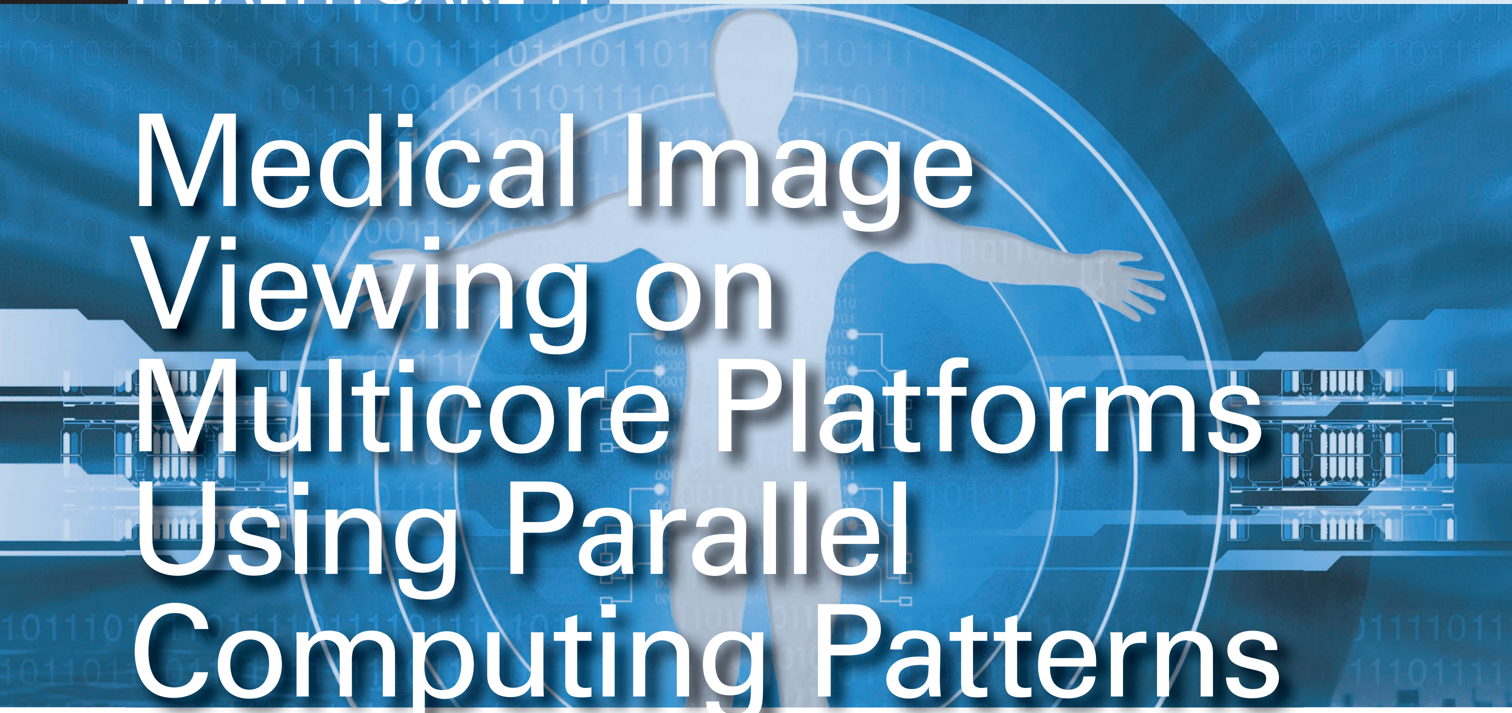
Medical Image Viewing on Multicore Platforms Using Parallel Computing Patterns



Yang-Ming Zhu and Steven M. Cochoff, *Philips Healthcare*

The parallel programming community has accumulated a significant amount of experience in the form of software patterns. Those patterns can be used in medical imaging applications on multicore platforms to improve startup time, runtime throughput, and algorithm reliability.

Moore's law predicts that the density of transistors on an integrated circuit increases exponentially, doubling approximately every two years. This has been true since the 1960s and should remain true for at least the next decade. Computing power has also been doubling every two years. To continue fulfilling this prediction, the semiconductor industry keeps reducing the line width of the lithography and increasing the clock speed of processors. Consequently, power consumption and heat dissipation have become a problem. To overcome this, vendors are using multicore or many-core architectures—having multiple CPUs work together on a single chip to improve performance by handling more work in parallel.

As multicore processors become ubiquitous, we'll have increasing computing power at our disposal. It makes sense to exploit the improved performance in medical image applications since they tend to be computationally demanding, and parallel programming can help. On a single computer system and in the same process address space, this primarily means using multithreading techniques. Although software and hardware vendors provide threading toolkits or development platforms, multithreaded programming is generally regarded as difficult. This is because most programmers were trained in sequential programming, and data and task parallelisms aren't always easy to identify. Moreover, numerous legacy applications exist, and it would be hard (if not impossible) to exploit multicore CPU advancements on those applications without significant rewriting.

The parallel programming community, on the other hand, has accumulated a significant amount of experience, which it has documented as *software patterns* for parallel programming. Software patterns codify the collective knowledge and experience of software experts. They provide proven solutions to repeated design problems in specific contexts, balancing many conflicting constraints. These patterns have been successfully applied to design new software architecture, restructure existing software to improve performance, and port legacy applications to parallel platforms.

We discuss a few parallel programming patterns and illustrate how we've applied them in our own practice. As performance is often vital to an application's success, we focus on those software patterns that can improve application performance. Although there are many performance metrics to consider, we focus on response time, which includes application startup time and runtime response. In addition, we illustrate how to use multicore and multithreading technologies to improve algorithm reliability.

Software Patterns for Parallel Programming

Kent Beck and Ward Cunningham first introduced the concept of a software pattern.[1] In a nutshell, software patterns provide not only proven solutions to repeated design problems in specific contexts, but also vocabularies for design, communication, and teaching. Design patterns became popular after the "Gang of Four" published their seminal book,[2] and a few authors worked in parallel to document design patterns about the same time.[3] In the beginning of the pattern movement, authors documented generic patterns. Currently, they're moving toward documenting technology-dependent, application domain-specific patterns. Nowadays, patterns are used to document almost every aspect of software development, from requirement engineering to the software development process itself.

Tim Mattson and his colleagues published a book on patterns for parallel programming,[4] which captures the experience and knowledge of parallel programming experts over the past few decades. They discuss software patterns in four design spaces (finding concurrency, algorithm structure, supporting structure, and implementation mechanisms). Although the book was written for generic parallel programming on a variety of platforms ranging from symmetric multiprocessors (SMP) and clusters to heterogeneous distributed systems, the ideas apply to parallel programming on multicore platforms. We discuss a few high-level patterns that we found useful in our own practice.

Decomposition Patterns

The key to parallel computing is to expose and exploit the concurrency in the problem or application. Here, concurrency means activities or tasks that can be executed by an execution unit (either a thread or a process) in parallel. Typically, there are two approaches to discovering concurrencies: *task decomposition* and *data decomposition*. Choosing to decompose a program into tasks or data is to some extent arbitrary, but the approach you choose takes a primary role in driving the design. In the end, you almost always need to do both, since a task decomposition usually implies a data decomposition, and vice versa.

Task Decomposition. In task decomposition, you decompose a problem into tasks. For the sake of computational efficiency, it's best if the tasks are independent so you can execute them independently and safely. If the tasks are interdependent, you must employ various synchronization primitives such as locks, critical sections, mutexes, or semaphores to ensure that the tasks execute correctly. When a task is implemented as a function call, this becomes *functional decomposition*. Medical imaging algorithms sometimes iterate, and if each iteration is independent, we can decompose the loop into different tasks (this is known as *loop-splitting*). Quite often, we change an existing sequential program to a parallel counterpart. Function calls or loops are common places to discover concurrency.

Data Decomposition. In data decomposition, you decompose the program data into chunks, and you can update each data chunk in an independent execution unit. Once the data is decomposed, you can also identify the tasks working on the data. In medical image viewing, you can partition the image to be rendered into different segments. Creating and updating one portion can be one simple computational task. On a multicore platform, we must perform data decomposition carefully to improve the data locality. To reduce cache-miss, all threads at a given time should work on the data that resides in the cache. To balance the load, the data chunks should be the same relative size if the computation complexity of the data is comparable.

Program Structuring Patterns

You can organize a parallel program by following one of three alternatives: task decomposition, data decomposition, or data flow among tasks, all of which have associated software patterns.[4] The simplest patterns are *task parallelism*, *data parallelism*, and *pipeline*. Because these patterns help structure the program at a higher level, they're considered architectural patterns. On multicore platforms, you often can parallelize only isolated portions of a sequential program. You can use the patterns to restructure the isolated portions as well.

Task parallelism. In the case of task parallelism, you base the software design directly on the decomposed tasks. It's particularly useful when you can arrange tasks into an array. There are three key elements in this design: *tasks*, *dependencies*, and *scheduling*.

Ideally, the number of tasks should be equal to or larger than the number of execution units (cores), and each task should have a sizable amount of computational complexity to make the effort worthwhile.

There are two kinds of dependencies: *order dependency* and *data dependency*. If there's an order dependency among tasks, you must satisfy the order constraint; otherwise, the result might not be correct. Scheduling must consider that dependency. When different tasks read or write the shared data, there's a data dependency. Sometimes, you can remove the data dependency through code transformation, or you can separate it using data replication.[4] When there's no interdependency among tasks, it's called *embarrassingly parallel*. Not surprisingly, many real life tasks are embarrassingly parallel problems. When there's dependency among tasks, you should synchronize them. You can use synchronization primitives such as locks, semaphores, monitors, or condition variables.

The primary concern of task scheduling is to balance the load on execution units. In static scheduling, you can assign the tasks to a fixed execution unit. This is useful when you know the number of tasks, each task is equally computationally expensive, and each execution unit has the same power. Otherwise, dynamic scheduling is a better choice from a load-balancing perspective.

Data parallelism. Data parallelism is concerned with organizing a program around a data structure that has been decomposed into chunks updated by different tasks. If the data structure is linear, such as an array, the data structure is partitioned into regular, contiguous chunks. As mentioned earlier, data decomposition implies task decomposition, and each task updates its local data chunk. If the task doesn't need data from other chunks, this degenerates to the embarrassingly parallel case. In a general case, there's data sharing among tasks. On a shared memory system, such as a multicore computer, the shared data must be protected to avoid errors and inconsistencies.

Similar to task parallelism, the data parallelism pattern has some key elements in its generic form: data decomposition, shared data handling, data updating, data distribution, and task scheduling. Mattson and his colleagues discuss these elements mainly for message-passing platforms.[4] We focus on shared memory environments such as multicore.

When decomposing the linear data structure, you must carefully consider the size and shape of the decomposition. A simple way to choose the size is to have all execution units handle the same amount of data. If the computation is equally expensive, they're relatively balanced. The decomposition should minimize the data chunk's surface area—that is, minimize data sharing among tasks.

Because the cache is always limited, data locality is another concern. The data decomposition should seek to reduce cache misses. Each data segment can be a consecutive chunk and thus can be contiguous in memory, or it can be interlaced from the original data. Processing shared data (data on the chunk's surface) requires synchronization. Data updating is trivial and similar to a sequential program for nonshared data. If the number of chunks equals the number of execution units, the data distribution and task scheduling are straightforward.

Pipeline. A good analogy for the pipeline pattern is doing laundry. To save time, we want the washer and dryer to work at the same time; while the washer is working on the second load, the dryer works on the first load. For the analogy, the washer and dryer are two execution units, and the clothes are the data to be processed. The pipeline pattern is generally useful when the flow of data among tasks is regular and one-way, and the order of tasks is static.

The laundry analogy captures the essence of this pattern. The processing has multiple, ordered stages (tasks), and the data flows through those stages linearly. There's a setup time to fill the pipe and a teardown time to drain it, and ideally setup and teardown times should be small compared to the whole process time. To fully use the execution units, each stage should have the same processing time. Compared to sequential processing, the speedup is equal to the number of stages.

Programming Support and Other Options. The level of support for these patterns differs depending on the programming environment. You can use the "parallel for" directive in OpenMP for both task and data parallelism (the "for" construct divides work among threads, and the "parallel" construct creates threads). In the Microsoft .Net Framework, you can use either threads (thread pool) or asynchronous calls to implement task parallelism. The .Net Framework 4 introduces a task construct and, with the task parallel library, you implicitly schedule tasks.[5] The parallel language integrated query (PLINQ) is directly based on data parallelism.[5]

There are a few other patterns that you can use to structure a parallel program, such as *divide and conquer*, *recursive data*, and *event-based coordination*. When implementing these patterns, you can use low-level patterns or idioms such as single program multiple data (SPMD), asynchronous call, loop parallelism, shared data, and fork/join. These idioms are tied to specific programming languages and development environments.

Examples Using Parallel Software Patterns

Here we discuss how we've used parallel patterns to improve the performance of the FusionViewer—an application used for multimodality image viewing, registration, and fusion.[6] The current version is developed with Microsoft .Net Framework 3.0 and C#, which is a programming environment that constrains how we implement the parallel patterns. The performance figure is measured on a Dell T7400 workstation (unless stated otherwise), which has two quad-cores (Xeon X5450 3 GHz), 12 Mbytes L2 cache, and 8 Gbytes of RAM. The operating system is Windows XP, the 64-bit professional edition. The numbers reported are an average of three independent runs.

Note that the design we discuss also works for SMP platforms. (We've done similar work on a Sun UltraSparc III, which is an SMP with 2 CPUs.) We sought to keep the design independent of specific multicore architectures and found that this leads to better portability and flexibility. Two example multicore architectures are the Intel chipset, where all cores share a common L2 cache, and the AMD chipset, which has an individual cache for each core. In general, it's not favorable to tie the implementation to a particular architecture unless the need is strongly dictated by performance requirements. In our case, the programming model is logically the same for SMP or multicore architectures.

Also note that fast or real-time medical image processing is an active area for research and practice, and researchers have extensively exploited various hardware platforms. Mainak Sen and his colleagues have reported a technique that maps the rigid registration onto field-programmable gate array (FPGA),[7] and Jahyun Koo and colleagues implemented a tissue classification algorithm on FPGA.[8] Sungchan Park and Hong Jeong developed a VLSI architecture for stereo matching,[9] and Guillem Pratx and his colleagues used a graphics processing unit for image reconstruction.[10] However, the design we discuss has minimal dependency on the hardware platforms.

Startup Time Improvement

For the FusionViewer, the application startup (wall clock) time includes both application and patient data loading. The first step in minimizing the startup time involved profiling the application. Based on our profiling, we identified that the application is I/O bound during the startup phase as it loads many image files. Typically, this includes 200 computed tomography (CT) slice files at 500 Kbytes each and 250 Positron Emission Tomography (PET) slice files at 50 Kbytes each.

CT, PET, and PET/CT

Whenever an x-ray beam passes through tissue, its intensity decreases. After a collection of x-ray beams have passed through the body, they're called a *projection*. Using a computer to execute mathematical algorithms, we can reconstruct a cross-sectional or volume image from projections—thus the term *computed tomography* (CT).

X-ray CT essentially computes a mapping of x-ray attenuation. Different tissues have different attenuation characteristics, so a CT image conveys structural or anatomic information. CT is widely used as a diagnostic procedure to detect a tumor's presence, location, and size, and to determine whether the tumor has spread throughout the body. CT can help guide a biopsy procedure and help plan and assess treatments.

In *Positron Emission Tomography* (PET) imaging, a positron-emitting radionuclide is introduced into a patient intravenously. The radionuclide is chemically combined with compounds normally used by the body or molecules that bind to receptors or other sites of drug action. When an emitted positron meets a nearby electron, they annihilate and give off two photons traveling in opposite directions, which are detected and correlated by a coincidence detection circuit. From a complete set of integrated radioactivities obtained at different views, the radionuclide distribution within the cross-sectional slice or volume can be reconstructed. Therefore, PET and nuclear medicine, in general, visualize physiological functions or functional metabolisms. Because cancerous tumors usually are more metabolically active than normal tissues, they manifest themselves differently than normal tissue on a PET image. PET is currently playing a significant role in oncology, neurology, cardiology, as well as pharmacology.

PET imaging detects diseases at the molecular or cellular level, long before CT can detect structural changes in the tissue or organ. It's beneficial to combine functional and anatomic imaging into a single hybrid scanner (the CT image is also used for attenuation correction during PET reconstruction). Today PET is sold almost exclusively as a combined PET/CT device. For similar reasons, SPECT (single-photon emission computed tomography)/CT and PET/MR (magnetic resonance) hybrid scanners are also commercially available.

(For more information on CT and PET scans, see the related sidebar.)

To reduce the I/O time, we stored the data in a compressed format on a disk and decompressed it during loading. A simple run length encoding compression achieves an approximate 2:1 compression ratio. This compression scheme improves the overall startup time, since the time used to decompress the data is less than half of the original loading time. We treat the decompression of each image as a task (task decomposition). Alternatively, if we start with the array of images to be decompressed, the same tasks would be identified through data decomposition. Because each image is independent, the decompression tasks are also independent. We used the task parallelism pattern to implement image decompression. Because the tasks are independent, this operation is embarrassingly parallel.

With C# and the .Net Framework 3.0, threads are created per task and managed using the fork/join pattern. The number of threads is configurable, so we can vary them to evaluate how the performance scales with the number of threads. We used parameterized threads, because we needed to pass the data structure to individual threads.

The data structure contains the information on the images that the thread will process. The thread entry function is the same for all threads, which we can identify as an instance of the SPMD pattern. Given the list of images, threads must know their identity to decide which images to process—the hallmark of the SPMD pattern. We can identify the shared list of images as the shared data pattern. We divide the images in an interlaced fashion. Consequently, each thread processes approximately the same number of images, which helps with load balancing. We paid special attention to cases where the number of images wasn't divisible by the number of threads. Data locality isn't a concern here, because each thread works on its local data.

We measured the decompression time as a function of the number of threads. Figure 1 shows the result. We used three CT image series, which have 853 images, 1,706 images, and 3,412 images (each CT image had 512 × 512 pixels and each pixel had two bytes). The results show that the loading time scales with the number of threads, up to a value of four. When there are more than four or five threads, the decompression time levels off. The performance degrades

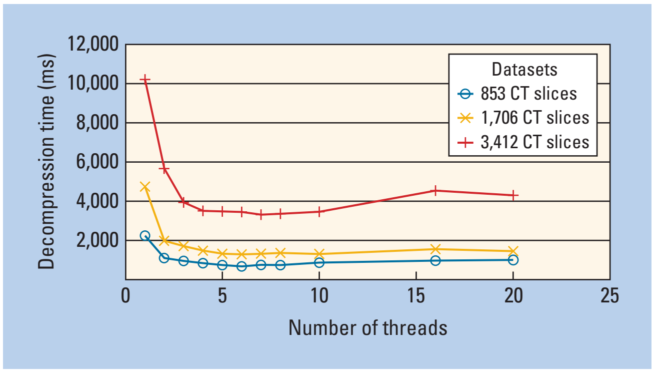


Figure 1. **Data decompression time as a function of the number of threads. The loading time scales with the number of threads, up to a value of four. The performance degrades when the number of threads approaches a value of 10.**

when the number of threads further increases toward a value of 10.

In this experiment, we decompressed the data after loading all images, which isn't optimal. In fact, we can decompress an image during loading, which leads to the pipeline pattern. In the pipeline implementation, we overlap the loading and decompression. Thus the decompression time is virtually hidden when decompression is faster than loading.

Because the image-loading code is in a low-level library, implementing the pipeline pattern is a bit tricky. We chose to use thread-pool threads, but we wanted to avoid creating new thread-pool threads because the process is slow in the .Net environment. To limit the number of active thread-pool threads, we chose to use a counting semaphore, which served as a gate to control when a new work item could be queued for worker threads. When the thread-pool thread finishes decompression, it just releases the semaphore. When all images are decompressed, an event is raised to notify the main thread.

To keep track of the number of processed images, we used a counter. The counter is a shared data element, and updating it is guarded by the use of an atomic operation (Interlocked class). In theory, we only needed two threads to manage the loading and decompression. However, because the order in which we access the image files might be inefficient from a file I/O viewpoint (for example, depending on how the files are physically stored on the disk, some files might have a long seek time), we chose to use asynchronous file I/O and issue more threads to access the compressed pixel data. The total loading and decompression times are 4,635 ms, 9,927 ms, and 18,177 ms for the three datasets. Similarly, the data loading times alone are 4,521 ms, 9,984 ms, and 18,276 ms. Clearly, the times are comparable, indicating that the loading time hid the decompression time.

We applied the same pipelining idea in another application we developed (the Syntegra application on a Sun UltraSparc III), where the application rescaled the images and offset them after loading. We used two threads, one loading the data and the other rescaling and offsetting the data. The two threads worked in tandem on the data in lock step, and the loading time similarly hid the processing time.

Runtime Throughput Improvement

In a respiratory gating study, clinicians acquire volumetric images corresponding to different respiratory phases. Typically, there are 10 phases and thus 10 volumetric datasets. The whole dataset is referred to as 4D data. For different purposes, the 4D data is projected along the time dimension to form 3D data. Typically, three kinds of projections are possible: maximum, minimum, and average intensity projections.

For lung tumors, the time-based maximum intensity projection will highlight the extent of tumor motion, since it's visually contrasted with air in the lungs. The time-based minimum intensity projection is useful for depicting where a portion of the tumor is consistently present throughout all phases of respiration. Both projections help define target tumor volumes, which are subsequently used for radiation therapy treatment. In addition, medical professionals can use the time-based average intensity projection for attenuation correction in PET reconstruction or for dose calculation in treatment planning.

The FusionViewer creates a projected volume at the user's request. The sequential implementation takes about 1.1 seconds for a 4D dataset with 10 phases, each containing 117 CT slices. To improve the application responsiveness, we parallelize the projection code. Focusing on the final 3D volumetric data, we can partition the volume into different chunks (data decomposition). The tasks to generate the projection are then identified. The implementation is again embarrassingly parallel. Similarly, we use fork/join and SPMD patterns in the implementation, and overall, we use the task parallelism pattern.

We measured the projection time as a function of the number of threads (see Figure 2). Generating the average projection takes slightly longer, because it involves more arithmetic. For all three projections, the projection time is nearly inversely proportional to the number of threads. When there are more than six or seven threads, we don't see much gain.

To understand why the improvement in projection time decreases so early, we wrote a test program using managed code to mimic the operations in the production code. The results (not shown here) show that the projection time is much slower than that obtained with the production code, but that the performance scales better with the number of threads as we approach eight threads. If we look at the differences between the production code and test code, we identify two memory-related differences.

First, the production code pins the data in memory and uses unsafe pointer operations to improve performance. Second, in the case of the production code, the pixel data is competing for memory with many other objects that reside in memory and are associated with the more complex runtime environment compared to the test code. We thus attribute the early decrease in projection time improvement to memory contention caused by the memory pinning and memory competition associated with the production implementation.

Parallel slow down because of memory contention and data sharing is a well-known issue in the high-performance computing community. The performance improves more slowly or even declines as the number of cores increases for data-intensive applications. Chip makers are considering this as they design next-generation multicore and many-core chips. As software designers, we must pay attention to data locality and memory usage on multicore platforms. We could have further investigated and optimized the code to improve the data locality. However, with eight threads, the projection time was close to 0.2 seconds, which is acceptable from an end-user viewpoint.

We used a similar data decomposition approach to visualize a single volume in our previous application (Syntegra). The maximum intensity projection (MIP) is a ray-tracing algorithm (a spatially based projection in comparison

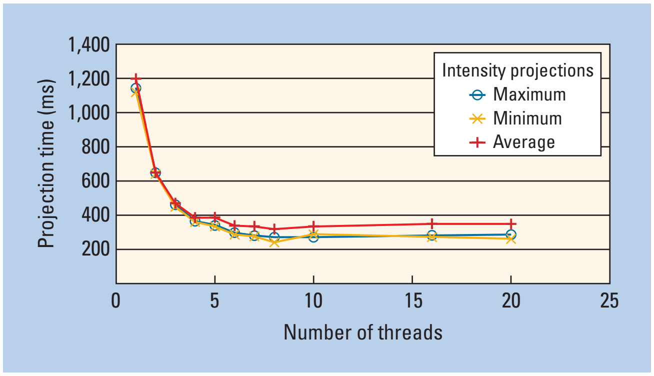


Figure 2. Projection time as a function of the number of threads. Generating the average projection takes slightly longer, because it involves more arithmetic. For all three projections, the projection time is nearly inversely proportional to the number of threads. When there are more than six or seven threads, we don't see much gain.

to the temporally based projection in the 4D case). It involves tracing a ray through a volume, whereby the maximum voxel along the ray determines the pixel value in the rendered 2D image. It's a popular visualization tool for PET data and is presented as a cine display of multiple MIPs at various viewing angles. It takes considerable time to generate a sequence of MIP images. To improve the speed, we divide the MIP image into chunks, and each thread works on one chunk. Using two threads, for example, reduced the MIP image generation time by nearly 50 percent.

Algorithm Reliability Improvement

Our discussion has focused on how to use parallel programming to decrease processing time. The fact that you can exploit multicore platforms to improve algorithm reliability is sometimes overlooked. Many algorithms involve random signals or processes, and the result often depends on the initial state. Given a computational result, it's hard to judge its correctness. This is particularly true for image registration problems, because these involve local optimization algorithms.

Image registration associates the image coordinates in two image spaces. It's the first step in many applications involving multiple images. In practice, automatic image registration is highly desirable. However, the registration results differ if you assume a different initial registration, use a different algorithm, use a different implementation of the same algorithm, or switch the roles of the images (reference and floating). On a multicore platform, you can run different scenarios in different threads (task parallelism), so you can identify the "best" result; or, if the discrepancy is high, you can notify the user.

We offer a simple example that demonstrates how to improve the reliability and robustness of an image registration application using multicore computing power. The registration algorithm is based on the iterative closest point (ICP) algorithm. Although we could parallelize the ICP algorithm itself, that's not our concern here. The points in the algorithm are geometric features extracted from images. Because the correspondence of the points is unknown, we used the ICP algorithm. Although the algorithm always converges monotonically to a local minimum with respect to the mean square error, there's no guarantee that the solution is, in fact, globally optimal. Various techniques have been introduced to tackle the problem, including multistart and trimmed ICP.

In our implementation, we swap the roles of two lists of points and add some perturbation to the initial registration (random variables uniformly distributed over [–5, 5] mm are added to the three translation offsets). Ideally, the resultant registration matrices will be inverted, and we'll be able to use this information to decide if the registration is acceptable. We multiply the two matrices, compare the product to the identity matrix, and decompose the resultant matrix to identify the translation and rotation components. The idea here is similar to the multistart method, except that when two lists are swapped, we may obtain a different number of pairs (we can pair multiple points in one list with a common point in another list).

Using the Microsoft .Net Framework, the idea is readily implemented using the asynchronous call pattern with the following pseudocode:

```
define delegate IcpAsynch
IcpAsynch icp = new IcpAsynch(CallIcp)
IAsyncResult ar =
   callIcp.BeginInvoke(…)
result = CallIcp(…)
ar.AsyncWaitHandle.WaitOne()
asynchResult = callIcp.EndInvoke(ar)
matrix =
   result.Matrix*asynchResult.Matrix
if (matrix close to identity)
   accept the result
else
   prompt the user
```

Here, a delegate is called asynchronously. This asynchronous call is executed on a thread-pool thread. On a multicore computer, it's executed by a different core. `CallIcp` is an ICP-based registration algorithm, and it's called in the current thread as well as in the thread-pool thread, but the lists of arguments are properly manipulated. When we applied this technique to the registration of a typical PET and PET image pair, the product of the two matrices yielded small translation offsets (1.64, 0.78, and –1.58 mm in three directions) and rotation angles (0.16, 0.00, and 0.15 degrees around three axes). This indicates that the two registration matrices are consistent, and it increases confidence in the registration's accuracy.

We can implement ICP with multistart on multicore systems as well. As more cores are available, we can run more copies of code in parallel with different perturbations to the initial registration matrix. By comparing the residual errors in each registration, we can select the registration that has the minimum residual error.

Exploiting the computing power of multicore platforms is a critical focus area for future application development. Following best practices and using expert experience can be an easy way to get started. By following these guidelines, you can systematically exploit computing power provided by the multicore platform to improve application startup time, runtime throughput, and algorithm reliability when developing medical imaging applications.

Acknowledgments

We thank Lynne Altomari for explaining the clinical use of 4D respiratory gated studies, Paul Farrell for comments on this article's technical content, Baruch Sabbah for providing the sequential data decompression code, and Marcel Weijenborg for the data loading-related discussions.

References

1. K. Beck and W. Cunningham, *Using Pattern Languages for Object-Oriented Programs*, tech. report CR-87-43, 1987; http://c2.com/doc/oopsla87.html.
2. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1994.
3. F. Buschmann et al., *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, John Wiley & Sons, 1996.

4. T.G. Mattson, B.A. Sanders, and B.L Massingill, *Patterns for Parallel Programming*, Addison-Wesley, 2005.
5. "Taking Parallelism Mainstream," Microsoft, 2008; http://msdn.microsoft.com/en-us/concurrency/ee847335.aspx.
6. Y.M. Zhu et al., "Improving Startup Performance for a Medical Image Viewing Application," *IT Professional*, Mar./Apr. 2008, pp. 38–45.
7. M. Sen et al., "Model-Based Mapping of Reconfigurable Image Registration on FPGA Platforms," *J. Real-Time Image Processing*, vol. 3, no. 3, 2008, pp. 149–162.
8. J.J. Koo, A.C. Evans, and W.J. Gross, "3-D Brain MRI Tissue Classification on FPGAs," *IEEE Trans. Image Processing*, vol. 18, no. 12, 2009, pp. 2735–2746.
9. S. Park and H. Jeong, "High-Speed Parallel Very Large Scale Integration Architecture for Global Stereo Matching," *J. Electronic Imaging*, vol. 17, no. 1, 2008, pp. 010501-1–010501-3.
10. G. Pratx et al., "Fast, Accurate and Shift-Varying Line Projections for Iterative Reconstruction Using the GPU," *IEEE Trans. Medical Imaging*, vol. 28, no. 3, 2009, pp. 435–445.

***Yang-Ming Zhu** is a lead software architect in the Nuclear Medicine Division at Philips Healthcare. His research interests focus on software engineering and image processing. Zhu received his PhD in bioelectronics from Southeast University, China. He's a member of the IEEE Computer Society and IEEE Signal Processing Society. Contact him at yzhu@computer.org; www.yzhu.org.*

***Steven M. Cochoff** is a software director in the Nuclear Medicine Division at Philips Healthcare. His research interests include medical image processing and clinical analysis. Cochoff received his MS degree in biomedical engineering from Case Western Reserve University. Contact him at steve.cochoff@philips.com.*

cn **Selected CS articles and columns are available for free at http://ComputingNow.computer.org.**

ADVERTISER INFORMATION
MARCH/APRIL 2010 • *IT PROFESSIONAL*

Advertiser	Page
Classified Advertising	13
SMS	13

Advertising Personnel
Marion Delaney
IEEE Media, Advertising Dir.
Phone: +1 415 863 4717
Email: md.ieeemedia@ieee.org

Marian Anderson
Sr. Advertising Coordinator
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: manderson@computer.org

Sandy Brown
Sr. Business Development Mgr.
Phone: +1 714 821 8380
Fax: +1 714 821 4010
Email: sb.ieeemedia@ieee.org

Advertising Sales Representatives

Recruitment:

Mid Atlantic
Lisa Rinaldo
Phone: +1 732 772 0160
Fax: +1 732 772 0164
Email: lr.ieeemedia@ieee.org

New England
John Restchack
Phone: +1 212 419 7578
Fax: +1 212 419 7589
Email: j.restchack@ieee.org

Southeast
Thomas M. Flynn
Phone: +1 770 645 2944
Fax: +1 770 993 4423
Email: flynntom@mindspring.com

Midwest/Southwest
Darcy Giovingo
Phone: +1 847 498 4520
Fax: +1 847 498 5911
Email: dg.ieeemedia@ieee.org

Northwest/Southern CA
Tim Matteson
Phone: +1 310 836 4064
Fax: +1 310 836 4067
Email: tm.ieeemedia@ieee.org

Japan
Tim Matteson
Phone: +1 310 836 4064
Fax: +1 310 836 4067
Email: tm.ieeemedia@ieee.org

Europe
Heleen Vodegel
Phone: +44 1875 825700
Fax: +44 1875 825701
Email: impress@impressmedia.com

Product:

US East
Dawn Becker
Phone: +1 732 772 0160
Fax: +1 732 772 0164
Email: db.ieeemedia@ieee.org

US Central
Darcy Giovingo
Phone: +1 847 498 4520
Fax: +1 847 498 5911
Email: dg.ieeemedia@ieee.org

US West
Lynne Stickrod
Phone: +1 415 931 9782
Fax: +1 415 931 9782
Email: ls.ieeemedia@ieee.org

Europe
Sven Anacker
Phone: +49 202 27169 11
Fax: +49 202 27169 20
Email: sanacker@intermediapartners.de