

Solving the fundamental problems of high-performance software development

By William Lundgren, Kerry Barnes, and James Steed

Programming languages must provide multiprocessor information to successfully move and process data in high-performance software. In this analysis, William, Kerry, and James present three algorithms, explain how to implement them efficiently on a multiprocessor target, and show how the Geda multiprocessor programming language can automate the implementation.

The fundamental problems of high-performance software development are how to get data into fast memory and how to process it efficiently from that memory. This may seem like an oversimplification, but consider the work that goes into both moving and processing data when programming a multiprocessor system under tight real-time processing requirements. The task of getting data into fast memory includes data movement both on a single processor (for example, cache utilization) and between processors. The application must be partitioned between processors to keep the load balanced and to realize the benefit of parallelization while avoiding deadlock. The task of processing data efficiently involves using every possible processing path. Compute-intensive portions of the application must be structured so that all arithmetic logic units are fully utilized, and Direct Memory Access (DMA) processing units must be fully utilized to ensure data transfer and processing happen simultaneously.

A few tools available today address software development for multiprocessor architectures. Current compilers are based on the assumption that the hardware is a von Neumann architecture (that is, a single processor with sequential operations). Developing multiprocessor solutions with current compilers requires the developer to be aware of the architecture of the target, as shown in Figure 1a, and embed the details of the multiprocessor behavior in the code. Embedding this target-specific information in the code creates untenable software complexity.

Easing the complexity of programming a multiprocessor or multicore architecture requires a compiler to be aware of all processors in the target architecture, as shown in Figure 1b. The multiprocessor compiler collects global information about the application, such as interprocessor communication, memory use, and CPU load, and uses that information to optimize resource utilization.

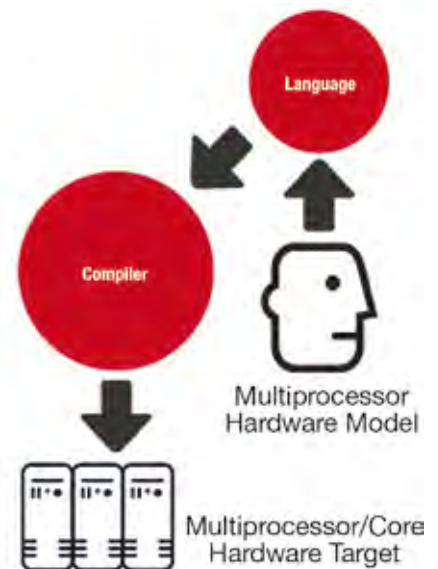


Figure 1a

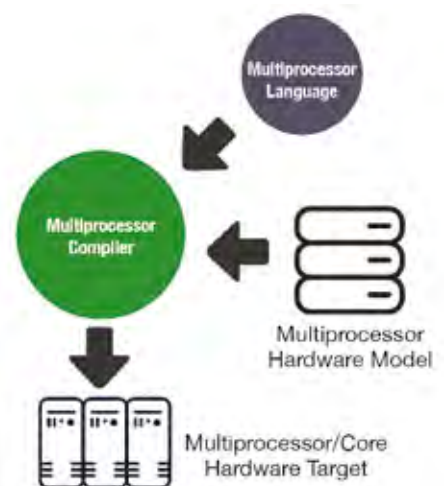


Figure 1b

To make use of a multiprocessor compiler, the programming language must provide the information required to collect this global information. The languages designed to work with von Neumann architectures unfortunately don't provide enough information. Therefore, a new language must be developed alongside the compiler to provide multiprocessor information. This programming language can be built on software libraries as long as they are kept simple with low overhead; libraries encapsulating complex behavior hide the detail required for the multiprocessor compiler to collect global information on the application.

Gedae is a multiprocessor programming language and compiler that targets multiprocessor systems, including multicore architectures such as the Cell Broadband Engine, and chip architectures, including general purpose processors, DSPs, and FPGAs. The following discussion will demonstrate how Gedae can help simplify high-performance software development.

Simple specification of data distribution

In the first example application, a distributed Euler method, multiple cores or processors are used in parallel to efficiently process the data set. The Euler method

is an iterative algorithm that performs a 3 x 3 neighborhood operation on a data matrix. To perform this algorithm in parallel, the developer must choose a data distribution of the matrix that minimizes the amount of interprocessor communication. For this example, the data is distributed row-wise in blocks; that is, if there are N rows and P processors, then the first N/P rows are given to the first processor, second N/P rows are given to the second processor, and so on.

For each iteration, every processor calculates a new value for its portion of the matrix. Some of these new values must be shared with adjacent processors to perform the next iteration. With this distribution, rows from adjacent processors must be communicated at each step in the iteration, as shown in Figure 2. Zero rows are inserted as boundary conditions in the top of the 0th processor's data and the bottom of the (P-1)st processor's data. As the Euler method iterates on the data matrix, this data movement ensures each processor has the data necessary to perform the 3 x 3 neighborhood operation. The communication pattern is regular – two rows are transferred to and from each processor at each iteration.

Gedae assists with the implementation of this application by automatically inserting all communication required to process the data set in parallel. As shown in Figure 3, the developer has two tasks: express the arithmetic of the algorithm (as encapsulated in the StepOnce node) and express the data distribution (as done by families of m_selrowK and m_addrows nodes). The specification of the data distribution

Gedae assists with the implementation of this application by automatically inserting all communication required to process the data set in parallel.

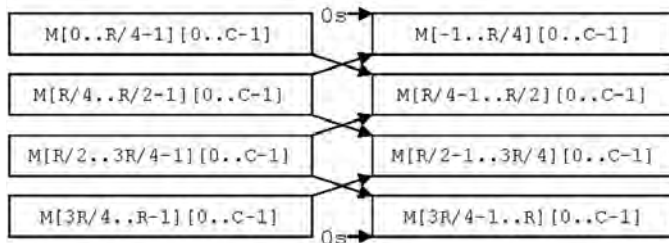


Figure 2

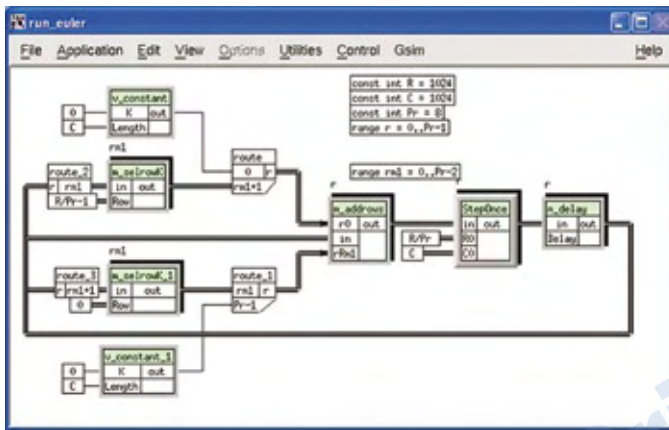


Figure 3

in this flow graph closely mirrors the data movement diagram in Figure 2. The range variables r and $rm1$ create an array of processing paths called a *family* when applied to the nodes in the graph, as indicated by the shadowing behind the nodes and arcs. The $m_selrowK$ nodes select a row of data from the matrix to transfer to adjacent processors. The $v_constant$ node provides zero rows, and $m_addrows$ concatenates the data from other processors to the rows on the local processor.

To create the parallel implementation, the developer uses Geda's implementation dialog to map the i^{th} member of the family of nodes to the i^{th} processor in the multiprocessor system. Geda automatically inserts the necessary communication, but the developer has many degrees of freedom over what is added. In this example, the developer makes sure the communication is nonblocking and DMA transfers are utilized. To process the distributed data efficiently on each processor, Geda builds its primitives atop a vector library called the *E library*, which provides basic arithmetic and signal processing operations. This E library is a light wrapper over hand-optimized routines created by the hardware vendor.

Dealing with lightweight processors

The second example application is a distributed sort application on a memory-limited architecture where the data set will not fit in a single processor's memory. An example of such an architecture is the

Cell Broadband Engine where each of the eight synergistic processing elements has 256 KB of local storage, with a larger bulk storage shared among all PEs. Similar to the finite element analysis in the first example, the data must be decomposed and distributed across multiple processors. However, in this application, memory is also a limiting factor, not just required throughput.

Assume at the beginning of the sort that each processor has N/P data elements where N and P are the data size and number of processors. To sort the distributed array in this algorithm, each processor locally calculates a histogram, and the histograms are collected to determine how the data should be redistributed. Each iteration of the algorithm has two sections of interprocessor communication. The first section collects the histogram on a single processor. This communication contains a small amount of data, but all data must go to and from one processor. Once the histogram is collected, the data redistribution is calculated and the information is broadcast to all processors. The second section of communication is the transfer of array data between processors according to the histogram results. The communication in the data redistribution is not regular; each data element may be sent to any processor, and potentially several elements are transferred during any iteration. Each processor merges the data left from the previous iteration and the new data received from other processors, resulting in sorted local data.

To implement this application, the developer must use memory efficiently so that the generated application fits within the architecture's limited resources. Gedae statically allocates all memory usage and includes several memory-packing algorithms to shrink the size of the allocated buffers. During compilation, Gedae analyzes the memory use at each stage in the statically determined order of execution and determines when each memory buffer can be reused. Memory used in the histogram calculation can be reused in later parts of the algorithm, such as when data is received from other processors and merged with the remaining local data. Furthermore, the flow graph language provides for the specification of *in-place* streams where inputs and outputs use the same memory buffer. For example, a single N/P in-place buffer on each processor can perform the merge operation in this example.

Gedae automatically inserts and manages all communication needed to sort the data, freeing the developer to concentrate on the algorithmic details of performing a distributed histogram and determining the data redistribution at each step. While the Euler method example has only $2*(P-1)$ transfers per iteration, this example adds many sends and receives to create the implementation. This inserted communication is displayed as green (send) and red (receive) bars in the Trace Table, as shown in Figure 4. The row labeled *part_control* in the table shows the execution of the distributed histogram calculation. This row is collapsed so that all the components are displayed in one line, and in this line the several send and receive pairs dominate the execution time of this portion of the application.

The rows labeled *vui_part_vvui* show each processor partitioning the data after

being told by the control processor how to redistribute it, and the *vvui_nconcat_vui* and *vui_sort* rows show each processor combining and merging the new data with the old. In this example, Gedae minimizes the work the developer has to do to implement the intricate communication patterns and helps make sure the application remains memory efficient.

Model-driven development deployed

The third example application is a radar application, which has two different operation modes and must process data at a high data rate. To meet this high data rate, each processor must be highly utilized. Because different modes have different execution times and require a different number of processors to execute, the load must be balanced by sending data to idle processors to reach high utilization. This example application has two modes. Mode A only requires one processor and has a short duration. Mode B requires two processors, and its duration is four times as long as Mode A. A round-robin scheduling algorithm is used for managing which processors are busy and determining where to process the next mode; that is, a manager processor keeps track of which processors are busy and sends new instances of modes to the first idle processor in the queue.

Gedae helps the developer solve this problem by simplifying the operation modes' specification and bringing the scheduling algorithm to the forefront while automating its implementation and data movement. The operation modes' specification is shown in Figure 5. This flow graph is executed on each of the processors. The scheduling algorithm determines which processor to process the mode on and sends this data downstream in a packet. When the processor is idle, it waits for a packet on the source of this flow graph. When it receives a packet from the control processor, the *cp_decode* node decodes it, extracting the mode ID, and branches the data to the subgraph, which implements the desired mode of operation. Thus, any processor can execute either mode, with the scheduling algorithm instructing each processor when to start processing data and which algorithm to use on that data.

This round-robin scheduling algorithm aptly utilizes multiple processors. Figure 6 shows a Trace Table for this radar application. The horizontal black bars indicate the processor is busy. The long black blocks on processors 101 and 102 indicate an execution of Mode B where the duration is four times longer than that of Mode A. If Mode B is not being run, the

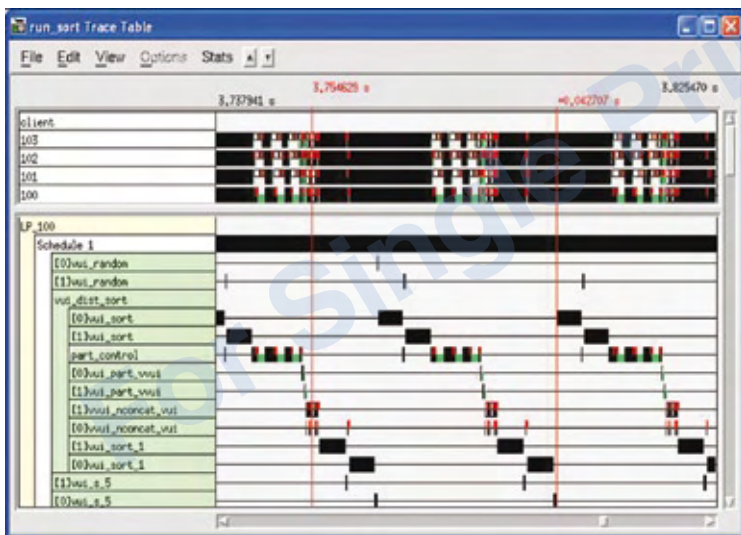


Figure 4

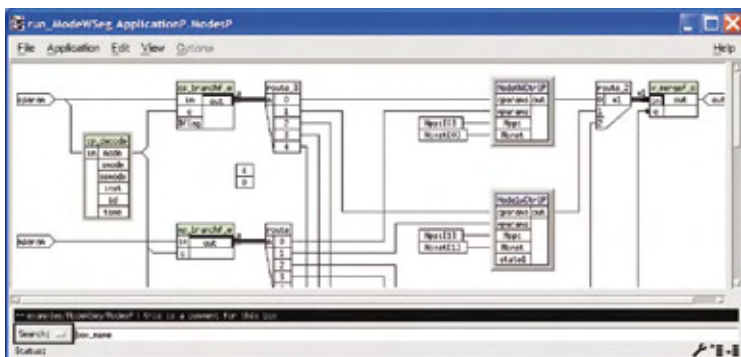


Figure 5



Figure 6

round-robin scheduler is free to send instances of Mode A to the processors used by B, as shown in the application's trace. This example implementation is inspired by an actual deployed radar system where the software modes, mode control, and dynamic load balancing were implemented, distributed, and deployed with Gedae.

Exposing what is pertinent; automating what is mundane

Creating high-performance software requires both moving data into fast memory and processing data once it gets there. To assist with high-performance software development, the Gedae model-driven development tool automates the mundane details so the developer does not have to implement them.

Further, Gedae exposes the pertinent details so the developer can code and optimize the application more effectively. Data distribution is a natural part of the flow graph language, as shown in the Euler method example. By directly accessing the vendor's optimized vector library, the application can run efficiently. Intricate communication patterns are implemented automatically, as shown in the array sort example, and the generated implementation's communication web can be easily viewed in the Trace Table. Fully deployed software systems can also be created, such as in the radar example where a round-robin scheduling scheme distributes tasks of different duration and processing requirements on many processors. Gedae significantly simplifies

high-performance software development without detriment to the delivered product's efficiency. **ECD**

William Lundgren is cofounder, president, and CEO of Gedae, Inc. William started his professional career at Corning Glass Works as a product development physicist. He later worked at the U.S. Air Force Institute of Technology and the U.S. Air Force Research Laboratories developing new speech and audio processing technologies. After leaving active duty in 1985, he moved to RCA Advanced Technology Laboratories, which became Lockheed Martin, and spent 16 years leading Gedae development. William has a BS in Physics from Rensselaer Polytechnic University, BS and MS degrees in Electrical Engineering from the U.S. Air Force Institute of Technology, and is All But Dissertation for his PhD in Electrical Engineering from the University of Pennsylvania.



Kerry Barnes is chief scientist and a founding member of Gedae. Before joining Gedae, Kerry was a principal member of the engineering staff at Lockheed Martin, ATL where he was responsible for signal processing systems software/hardware, single-chip



FFT design, direct digital frequency synthesizer design and implementation, and various software tools and applications development projects. He earned a BS in Electrical Engineering from Lehigh University and an MS in Computer and Information Science from the University of Pennsylvania.

James Steed is director of software development and a founding member of Gedae. Prior to Gedae, James worked at Lockheed Martin where he was responsible for developing the embeddable library of functions, including testing and creating a database and search utility. His most prominent project is the development of Gedae's new RTL language. James earned a BS in Computer Science from Cornell University and an MS in Computer Science from North Carolina State University.



To learn more, contact the authors at:

Gedae, Inc.

1247 North Church Street, Suite 5
Moorestown, NJ 08057
856-231-4458
wlundgren@gedae.com
kbarnes@gedae.com
jsteed@gedae.com
www.gedae.com

What is Gedae?

Gedae is an integrated application development environment. It has a language for describing an architecture-independent functional specification, a multiprocessor virtual machine, and a multiprocessor compiler that transform the specification into one that runs on a virtual machine. The following topics are key to understanding Gedae:

- **Language:** The language was developed with two requirements: All functionality must be easily expressible and the language must be transformable into an efficient implementation on the virtual machine. The Gedae language consists of the Gedae Primitive Language and the Gedae Graph Language. The Primitive Language is geared toward expressiveness, while the Graph Language allows for the hierarchical development of graphs containing primitives, parameters, and other Gedae graphs. The resulting language permits direct expression of signal and data processing algorithms, distribution for providing load balancing and fault tolerance, and application control.
- **Virtual machine:** The language and machine were codesigned to achieve maximum efficiency. The virtual machine contains a runtime kernel that executes components generated by the compiler. The virtual machine also allows for vendor-specific processing optimizations, such as setting data transfer parameters. A thin layer over the vendor-provided vector-processing libraries allows primitives to execute efficiently.
- **Compiler:** As the central part of Gedae, the multiprocessor compiler builds an efficient application from code expressed in the Gedae language to run on the Gedae virtual machine. The compiler passes are automated but can be guided by the user-supplied implementation parameters to control distribution, strip mining, data transfers, scheduling priorities, queue policies, and memory management.