SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2009; **39**:641–660
Published online 31 December 2008 in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.915

Scopira: an open source C++ framework for biomedical data analysis applications

Aleksander B. Demko[1,2] and Nick J. Pizzi[1,2,*,†]

[1]*Institute for Biodiagnostics, National Research Council of Canada, Winnipeg, Canada*
[2]*Department of Computer Science, University of Manitoba, Winnipeg, Canada*

SUMMARY

In many biomedical research laboratories, data analysis and visualization algorithms are typical prototypes using an interpreted programming language. If performance becomes an issue, they are ported to C and integrated with interpreted systems, not fully utilizing object-oriented software development. This paper presents an overview of Scopira, an open source C++ framework suitable for biomedical data analysis and visualization. Scopira provides high-performance end-to-end application development features, in the form of an extensible C++ library. This library provides general programming utilities, numerical matrices and algorithms, parallelization facilities, and graphical user interface elements. Copyright © 2008 John Wiley & Sons, Ltd.

Received 17 July 2008; Revised 12 November 2008; Accepted 22 November 2008

KEY WORDS: data analysis; visualization; software engineering; application development framework; distributed computing

1. INTRODUCTION

The initial driving force for this project was to develop a comprehensive, object-oriented programming architecture using C++ for the development of applications relating to exploratory data analysis of magnetic resonance images (MRI), especially functional MRI [1]. Subsequently, we expanded the architecture to deal with confirmatory and exploratory biomedical data analysis, visualization, and interpretation, in general. This approach strikes a balance between slow interpreted languages such as IDL [2,3] and MATLAB® [4,5] and fast compiled languages such as C and

*Correspondence to: Nick J. Pizzi, Institute for Biodiagnostics, National Research Council of Canada, 435 Ellice Avenue, Winnipeg MB, Canada R3B 1Y6.
†E-mail: pizzi@nrc-cnrc.gc.ca, pizzi@cs.umanitoba.ca

FORTRAN. Although well suited for algorithm prototyping and *ad hoc* data visualization, interpreted languages are simply not suitable for application development. Conversely, C and FORTRAN, although efficient, lack basic and expected language features such as object orientation or basic memory management required for building large-scale applications. C++ was chosen to straddle the two extremes, and even though it has been somewhat overshadowed by newer languages such as Java or C#, it is still the only language with features such as generics and object orientation that compile into efficient machine code.

Our motivation behind the design of Scopira was to satisfy the needs of three categories of users within the biomedical research community: developers, scientists/technologists, and data analysts. With the design, implementation, and validation of new biomedical data analysis software, developers typically need to incorporate legacy systems often written in interpreted languages. When this is coupled with the facts that, in a research environment, user requirements often change (sometimes radically) and that biomedical data are becoming ever more complex and voluminous, a development framework must be versatile, extensible, and exploit distributed, generic, and object-oriented programming paradigms. For the biomedical scientist or technologist, data analysis tools must be intuitive with responsive interfaces that operate both effectively and efficiently. Finally, the biomedical data analyst has requirements straddling those of the developer and the scientist. With an intermediate level of programming competence, they require a relatively intuitive development environment that can hide some of the low-level programming details, while at the same time allowing them to easily set up and conduct numerical experiments that involve parameter tuning and high-level looping/decision constructs. As a result of this motivation, the emphasis with Scopira [6] has been on high-performance, open-source development and the ability to easily integrate other C/C++ libraries used in the biomedical data analysis field by providing a common OOP API for applications. This library provides a large breadth of services that fall into the following four component categories:

Scopira Tools provide extensive programming utilities and idioms useful for all application types. This category contains the reference counted memory management system, flexible/redirectable flow input/output system, which supports files, file memory mapping, network communication, and check sum calculation, as well as object serialization and persistence, reproducible and tunable random number generation, universally unique identifies (UUIDs) and XML parsing and processing.

The *Numerical Functions* all build on the core n-dimensional *narray* concept (see Section 4). C++ generic programming is used to build custom, high-performance arrays of any data type and dimension. General mathematical functions build on the *narray*. A large suite of biomedical data analysis and pattern recognition functions are also available.

Multiple APIs for *Parallel Processing* are provided, allowing algorithms to scale with available processor and cluster resources. Scopira provides easy integration with native operating system threads, MPI [7,8] and PVM [9,10] libraries. A Scopira-based object-oriented framework, *Scopira Agents Library*, is included, which may be embedded into desktop applications allowing them to use computational clusters automatically, when detected. Unlike other parallel programming interfaces such as MPI and PVM, Scopira's facilities provide an object-centric strategy with support for common parallel programming patterns and approaches.

Finally, a *Graphical User Interface* (*GUI*) *Library* based on GTK+ is provided. This library provides a collection of useful widgets including a scalable numeric matrix editor, plotters, image and viewers as well as a plug-in platform and a 3D canvas based on OpenGL® [11,12].

DOI: 10.1002/spe

The following section describes several Scopira-based data analysis applications used by the biomedical research community. In the subsequent four sections we describe, in turn, each of the Scopira component categories, which are followed by some concluding remarks.

2. APPLICATIONS

We implemented several biomedical data analysis applications using Scopira. Some are in-house, proprietary, and highly specialized systems, while others are open-source applications that are available to the biomedical research community at large. These applications run the gamut from confirmatory to exploratory data analysis, image processing, pattern recognition, classification, and visualization. We briefly present three applications developed using Scopira.

One Scopira-based application is EvIdent® [13], an exploratory data analysis system for rapidly investigating novel events in a set of two- or three-dimensional images (e.g. MRI, infrared, spectroscopic maps, etc.) as they evolve over time or frequency (or any other analysis dimension). For instance, in a series of functional magnetic resonance neuroimages, novelty may manifest itself as neural activations over a time course (see Figure 1). The core of the system is an enhanced variant of the fuzzy c-means clustering algorithm [14]. Fuzzy clustering obviates the need for models of the underlying requisite biological function, models that are often statistically suspect. EvIdent® offers several innovations: (i) biomedical researchers may probe for unanticipated but domain-significant structure in the data, (ii) flexible generation of unbiased, testable models, (iii) rapid analysis of data

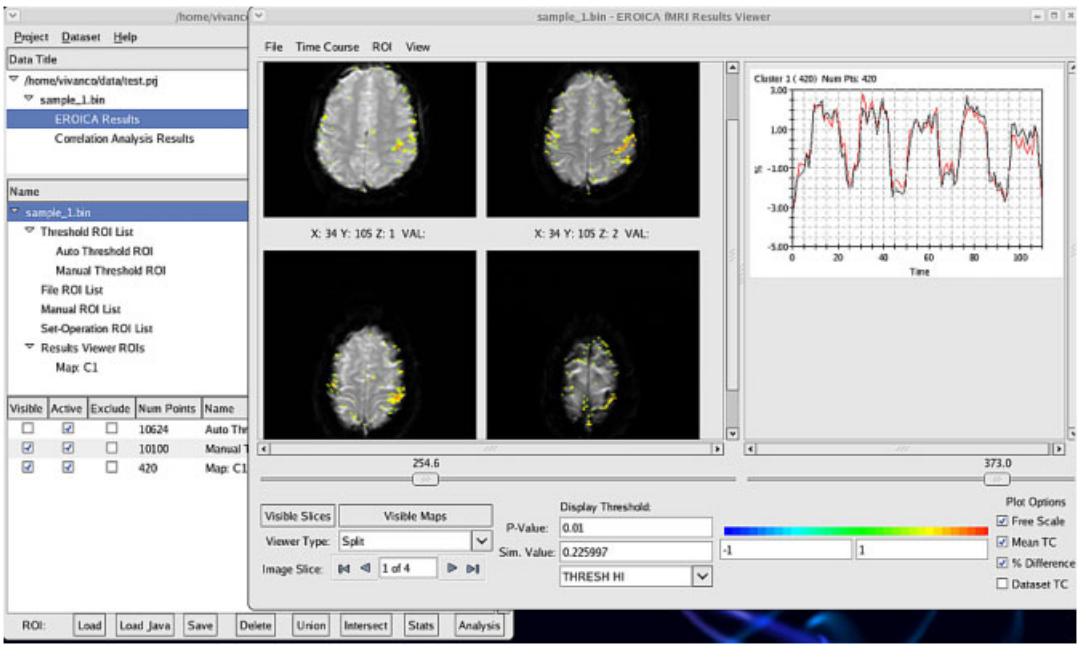


Figure 1. Functional MRI activation map viewer in EvIdent®.

DOI: 10.1002/spe

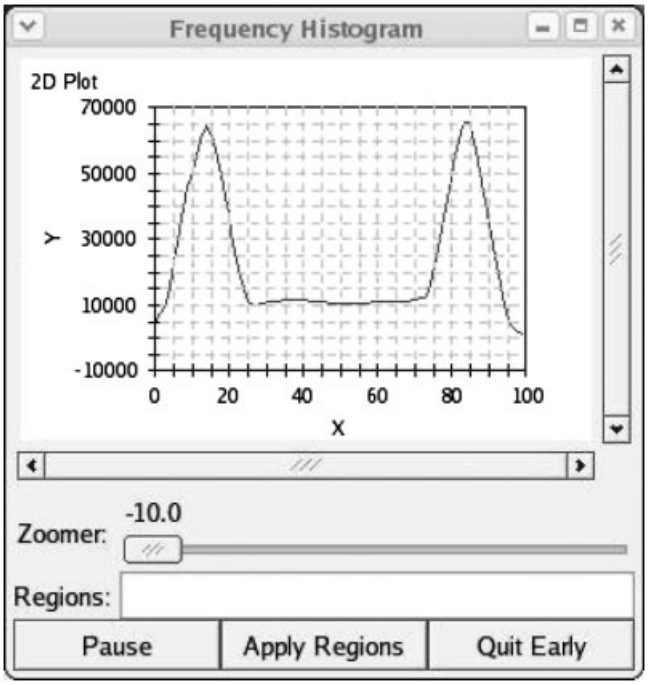


Figure 2. Feature frequency histogram used by SFS, a Scopira-based parallelized biomedical data classification system.

in complex cognitive experiments, and (iv) excellent precursor and complement to any model-based inferential method.

Another application we implemented with the Scopira framework is stochastic feature selection (SFS), an iterative and highly parallelized feature dimensionality reduction technique [15] for the classification of complex voluminous biomedical data. SFS randomly assigns the original data set samples (e.g. magnetic resonance spectra) into design and test sets. Once the design phase is complete (i.e. classification coefficients have been determined), the test set is used to externally validate the classification performance. The stochastic nature of SFS is controlled by a feature frequency histogram (see Figure 2) whereby the performance of each classification iteration is assessed using a fitness function. An *ad hoc* cumulative distribution function, constructed from this histogram, is iteratively used to randomly sample new features (rather than each feature having an equal likelihood of being selected for a new classification iteration, only those features used in previous 'successful' iterations are selected). Via Scopira's parallelization facilities (see Section 5), SFS bundles classification iterations to minimize inter-process communication and maximize CPU loads. Furthermore, while SFS exploits parallelism, it remains (optionally) strictly deterministic, that is, results are perfectly reproducible regardless of computational load (an extremely useful benefit for biomedical research). The third Scopira-based application involves the analysis, visualization (via Scopira and VTK), and interpretation of biomedical images required using optical coherence tomography (OCT) [16], an optical imaging modality that provides micrometer scale resolution

DOI: 10.1002/spe

morphological images. OCT is similar to ultrasound in operation except that low coherent near infrared light is used instead of sound. The light is focused onto a sample and back reflections from within the sample are recorded to create a morphological image of the interior structure of the sample. The back reflections occur from changes in optical density at tissue boundaries and cellular structures. The three-dimensional morphological images have an axial resolution of 10 μm and a transverse resolution of 25 μm, which are superior to standard ultrasound images. The coherence requirement of OCT in highly scattering biological tissue limits penetration depths to 2 mm. However, the method is fully implemented in fiber optics, allowing sub-millimeter probes to collect images via catheters and endoscopes [17].

3. PROGRAMMING UTILITIES

Scopira consists of modular subsystems that can be used as needed by developers. The *Scopira Tools* subsystem provides generic facilities useful in many programming domains, not just numerical and scientific computing.

The standard C++ library is comparatively slow in adopting and standardizing new functionality. This is done to maximize quality, but unfortunately forces developers to go to third-party libraries for needed functions. This can quickly lead to many external library dependencies, each with their own disjoint designs and interface styles. This leads to more complex and fragile user code, as developers try to bridge various libraries into their own application.

Many of the core functions (especially in the *Scopira Tools* subsystem) can be found in other libraries (such as Boost). However, rather than force the user to integrate many libraries for various features, it was deemed much more efficient to provide these basic features all within Scopira itself. Large libraries (such as GUI libraries or MPI) are not re-implemented, but simply augmented. Although this can be considered poor code reuse, the benefits of providing these functions within one library, all via a common and consistent interface was too great to ignore. This technique of developing a common interface to basic functions over importing libraries is common to many large libraries, such as MFC [18,19], Qt [20,21], wxWidgets [22,23], and VTK [24,25].

Libraries such as Boost [26,27] try to alleviate this by providing various common facilities in a high-quality library, with faster implementation cycles than the standard C++ library. For Scopira, Boost did provide some features readily (such a threads), others were in development (such as smart pointers), took a long time to develop (such as serialization), stalled in their development (such as UUIDs) or decided not to implement (such as XML processing). However, these small features were not deemed significant enough to introduce a new library dependency and programming interfaces. Perhaps as various Boost libraries get adopted in future C++ standards, they will become the common building blocks of many more C++ libraries, frameworks, and applications.

3.1. Memory management

An *intrusive* reference counting scheme provides the basis for memory management. The scheme is considered intrusive as it records an object's reference count within the object itself, typically by having the object descend from a common base class. Unlike many referencing counting systems (such as those in VTK [7] and GTK+ [28,29]), Scopira's system uses a decisively symmetric concept. References are only added through the *add_ref* and *sub_ref* calls—specifically, the object

DOI: 10.1002/spe

itself is born with a reference count of zero. This greatly simplifies the implementation of smart pointers and easily allows stack allocated use (by passing the reference count), unlike VTK and GTK+ where objects are born with a reference count and a modified reference count, respectively.

Scopira implements a template class *count_ptr* that emulates standard pointer semantics while providing implicit reference counting on any target object. Alternatively, the *intrusive_ptr* from the Boost library may also be used, as Scopira's reference counting scheme is compatible with its requirements. With either smart pointer, reference management becomes considerably easier and safe, a vast improvement over C's manual memory management.

Boost's *smart_ptr* template class was also considered, but initially ruled out for a variety of reasons. At the time of the reference counting scheme's design (2001), *smart_ptr*'s development was still in a state of flux, and it was deemed onerous to require developers to install Boost for this single class. The class provides non-intrusive reference counting, meaning that it keeps the reference count outside of the target object. Although this has huge flexibility benefits (as it can be applied to any type), it requires the programmer to be diligent (and more verbose) and always use the class when passing the class between methods and functions. We found that allowing less experienced C++ programmers to casually convert their reference back to standard pointers occasionally allows for much more concise and familiar code while maintaining proper reference counts, a feat that is impossible (due to their design) with non-intrusive smart pointers. However, Boost's *smart_ptr* implementation is on track to be included in the next C++ standard, which will greatly increase its exposure and familiarize a much larger portion of C++ programmers to flexible and powerful reference counting memory management, providing a common standard that all C++ libraries can use.

3.2. Input/Output

Scopira provides a flexible, polymorphic, and layered input/output system (see Figure 3). Flow objects may be linked dynamically to form I/O streams. Scopira includes *end flow* objects, which terminate or initiate a data flow for standard files, network sockets, and memory buffers. *Transform flow* objects perform data translation from one form to another (e.g. binary-to-hex), buffer consolidation and ASCII encoding. Future transformers include CRC calculators, compressors, and cryptographic ciphers. *Serialization flow* objects provide an interface for objects to encode their data into a persistent stream. Through this interface, large complex objects can quickly and easily

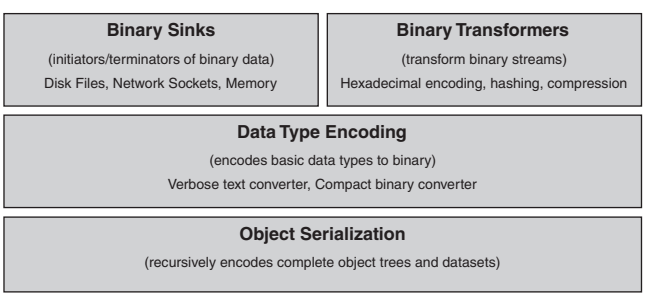


Figure 3. Scopira input/output stack.

DOI: 10.1002/spe

encode themselves to disk or over a network. Upon reconstruction, the serialization system re-instantiates objects from type information stored in the stream. Shared objects—objects that have multiple references—are serialized just once and properly linked to multiple references.

3.3. Configuration and plug-ins

A platform-independent configuration system is supplied via a central parsing class. This class is able to accept input from a variety of sources (configuration files, command line parameters, etc.) and present them to the programmer in one consistent interface. The programmer may also store settings and other options via this interface, as well as build GUIs to aid in their manipulation by the end user.

Using a combination of the serialization-type registration system and C++'s native RTTL functions, Scopira is able to dynamically (at runtime) allow for the registration and inspection of object types and their class hierarchy relationships. From this, an application plug-in system can be trivially built by allowing external modules (dynamic link libraries) to register their own types as being compatible with an application, providing a platform for third-party application extensions.

3.4. Other utilities

Finally, the tools subsystem provides a variety of other services and interfaces. Native operating system threads (via the POSIX threads interface) are presented as C++ objects, with mutex locking and shared areas accessed via classes that follow the *Resource Acquisition Is Initialization* principle. Generic arrays provide a lightweight (yet still STL like) array class that is simpler than STL's vector class and not specific to numeric computing as is Scopira's *narray*. Random number generation (inspired by Boost's random library) is also included. UUIDs and uniform resource locators (URLs) are provided. XML processing (provided by the libxml2 library) is an optional feature, allowing one to build open and easy to use data file formats.

4. N-DIMENSIONAL DATA ARRAYS

4.1. Existing arrays

The C and C++ languages provide the most basic support for one-dimensional arrays, which are general and are closely related to C's pointers. However, although usable for numerical computing, they do not attempt to provide the additional functionality that scientists demand, such as easy memory management, mathematical operations, or fundamental features such as storing their own dimensions. Multiple-dimensional arrays are even less used in C/C++, as they require compile-time dimension specifications, drastically limiting their flexibility.

The C++ language, rather than design a new numeric array type, provides all the necessary language features for developing such an array in a library. Generic programming (via C++ templates, that allow code to be used for any data types at compile time), operator overloading (e.g. being able to redefine the plus '+' or assignment '=' operators), and inlining (for performance) provide all the tools necessary to build a high-performance, usable array class.

DOI: 10.1002/spe

Users have created their own libraries to fill the void left by the lack of standardized multi-dimension array classes in C++. These libraries vary in performance, API style, and focus. Some of the better established packages will be discussed here.

The highly regarded Boost C++ libraries [7] contain not one, but two numerical array libraries, both introduced in version 1.29 of the library collection: *Boost.MultiArray* and *uBLAS*.

Boost.MultiArray [in boost] provides a basic, but complete n-dimensional array class with support for views and slices. The library, like many of those in the Boost collection, utilizes advanced C++ features and idioms to achieve their goals of performance and completeness, sometimes sacrificing ease of use for newer C++ programmers. This library, at its core has the most in common with Scopira narray classes (see the following section), differing mainly in their notions of element access and use of temporaries.

uBLAS is a C++ library [in boost] that provides BLAS functionality for a variety of different matrix types. Building on Basic Linear Algebra Sub-programs (BLAS) FORTRAN library, *uBLAS* is designed with performance in mind (especially with the goal of being no worse that the FORTRAN predecessors) and focuses on linear algebra operations and matrix data types. The library supports a variety of matrix types (including dense, packed, and sparse matrices) and does not generalize at all to larger dimensions.

The Blitz++ library [30] is an older library that provides an n-dimensional array class, complete with slicing. The API focuses on the array classes itself, and does not offer a collection of algorithms, or interpolation aids with visualization systems or other libraries. The development of Blitz++ has slowed after a decade, and has switched to a maintenance mode without reaching a seminal 1.0 version.

Although there are numerous implementations of n-dimensional array classes, algorithm developers and users often need not be too concerned with over committing or being locked into one particular implementation. Owing to the large influence of the C++ STL on the various library developers, there are only small set of element access styles that are used. Many also offer raw C-array-like access to ease interfacing with other libraries. Using simple adapter classes or systematic source code factoring, developers may quickly update their code to work with any new libraries.

4.2. narray class

Rather than force the developer to add another dependant library for an array class, Scopira provides n-dimensional arrays through its *narray* class. This class takes a straightforward approach, implementing n-dimensional arrays as any C programmer would have, but providing a type safe, templated interface to reduce programming errors and code complexity. The internals are easy to understand, and the class works well with standard C++ library iterators as well as C arrays, minimizing lock-in and maximizing code integration opportunities.

Using basic C++ template programming, we can see the core implementation ideas in the following code snippet:

```
template <class T, int DIM> class narray {

 T* dm_ary;                 // actual array elements

 nindex<DIM> dm_size;       // the size of each of the dimensions
```


DOI: 10.1002/spe

```
  T get(nindex<DIM> c) const {

        assert(c<dm_size);

        return dm_ary[dm_size.offset(c)];

  }
 }
```

From this code snippet we can see that an *narray* is a template class with two compile-time parameters: *T*, the element data type (*int*, *float*, etc.) and *DIM*, the number of dimensions (1, 2, 3, etc.). The actual elements are stored in a dynamically allocated C array, *dm_ary*. The dimension lengths are stored in an *nindex* type, a generic class that is used to store array offsets.

A generalized accessor is provided, which uses the *nindex*-offset method to convert the dimension-specific index and size of the array into an offset into the C array. This generalization works for any dimension size.

Another feature shown here is the use of C's *assert* macro to check the validity of the supplied index. This boundary check verifies that index is indeed valid otherwise failing and terminating the program while alerting the user. This check greatly helps the programmer during the development and testing stages of the application, and during a high-performance/optimized build of the application, these macros are transparently removed, obviating any performance penalties from the final, deployed code.

More user-friendly accessors (such as those taking an *x* value or an *x* and *y* value directly) are also provided. Finally, C++'s operator overloading facilities are used to override the bracket '[]' and parenthesis '()' operators to give the arrays a more succinct and natural feel, over explicit *get* and *set* method calls.

4.3. nslice class

The *nslice* template class is a virtual n-dimensional array that is simply a *reference* to an *narray*. The class only contains dimension specification information and is easily copyable and passable as function parameters. Element access translates directly to element accesses in the host *narray*. An *nslice* must always be of the same numerical type as its host *narray*, but can have any dimensionality less than or equal to the host. This flexibility is very powerful; one could have an one-dimensional vector slice from a matrix, cube or five-dimensional array, for example. Matrix slices from volumes are also quite common (see Figure 4). These sub slices can also span any of the dimensions/axes, something not possible with simple pointer arrays (for example, matrix slices from a cube array need not follow the natural memory layout order of the array structure).

4.4. Memory mapping

The *narray* class provides hooks for alternate memory allocation systems. One such system is the *DirectIO* mapping system. Using the memory mapping facilities of the operating system (typically via the *mmap* function on POSIX systems), a disk file may be *mapped* into memory. When this

DOI: 10.1002/spe

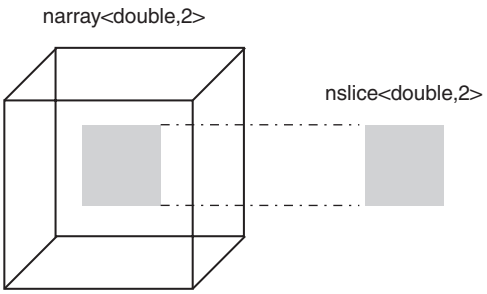


Figure 4. An nslice reference into an narray data.

memory space is accessed, the pages of the files are loaded into memory transparently. Writes to the memory region will result in writes to the file.

This allows files to be loaded in portions and on demand. The operating system will take care of loading and unloading the portions as needed. Files larger than the system's memory size can also be loaded—the operating system will keep only the working set portion of the array in memory. The programmer must be aware of this, however, and take care to keep the working set within the memory size of the machine. If the working set exceeds the available memory size, performance will suffer greatly as the operating system pages portions to and from disk (this excessive juggling of disk-memory mapping is sometimes called 'page thrashing').

5. PARALLEL PROCESSING

With the increasing number of processors in both the users' desktops and in cluster server rooms, computationally intensive applications and algorithms should be designed in a parallel manner if they are to be relevant in a future that depends on multiple-core and cluster computing as a means of scaling processing performance. To take advantage of the various processors within a single system or shared address space (SAS), developers need only utilize the operating system's thread API or shared memory services. However, for applications that would also like to utilize the cluster resources to achieve greater scalability, explicit message passing (MP) is used. Although applying a SAS model to cluster computing is feasible, to achieve the best performance and scalability results, an MP model is preferred [31].

Scopira includes support for two well-established MP interfaces, MPI and PVM, as well as a custom, embedded, object-oriented MP interface designed for ease of use and deployment.

5.1. MPI and PVM

This subsystem provides a set of *narray* aware template functions and input/output classes that allow developers to easily interface with the MPI and PVM programmer's APIs. Using C++ trait classes for type information and the size data already stored in *narray*, these functions drastically reduce the amount of parameters needed from the programmer thereby reducing common mistakes when using these libraries.

DOI: 10.1002/spe

5.2. Scopira agents library

The Scopira Agents Library (SAL) is a parallel execution framework extension with several notable goals particularly useful to Scopira-based applications. The API, which is completely object-oriented, includes functionality for: (i) using the flow system for messaging, (ii) task movement, (iii) check-pointing (supporting both primitive and basic data types as well as user-defined objects), and (iv) the registration system for task instantiation.

SAL introduces high-performance computing to a wider audience of users by permitting developers to build standard cluster capabilities into desktop applications, allowing those applications to pool their own as well as cluster resources. This is in contrast to the goals of MPI (providing a dedicated and fast communications API standard for clusters) and PVM (providing a virtual machine architecture among a variety of powerful platforms).

By design, SAL borrowed a variety of concepts from both MPI and PVM. SAL, like PVM, attempts to a build a unified and scalable 'task' management system with an emphasis on dynamic resource management and interoperability. Users develop intercommunicating task objects. Tasks can be thought of as single processes or processing instances, except they are implemented as language objects and not operating system processes. An agent manages one or more tasks, and teams of agents communicate with each other to form computational networks (see Figure 5). The tasks themselves are coupled with a powerful MP API inspired by MPI. Unlike PVM, SAL also focuses on ease of use: emphasizing automatic configuration detection and de-emphasizing the need for infrastructure processes.

When no cluster or network computation resources are available, SAL uses operating system threads to enable multi-programming within a single OS process and thereby embedding a complete MP implementation within the application (greatly reducing deployment complexity). Applications always have an implementation of SAL available, regardless of the availability or access to cluster resources. Developers may always use the MP interface, and their application will work with no configuration changes from both single machine desktop installations to complete parallel compute cluster deployments.

The mechanics and implementation of the agents and their load balancing system are built into the agents extension library, and thereby, Scopira applications. Users do not need to install additional software, nor do they need to explicitly configure or setup a parallel environment. This is paramount

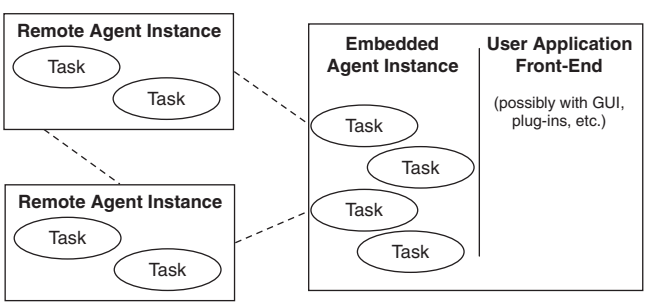


Figure 5. SAL tasks and agents topology.

DOI: 10.1002/spe

in making cluster and distributed computing accessible to the non-technical user, as it makes it a transparent feature in their graphical applications.

5.2.1. Messaging

SAL provides an object-oriented, packet based and routable (like PVM, but unlike MPI) API for MP. This API provides everything needed to build multi-threaded, cluster-aware algorithms embeddable in their applications.

Tasks are the core objects that developers build for the SAL system. A task represents a single job or instance in the agent system, which is analogous to a process in an operating system. However, they are almost never separate processes, but rather grouped into one or more agent processes that are embedded into the host application. This is unlike most existing parallel APIs, that allocate one OS process per task concept, that, although conceptually simpler for the programmer, incurs more communication and start-up overhead, as well as making task management more complex and OS dependent. The tasks themselves are language-level objects but are usually assigned their own operating system threads to achieve pre-emptive concurrency.

A context object is a task's gateway into the SAL MP system. There may be many tasks within one process; hence, each will have differing context interfaces—something not feasible with an API with a single, one-task-per-process model (as used in PVM or MPI). This class provides several facilities, including: task creation and monitoring; sending, checking and receiving messages; service registration; and group management. It is the core interface a developer must use to build parallel applications with SAL.

Developers often launch a group of instances of the same task time, and then systematically partition the problem space for parallel processing. To support this popular paradigm of development, SAL's identification system supports the concept of *groups*. A group is simply a collection of N task instances, where each instance has a *groupid* $\in [0, N-1]$. The group concept is analogous to MPI's communicators (albeit without support for complex topology) and PVM's named groups. This sequential numbering of task instances allows the developer to easily map problem work units to tasks. Similar to how PVM's group facility supplements the task identifier concept, SAL groups build on the UUID system, as each task still retains—and may use—their underlying UUID for identification.

The messaging system within SAL is built on both the generic Scopira I/O layer as well as the UUID identification system. SAL employs a packet-based (similar to PVM) message system, where the system only sends and routes complete messages, and not the individual data primitives (as MPI can and often does) and objects within them. Only after the sending task completes and commits a message is it processed by the routing and delivery systems. The SAL agent uses OS threads to transport the data, freeing the user's thread to continue to work. Overlapping IO to increase processor utilization is also used by some implementations of MPI, such as USFMPI [32].

Sending (committing) the data during the *send_msg* object's destruction (that is, via its destructor) was the result of an intentional design decision. In C++, stack objects are destroyed as they exit scope. The user should therefore place a *send_msg* object in its own set of scope-braces, which would constitute a sort of 'send block'. All data transmissions for the message would be done within that block, and the programmer can then be assured that the message will be sent at the end of the scope block without having to remember to do a manual send commit operation. Similarly,

DOI: 10.1002/spe

the receiver uses a *recv_msg* object to receive, decode and parse a message packet, all within a braced 'receive block.'

The following code listing provides an example of a task object that, via its context interface (the interface to the message network), is sending a variety data objects using the object-oriented messaging API:

```
// declare my task object and its run method

class mytask : public agent_task_i

  { public: virtual void int run(task_context &ctx); };

int mytask::run(task_context &ctx) {

   // send some data to the master task

  narray<double,2> a_matrix;

  {      // this scope (or send) block encapsulates the sent message

   send_msg msg(ctx, 0);   // prepare the message to task #0


   msg.write_int(10); // send one integer

   msg.save(a_matrix); // send a matrix - type safe

   msg.save(user_object);    // send a user object - via serialization


   // at this point, msg's destructor will be called (automatically)

  // triggering the sending of the message

 }
}
```

5.2.2. *Scheduling engines*

SAL consists of four components. The central component is the messaging API. Generic services and other messaging interfaces utilize this API. Under the hood, the agent infrastructure code implements the API, which in turn selects and loads an engine (the task and messaging system) at runtime. The engine component implements the bulk of the API code and is responsible for task management, message transport, and processor management. SAL currently has two types

DOI: 10.1002/spe

of engines, a 'local' engine that uses operating system threads on a single host machine and a 'network' implementation that is able to utilize a network of workstations.

The 'local' engine is a basic multi-threaded implementation of the SAL API. It uses the operating system's threads to implement multiprocessing within the host application process. That is, the engine lacks the networking abilities to manage separate nodes and intercommunication but is able to use all the processors on the host machine by using operating system threads within the host process.

As this engine is contained within a single process, it is the fastest and easiest to use for application development and debugging. The programmer may fully design and test their parallel algorithm and its messaging logic before moving to a multi-node deployment. Furthermore, as multi-processor and multi-core desktop systems become more commonplace, this basic engine may also be useful for low to mid-range deployments and may be suitable for users who may not need full cluster resources in most situations. The local engine is always available and requires no configuration from the user. Developers need not write a dedicated non-MP versions of their algorithms simply to satisfy users that may not go to the trouble of deploying a cluster.

The local engine does no load balancing. As the engine provides as many worker threads as active tasks, it relies on the operating system's ability to manage threads within the processors. This works quite well, when the number of tasks instantiated into the system is a function of the number of physical processors, as encouraged by the API. As there is only one primary user/initiator in a local engine (that is, the host application's user), the number of task groups in the system is predictable (often, one).

In summary, we find the implementation of the local engine to be relativity straightforward. Without the complexities of network communication, the engine implementation itself is simply a collection of shared associative arrays, with various levels of mutexes and conditions all shared by a group of worker operating system threads. This makes for a perfect reference implementation of the API, useful for both debugging and for production deployments where the user's desktop machine is of sufficient processing power.

5.2.3. Network engine

The network engine implements the SAL API over a collection of machines connected by an IP-based network; typically Ethernet. The cluster can be a dedicated computer cluster, a collection of user workstations, or a combination. The engine itself provides inter-node routing and management, leaving the local scheduling decisions within each node up to a local-engine derived manager.

An SAL network stack has two layers (see Figure 6). The lower transport layer contains the agents themselves (objects that manage all the tasks and administration on a single process) and their TCP/IP-based links. The agents virtualize and present the messaging layer, where tasks can send messages to each other using their UUIDs, ignorant of the IP layer or the connection topology of the agents themselves. For simplicity and efficiency, an SAL network (like PVM) has a master agent residing on one process. This master agent is responsible for the allocation, tracking, and migration of all the tasks in the system. It is assumed that within a single site deployment of an SAL network, at least one stable server (i.e. non-user desktop) machine could be found to assume this role. A centralized master allows for simpler and faster task administration.

The network engine uses a combination of URL-like direct addressing and UDP/IP broadcast-based auto-discovery in building the agent network. The simplest and most popular sequence is to

DOI: 10.1002/spe

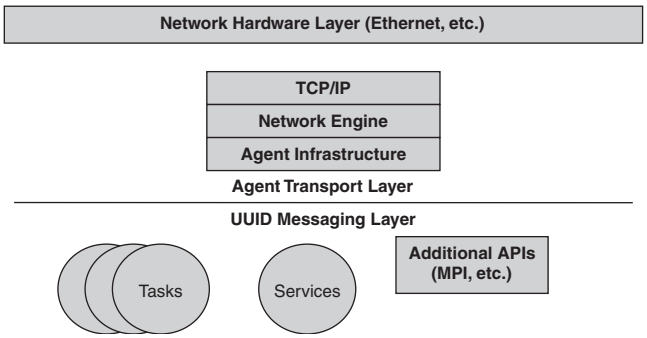


Figure 6. The SAL network stack.

start an application in *auto discovery mode*. When a network engine starts, it searches the local network for any other agent peers and, if found, joins their network. If no peers are found, then it starts a network consisting of itself as the only member and assumes the master agent role. Users may also key in the master's URL directly, connecting them explicitly to a particular network.

In addition to its critical routing functions, the master agent is also responsible for all the task tracking and management within the network. By centralizing this information, load and resource allocation decisions can be made instantly and decisively.

For each agent peer, the master tracks its load, routing policy (direct or indirect routing) and task running policy. Specifically, each agent is able to specify what types of jobs it is willing to accept: all jobs, no jobs (useful for desktop nodes or front end nodes) and only self-initiated jobs (for agents that are present only for their own jobs).

All task instantiation requests are handled by the master agent. When a task within an agent requests the creation of more tasks, the request is routed by the hosting agent to the master agent. Based on the current loads and hosting policies of the various sub agents, the master relays the request to the chosen agents. The agents then create the actual tasks report back to the master, which in turn reports back to the initial agent and task.

5.2.4. Services

Services or service tasks within SAL are tasks that provide well known functions and services to other tasks. These services are typically persistent (much like a server process in an operating system) that wait to process requests from client tasks. They may be started at network boot time or demand-loaded as needed. The tasks themselves receive no special treatment nor use any special APIs; they are normal tasks within the agent system. Rather, an agent is defined by the services it provides via a well known and published messaging protocol. Service providers may be application-specific or general utility function providers. Currently, the service task design pattern may play a variety of roles.

A *monitor service* allows tasks to register themselves as monitors of other tasks, either being notified or perhaps killed when the watched tasks terminates. This service forms the basis for fault tolerant computing, providing cleanup services for when key tasks within a job abruptly terminate. An *administration service* can provide the basic functionality need for general system monitoring

DOI: 10.1002/spe

and administration. Client tasks can either perform automated, routine maintenance as well as present this information to the user, both graphically and in a report manner. A *job manager service* (where 'job' refers to a collection of cooperative tasks) is used to track user-visible jobs in the system. This allows a user to 'detach' or disconnect their client application from the agent system and leave their jobs running unattended. Upon return, the user is presented with a list of jobs (and their completion states). The user then resumes interacting with a selected job. Specific devices, instruments, and license-limited software could be accessed through a *representative service*. This allows a unique resource to be protected and managed by a sole process, with which all tasks must submit requests. For specific applications, *pseudo-random number generation* may also be centralized. This allows job reproducibility (critical for algorithm testing and development, and scientific publishing of biomedical research), as a distributed set of tasks must still contact a single, managing source for their random number sequences. Finally, a *file or data set service* may provide centralized access to data files. This may be done for ease of use (consolidation of all the files into one name space), access control or simply because the files are only available at fixed agents/hosts (this is particularly useful for cluster configurations without a shared file system). Arbitrary user authentication and access control may also be implemented to further refine the files available to a particular task or job set.

6. GRAPHICAL USER INTERFACE LIBRARY

This subsystem provides a basic graphical API wrapped around GTK+ [28] and consists of widget and window classes that become the foundation for all GUI widgets in Scopira. More specialized and complex widgets, particularly useful to numerical computing and visualization, are also provided. This includes widgets useful for the display of matrices, 2D images, bar plots, and line plots. Developers can use the basic GUI components provided to create more complex viewers for a particular application domain.

6.1. Specialized interface widgets

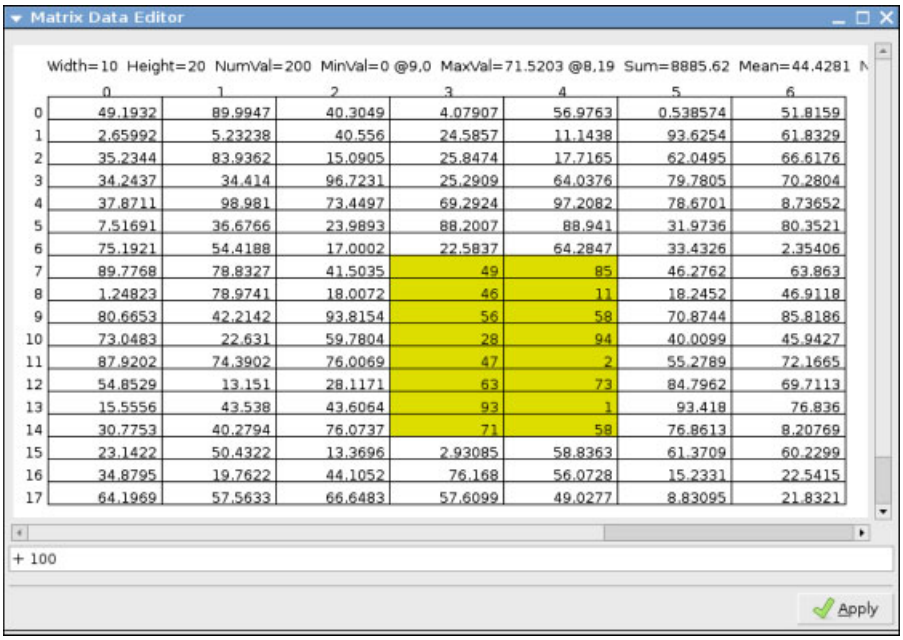
The Scopira GUI subsystem provides useful user-interface tools (widgets) for the construction of graphical, scientific applications, with particular focus on the biomedical research domain. These widgets complement the generic widgets provided by the GTK+ widget library with additional widgets for the visualization and inspection of numeric array data.

A matrix/spreadsheet-like widget (see Figure 7) is able to view and edit arrays (often, but not limited to matrices) of any size. This extensible widget is able to operate on Scopira *narray*s natively. The widget supports advances functionality such as bulk editing via an easy to use, stack-based macro-language. This macro-language supports a variety of operations of setting, copying, and filter selecting data within the array.

A generic plotting widget (see Figure 1) allows the values of Scopira *narray*s to be plotted. The plotter supports a variety of plotting styles and criteria, and the user-interface allows for zooming, panning and other user customizations of the plot.

An image viewer (see Figure 1) allows fully zooming, panning, and scaling of *narray*s, useful for the display of image data. The viewer supports arbitrary color mapping, includes a legend display and supports a tiled view for displaying a collection of many images simultaneously.

DOI: 10.1002/spe



	0	1	2	3	4	5	6
0	49.1932	89.9947	40.3049	4.07907	56.9763	0.538574	51.8159
1	2.65992	5.23238	40.556	24.5857	11.1438	93.6254	61.8329
2	35.2344	83.9362	15.0905	25.8474	17.7165	62.0495	66.6176
3	34.2437	34.414	96.7231	25.2909	64.0376	79.7805	70.2804
4	37.8711	98.981	73.4497	69.2924	97.2082	78.6701	8.73652
5	7.51691	36.6766	23.9893	88.2007	88.941	31.9736	80.3521
6	75.1921	54.4188	17.0002	22.5837	64.2847	33.4326	2.35406
7	89.7768	78.8327	41.5035	49	85	46.2762	63.863
8	1.24823	78.9741	18.0072	46	11	18.2452	46.9118
9	80.6653	42.2142	93.8154	56	58	70.8744	85.8186
10	73.0483	22.631	59.7804	28	94	40.0099	45.9427
11	87.9202	74.3902	76.0069	47	2	55.2789	72.1665
12	54.8529	13.151	28.1171	63	73	84.7962	69.7113
13	15.5556	43.538	43.6064	93	1	93.418	76.836
14	30.7753	40.2794	76.0737	71	58	76.8613	8.20769
15	23.1422	50.4322	13.3696	2.93085	58.8363	61.3709	60.2299
16	34.8795	19.7622	44.1052	76.168	56.0728	15.2331	22.5415
17	64.1969	57.5633	66.6483	57.6099	49.0277	8.83095	21.8321

Figure 7. Matrix editor.

Miscellaneous widgets such as a 'joystick' control (that permits discrete, cardinal direction panning), VCR buttons (that present 'play', 'pause', etc. type buttons) and a random seed editor are also provided. A simplified drawing canvas interface is included that permits developers to quickly and easily build their own custom widgets. Finally, Scopira provides a *Lab* facility to rapidly prototype and implement algorithms that need casual graphical output. Users code their algorithm as per usual, and a background thread handles the updating of the graphical subsystem and event loop.

6.2. Model/view plug-in framework

Scopira provides an architecture for logically separating models (data types) and views (graphical widgets that present or operate on that data) in the application. This view–model relationship is then registered at runtime. At runtime, Scopira pairs the compatible models and views for presentation to the user. A collection of utility classes for the easy registration of typical objects types such as data models and views are provided. This registration mechanism succeeds regardless of how the code was loaded; be it as part of the application, as a linked code library, or as an external plug-in.

Third parties can easily extend a Scopira application that utilizes models and views extensively. Third-party developers need only register new views on the existing data models in an application, then load their plug-in alongside the application to immediately add new functionality to the application. The open-source C++ image processing and registration library ITK [33,34] have been successfully integrated into Scopira applications at runtime using the registration subsystem.

DOI: 10.1002/spe

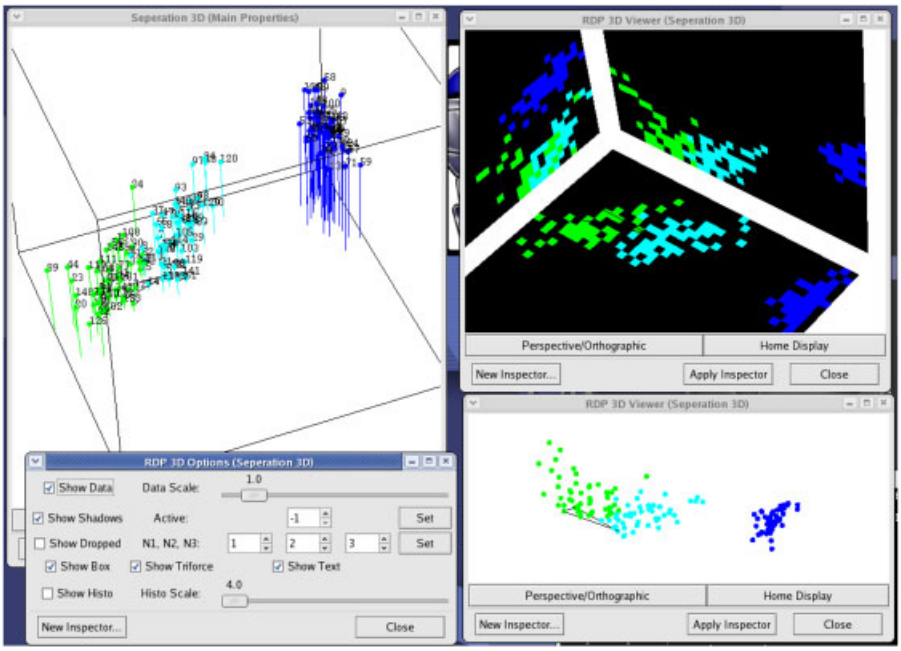


Figure 8. An example of Scopira's 3D visualization facilities.

A *model* is defined as an object that contains data and is able to be *monitored* by zero or more *views*. A *view* is an object that is able to bind to and listen to a model. Typically, views are graphical in nature, but in Scopira non-graphical views are also possible. A *project* is a specialized model that may contain a collection of models and organize them in a hierarchical manner. Full graphical Scopira applications are typically project-oriented, allowing the user to easily work with many data models in a collective manner. A basic project-based application framework is provided for developers to quickly build GUI applications using *models* and *views*.

6.3. 3D visualization

A complementary subsystem provides the base OpenGL-enabled widget class that utilizes the GTKGLExt library [35]. The GTKGLExt library enables GTK+-based applications to utilize OpenGL for 2D and 3D visualization. Scopira developers can use this system to build 3D visualization views and widgets, which allows for greater data exploration and processing (an example of which is in Figure 8). Integration with more complete visualization packages such as VTK [25] is also possible.

7. CONCLUSION

Overall, Scopira has successfully achieved its main objectives. Its strength lies in three major areas: core numerical algorithm development, parallel computing, and graphical application development.

DOI: 10.1002/spe

Its numerical array data structures, algorithms, and general tools provide our C++ developers a high-performance, robust, and common tool kit to use when developing and exchanging algorithm ideas and implementations. Often we prototyped and experimented with algorithms in slower, interpreted languages, but then re-implemented the algorithm in C++ and Scopira. This final version was much faster, used less memory, and was much easier to deploy than the original interpreted version. Scopira's existing algorithms and useful runtime error-checking also aided developers who would have used raw C++ facilities otherwise.

The embeddable Scopira Agents Library (SAL) also permits the rapid parallelization of algorithms that need to scale to more processors. This library is embeddable within Scopira, and unlike MPI is designed for easy end user deployment. Developer's efforts into parallelizing their applications are not only utilized in house during experimentation, but are preserved through to deployment and usable with even novice users.

Finally, Scopira aids in the development of user deployable applications by providing a complete multi-platform foundation and collection of visualization widgets for building rich, interactive applications. By making their numerical programs interactive and easier to use, users are encouraged to explore and utilize the programs while lowering the barrier of entry for new users.

REFERENCES

1. Huettel SA, Song AW, McCarthy G. *Functional Magnetic Resonance Imaging*. Sinauer Associates: Sunderland, 2004.
2. Visual Information Solutions. http://rsinc.com/idl [10 July 2008].
3. Bowman KP. *An Introduction to Programming with IDL*. Elsevier: Burlington, 2006.
4. The MathWorks. http://www.mathworks.com [10 July 2008].
5. Sigmon K, Davis TA. *Matlab Primer*. CRC Press: Boca Raton, 2004.
6. Demko AB, Pizzi NJ, Somorjai RL. Scopira—A system for the analysis of biomedical data. *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, Winnipeg, Canada, 12–15 May 2002; 1093–1098.
7. Message Passing Interface Forum. http://www.mpi-forum.org [10 July 2008].
8. Snir M, Gropp W. *MPI*: *The Complete Reference*. MIT Press: Cambridge, 1998.
9. PVM: Parallel Virtual Machine. http://www.csm.ornl.gov/pvm [10 July 2008].
10. Geist A, Beguelin A, Dongarra J, Jiang W, Manchek R, Sunderam VS. *PVM*: *Parallel Virtual Machine*: *A Users' Guide and Tutorial for Network Parallel Computing*. MIT Press: Cambridge, 1994.
11. OpenGL®: The Industry's Foundation for High Performance Graphics. http://www.opengl.org [10 July 2008].
12. Hill FS, Kelley SM. *Computer Graphics Using OpenGL*. Prentice-Hall: Upper Saddle River, 2006.
13. Pizzi N, Vivanco R, Somorjai RL. EvIdent: A functional magnetic resonance image analysis system. *Artificial Intelligence in Medicine* 2001; **21**:263–269.
14. Bezdek J, Ehrlich R, Full W. FCM: The fuzzy c-means clustering algorithm. *Computational Geosciences* 1984; **10**: 191–203.
15. Pizzi NJ. Classification of biomedical spectra using stochastic feature selection. *Neural Network World* 2005; **15**(3): 257–268.
16. Huang D, Swanson EA, Lin CP, Schuman JS, Stinson WG, Chang W, Hee MR, Flotte T, Gregory K, Puliafito CA, Fujimoto JG. Optical coherence tomography. *Science* 1991; **254**:1178–1181.
17. Brezinski ME, Tearney GJ, Boppart SA, Swanson EA, Southern JF, Fujimoto JG. Optical biopsy with optical coherence tomography: Feasibility for surgical diagnostics. *Journal of Surgical Research* 1997; **71**:32–40.
18. MFC Feature Pack for Visual C++ 2008. http://msdn.microsoft.com/en-us/library/bb982354.aspx [10 November 2008].
19. Jones R. *Introduction to MFC Programming with Visual* C++. Prentice-Hall: Upper Saddle River, 2000.
20. Qt Cross-Platform Application Framework. http://trolltech.com/products [10 November 2008].
21. Dalheimer MK. *Programming with Qt*. O'Reilly Media: Cambridge, 2002.
22. WxWidgets: Cross-Platform GUI Library. http://www.wxwidgets.org/ [10 November 2008].
23. Smart J, Hock K, Csomor S. *Cross-platform GUI Programming with wxWidgets*. Prentice-Hall: Upper Saddle River, 2005.
24. VTK: The Visualization Toolkit. http://www.vtk.org [10 July 2008].
25. Schroeder W, Martin K, Lorensen B. *Visualization Toolkit*: *An Object-oriented Approach to 3D Graphics*. Kitware: Clifton Park, 2006.

DOI: 10.1002/spe

26. Boost C++ Libraries. http://www.boost.org [10 July 2008].
27. Karlsson B. *Beyond the C++ Standard Library*: *An Introduction to Boost*. Addison-Wesley Professional: Reading, 2005.
28. The GTK+ Project. http://www.gtk.org [10 July 2008].
29. Krause A. *Foundations of GTK+ Development*. Springer: New York, 2007.
30. Blitz++: Object-oriented Scientific Computing. http://www.oonumerics.org/blitz [10 July 2008].
31. Shan H, Singh JP, Oliker L, Biswas R. Message passing and shared address space parallelism on an SMP cluster. *Parallel Computing* 2003; **29**:167–186.
32. Caglar SG, Benson GD, Huang Q, Chu C-W. USFMPI: A multi-threaded implementation of MPI for Linux clusters. *Proceedings of the International Conference on Parallel and Distributed Computing and Systems*, Marina del Rey, U.S.A., 3–5 November 2003; 104–109.
33. ITK: NLM Insight, Segmentation & Registration Toolkit. http://www.itk.org [10 July 2008].
34. Ibanez L, Schroeder W. *The ITK Software Guide 2.4*. Kitware: Clifton Park, 2005.
35. GTKGLExt: Main Page. http://www.k-3d.org/gtkglext/ [10 July 2008].

DOI: 10.1002/spe