

# Ouroboros: A Tool for Building Generic, Hybrid, Divide & Conquer Algorithms

*J. R. Johnson, I. Foster*

This article was submitted to  
Supercomputing 2003, Phoenix, AZ, November 15-21, 2003

**May 1, 2003**

*U.S. Department of Energy*

Lawrence  
Livermore  
National  
Laboratory

## DISCLAIMER

This document was prepared as an account of work sponsored by an agency of the United States Government. Neither the United States Government nor the University of California nor any of their employees, makes any warranty, express or implied, or assumes any legal liability or responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or the University of California, and shall not be used for advertising or product endorsement purposes.

This is a preprint of a paper intended for publication in a journal or proceedings. Since changes may be made before publication, this preprint is made available with the understanding that it will not be cited or reproduced without the permission of the author.

This work was performed under the auspices of the United States Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

This report has been reproduced directly from the best available copy.

Available electronically at <http://www.doc.gov/bridge>

Available for a processing fee to U.S. Department of Energy  
And its contractors in paper from  
U.S. Department of Energy  
Office of Scientific and Technical Information  
P.O. Box 62  
Oak Ridge, TN 37831-0062  
Telephone: (865) 576-8401  
Facsimile: (865) 576-5728  
E-mail: [reports@adonis.osti.gov](mailto:reports@adonis.osti.gov)

Available for the sale to the public from  
U.S. Department of Commerce  
National Technical Information Service  
5285 Port Royal Road  
Springfield, VA 22161  
Telephone: (800) 553-6847  
Facsimile: (703) 605-6900  
E-mail: [orders@ntis.fedworld.gov](mailto:orders@ntis.fedworld.gov)  
Online ordering: <http://www.ntis.gov/ordering.htm>

OR

Lawrence Livermore National Laboratory  
Technical Information Department's Digital Library  
<http://www.llnl.gov/tid/Library.html>

# Ouroboros: A Tool for Building Generic, Hybrid, Divide & Conquer Algorithms\*

John R. Johnson<sup>1,2</sup>

Ian Foster<sup>1,3</sup>

<sup>1</sup>Department of Computer Science, University of Chicago, Chicago, IL 60637, U.S.A.

<sup>2</sup>Computing Applns. & Research Dept., Lawrence Livermore Natl. Lab., Livermore, CA, 94551, U.S.A

<sup>3</sup>Math and Computer Science Division, Argonne National Laboratory, Argonne, IL 60439, U.S.A.

## Abstract

A hybrid divide and conquer algorithm is one that switches from a divide and conquer to an iterative strategy at a specified problem size. Such algorithms can provide significant performance improvements relative to alternatives that use a single strategy. However, the identification of the optimal problem size at which to switch for a particular algorithm and platform can be challenging. We describe an automated approach to this problem that first conducts experiments to explore the performance space on a particular platform and then uses the resulting performance data to construct an optimal hybrid algorithm on that platform. We implement this technique in a tool, *Ouroboros*, that automatically constructs a high-performance hybrid algorithm from a set of registered algorithms. We present results obtained with this tool for several classical divide and conquer algorithms, including matrix multiply and sorting, and report speedups of up to six times achieved over non-hybrid algorithms.

## 1 Introduction

We address the challenge of designing *performance portable algorithms*, i.e., algorithms that can perform well, without manual intervention, on a wide variety of platforms.

The traditional analytic approach to this problem applies the *macroscope* of asymptotic analysis to an algorithm within the abstraction of an *ideal* RAM machine in which all memory locations

are equidistant and instructions are issued sequentially and in order. Asymptotic analysis and the RAM model represent an effective framework for classifying and relating abstract algorithms that can provide useful guidance to the algorithm designer. However, the performance of real computational systems deviates from that of ideal machines, to degrees that can range from inconsequential to pathological. Indeed, interactions between the structure and parameterization of an algorithm and the architectural and software characteristics of a target system fundamentally determines the overall performance of the algorithm. Favorable and unfavorable algorithm structure–system architecture combinations can differ in performance by orders of magnitude.

There are many aspects of algorithm/architecture structure that influence performance and the potential parameter space can be intractably large for an exhaustive analysis. We focus here on the important class of divide and conquer (D&C) problems and the single algorithmic parameter of problem size.

It is well known that the performance of D&C algorithms such as sorting, matrix multiply, and fast Fourier transform (FFT) can be improved significantly by switching to an iterative implementation when the problem size drops below a certain machine-dependent threshold [6]. The optimal algorithm is thus a hybrid. However, for a given operation, such as sorting or matrix multiply, there may be a plethora of potential algorithms to use when building a hybrid. Selecting the appropriate choice of algorithms and switching point to create the hybrid implementation for a given instance can be a daunting task for the algorithm/application designer even when the target system is known. When the target system is unknown at development time—as in the case of a portable library, a library with a lifetime that will span

---

\* This work was performed under the auspices of the U. S. Department of Energy by the University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48.

multiple hardware generations, or a Grid computation—it may not be possible to design an algorithm analytically that will run near optimally on all potential systems.

Our approach to this problem, incorporated in the Ouroboros system, is to use empirical methods to determine automatically a collection of implementations that together constitute a satisfactory high-performance hybrid code. In all cases that we have studied to date, Ouroboros succeeds in detecting automatically the best threshold for switching between D&C and iterative implementations. In the case of sorting, Ouroboros produces a hybrid algorithm that is as much as six times faster than either the iterative or D&C approach alone.

## 2 Related Work

Several analytic models have been proposed for describing and predicting arbitrary machine performance. Some models focus on narrow substructures of the algorithm/architecture interface (e.g., communication or I/O) [8, 17], while others are applied to large applications [1, 3, 16, 23]. However, these models typically provide large performance bounds and/or have narrow scope.

Another approach to algorithm design combines empirical methods with techniques for exploiting a priori knowledge of the general structure of components in a hierarchical memory system (such as register/cache blocking, loop unrolling, etc.) [2, 12, 13, 18, 22, 24, 28]. Typically these *empirical adaptive algorithms* are generated automatically from a code suite that performs extensive testing (offline or online) on a particular platform to identify good configurations. These successful studies have been applied successfully to generate highly-tuned kernel operations, and provide an example of algorithms that can be expected to perform well on a wide variety of machine/system architectures.

Ouroboros synthesizes the concepts developed in these early studies into a generalized framework and tool to aid the algorithm designer. It is distinct from these approaches in that it employs an analytical framework based solely on the *behavior* of an instance of an algorithm on a particular system rather than the interaction of its *structure* with the system and thus relieves the algorithm designer from explicitly understanding all the nuances of a given algorithm implementation or its pathological behavior on a

runtime system. This purposeful relinquishing of detail still yields high performance algorithms while allowing the framework to be more general.

## 3 Analytic Framework

A traditional analysis expresses a D&C algorithm in terms of a recurrence relation:

$$f(n) = \sum a_i f(n/b_i) + c(n) \quad (1)$$

where the  $a_i$  and  $b_i$  are constants and  $c$  is a function describing the cost of splitting and merging at the given level of recursion [6].

The behaviorally based analytical framework employed in Ouroboros uses a more restrictive recurrence:

$$f(n) = \sum a_i f_\sigma(\sigma_i) + c_i(n) \quad (2)$$

Here,  $\sigma_i = n/b_i$ ,  $c(n)$  is a function of  $n$ , and  $a_i$  and  $b_i$  are constants.  $\sigma_i$  represents the size of the problem at level  $i$ .  $f_\sigma$  is the function to be applied to a problem of size  $\sigma$ .  $c_i(n)$  is the cost of the divide and combine at this level of the recurrence.

The distinction between (1) and (2) is in the choice of  $f_\sigma$ . To make the description of  $f_\sigma$  explicit we use the following formalism.

Let  $\Phi$  be a family of functions,  $\Phi = \{f_0, \dots, f_j\}$ . Define the relation that admits a function to this set as follows:

- 1) If  $f_i$  and  $f_j$  are in  $\Phi$  then  $f_i$  and  $f_j$  operate on the same data type and format and have the same signature.
- 2) If  $f_i$  and  $f_j$  are in  $\Phi$  then  $f_i$  and  $f_j$  perform the “same” computation.

Now for each data size,  $s$ , we can induce an ordering on  $\Phi$  based on the performance of each  $f_i$  on the  $s$  for a particular machine.  $F_\sigma$  is then defined to be the “best” performing function for a data size  $\sigma$ . (For clarity, we have dropped the explicit reference to the machine in our notation. It is to be understood that  $\sigma$  refers to  $\sigma_\mu$  for some machine  $\mu$ .)

We refer to (2) as a *generalized D&C (GDC) algorithm*.

## 4 Ouroboros

Ouroboros is a tool for implementing GDCs. It operates by performing empirical tests of a collection of registered algorithms on a particular architecture for a range of problem sizes, and then using the resulting data to construct a hybrid algorithm, via the selection of the optimal algorithm variants for each problem size. In other words, it computes and stores  $f_\sigma$  for each  $\sigma$  in a range of values.

We can see that for D&C algorithms there exists a problem size  $s_0$  such that for problem sizes  $s > s_0$ , a D&C approach is preferred to an iterate approach while for problems of size  $s < s_0$  the iterative version achieves better performance. We can express this assumption as follows:

$$\Sigma P(s) < P(\Sigma s) \quad \Sigma s > s_0 \quad (3)$$

where  $P(s)$  indicates the performance of the function on data of size  $s$ .

This expression implies that to address the threshold problem we need only test problem sizes  $\{1, \dots, s_0\}$ , where  $s_0$  is again understood to be machine dependent. Once the  $f_\sigma$  are determined for each  $s$  in  $\{1, \dots, s_0\}$  they can be stored for use at runtime.

However, there exist D&C problems that *depend* upon switching to an iterative approach for small sizes in order to perform better than a strictly iterative approach. For example, Strassen matrix multiply has an expected performance bound of  $\Theta(n^{\lg 7}) = O(n^{2.81})$ . But it also has a large hidden constant. If the recursion is allowed to continue to the smallest base state, then in practice, Strassen *always* performs worse than naïve matrix multiply, which has an expected  $O(n^3)$  performance bound. Thus, no crossover point,  $s_0$ , can be found and equation (3) does not hold.

We address this problem by developing a more restrictive formalization of the D&C assumption that captures the recursion sensitivity of an algorithm such as Strassen matrix multiply. We express this formalization as follows:

$$\Sigma P(s|\tau) < P(\Sigma(s|\tau)) \quad \Sigma s > s_0 \geq \tau \quad (4)$$

where  $\tau$  is the explicit crossover point and  $P(s|\tau)$  is the performance of the function on size  $s$  with crossover  $\tau$ . Without loss of generality,  $\tau$  is assumed to be 1 in the case of an iterative function.

We are now ready to test our algorithm variants. We assume that we can classify available

algorithms for a problem into two classes, iterative and recursive. We first test problem sizes  $\{1, \dots, s_0\}$  for the iterative function(s) and determine the  $f_\sigma$ . Next we test the recursive function(s). If we find a crossover point then we can use equation (3). If not, we rerun the tests varying  $\tau$  and using  $if_\tau$  for the iterative steps where  $if_\tau$  refers to the *best iterative* function for size  $\tau$ . From this,  $s_0$  can be defined as  $\min(\sigma)$  such that  $if_\sigma > rf_{\sigma\tau}$  for all  $\tau$  where  $rf_{\sigma\tau}$  is the *best recursive* function on size  $\sigma$  that switches to the best iterative function for size  $\tau$ .

At run-time a problem of a given size  $N$  is factored into problems of smaller sizes. Once the problem size drops below  $s_0$ , the appropriate  $f_\sigma$  is selected and applied to the problem.

Ouroboros implements this technique via the use of three components: function registration, installation test and analysis routines, and run-time configuration mechanisms.

### 4.1 Function Registration

This component allows the designer to register known algorithms for use by the *installation and test* and *run-time configuration* routines to construct a hybrid algorithm.

The designer supplies a collection of functions that all perform the “same” computation. Once registered, these functions are stored as an array of pointers. For each function pointer in the array there is also an associated metadata structure that describes the function. The metadata structure contains information such as the type of data on which the function operates (double, int, etc..), the function signature (e.g., for matrix multiply does it accept an additional array for scratch), the data-layout (e.g., 1-D array, HDF-5, etc..), constraints (e.g., power-of-two input), and annotations (i.e. anything else the algorithm designer would like to add).

The registry also maintains miscellaneous indexes that are used to filter functions with metadata constraints. For example, if the problem size is an odd prime, then functions that only operate on power-of-two are filtered. Or if the problem data type is double then integer-only functions are filtered.

If the function is recursive then the function’s source is instrumented replacing recursive calls with a wrapper that encapsulates the calls to the appropriate function for the given problem size.

#### 4.2 Installation Test & Analysis Routines

At build time, the test and analysis routines create a performance matrix for all registered functions by empirically testing each function on a range of data sizes (e.g., 1 to  $s$  for some  $s$ ). For a given data size, the functions are sorted by performance. This information is stored as an array that maps problem size to the function registry index of the function that performed best for that size. The test and analysis routines also empirically determine the value  $s_0$ , used to bound size of the performance matrix.

#### 4.3 Run-time Configuration

At run-time a problem of a given size  $N$  is factored into problems of smaller sizes. The factoring is done in order to maximize the use of factor sizes that have good performance. The result is a generated code, (e.g., `sort(int size, ...)`) that consists of a single switch that applies the appropriate registered function for the problem size `size`.

To show that this factoring can be done on a problem of size  $N$ , we make use of the notion of a *well-behaved GDC algorithm*. Essentially this requirement assumes that beyond a certain point,  $n_0$ , as size increases, the performance of the function decreases although not necessarily monotonically. Given this assumption, we just start at the best performing size  $< n_0$  and walk down the list of sizes in order of performance until we find all the factors less than  $n_0$  for our given problem size. Once we have exhausted this list, we select the remaining factors of  $N$ , all greater than  $n_0$  to be as small as possible.

## 5 Experimental Studies

We have applied Ouroboros to two classical algorithm problems: *sorting* (merge, insertion) and *dense matrix multiply* (Strassen, naïve). The data for each problem uses randomly generated double precision arrays. Tests are run on a single processors of the machines described in Table 1.

Table 1: Processors used for experiments

Processor	Processor Speed	L1 Cache Size	OS
POWER 3	375MHz	64K	AIX
Alpha Ev68	1GHz	64K	tru64_5

## 5.1 Sorting

### 5.1.1 POWER 3

Figure 1 shows the performance of insertion sort and merge sort for small problem sizes on the POWER 3. The x-axis represents the problem size (i.e. vector length). The y-axis represents time. We see that between problem sizes of 100 and 200 merge sort starts performing better than insertion sort. Ouroboros selected a crossover point of 124. In other words, Ouroboros chose to use merge sort until the problem size dropped below 124, at which point it switched to insertion sort. The resulting performance comparison between merge sort and the Ouroboros hybrid for larger sizes is shown in Figure 2. A speedup of greater than six times is achieved for some problem sizes.

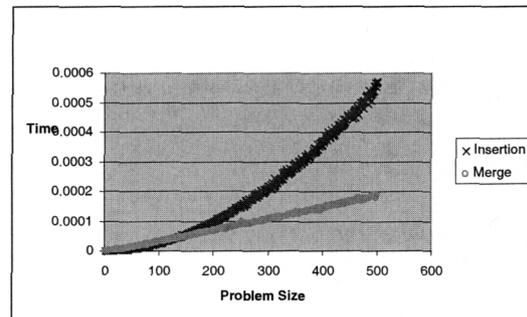


Figure 1: POWER 3 crossover detection

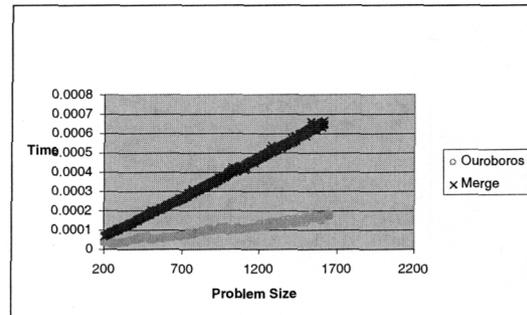
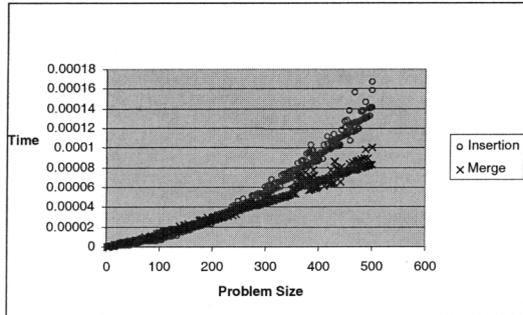


Figure 2: POWER 3 with crossover of 124

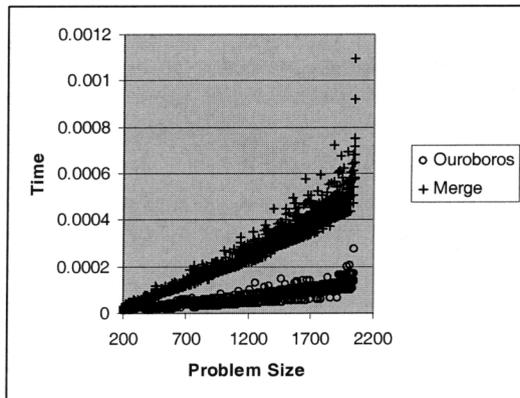
### 5.1.2 Alpha Ev68

Figure 3 shows the performance of insertion sort and merge sort for small problem sizes on the Alpha Ev68. From this figure the actual crossover point is less clear, but occurs somewhere between 190 and 300. Ouroboros selected 190.



**Figure 3: Alpha Ev68 detection of crossover**

The resulting performance comparison between merge sort and the Ouroboros hybrid for larger sizes is shown in Figure 4. We achieve a speedup of up to six times for some problem sizes.



**Figure 4: Alpha Ev68 with crossover of 190**

Note that the crossover point of 190 selected by Ouroboros on the Alpha Ev68 would also have worked well on the POWER 3. Similarly, the crossover point of 124 on the POWER 3 would also have produced good results on the Alpha Ev68. However, this correspondence is a fortuitous artifact of the way in which Ouroboros selects the crossover point and not necessarily inherent in the interaction of the algorithm with the architecture. For example, a crossover point of 295 could have been chosen for the Alpha Ev68 without significantly changing its performance signature for larger problem sizes. If, however, this value were used on POWER 3 rather than 124, the speedup, as can be seen from Figure 1, would be diminished.

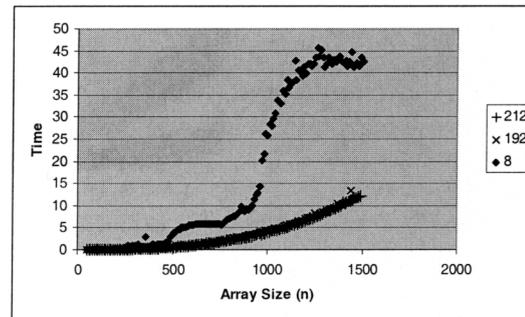
## 5.2 Matrix Multiply

The test suite for matrix multiply uses the *gemmw* implementation of Winograd's variant of Strassen matrix multiply and the BLAS 3 *gemm* implementation of naïve matrix multiply, both available from *netlib*. It should be noted that the

distribution of *gemmw* is configured to crossover from the recursive *gemmw* to the iterative BLAS 3 *gemm* at a handful of hardcoded machine-dependent sizes. For the tests described here, we modified the code to allow this parameter to vary in the Ouroboros framework.

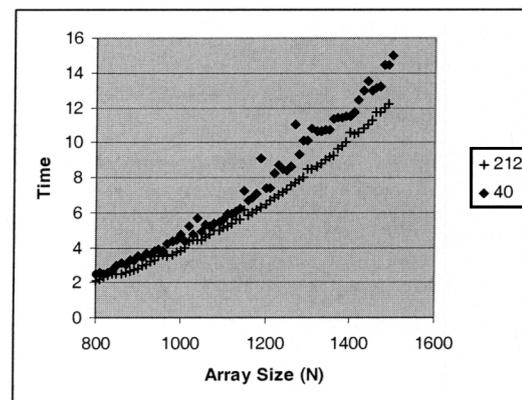
### 5.2.1 POWER 3

Figure 5 shows the performance of matrix multiply on the POWER 3 for various crossover points. The x-axis represents the array size (i.e. an array size of  $N$  is an  $N \times N$  matrix). The y-axis represents time. Ouroboros chose a crossover of 212. This yields very close performance to the *gemmw* hard-coded crossover of 192 for AIX systems. Some analytical models have suggested a crossover as low as 8. As can be seen in Figure 5, on the POWER 3, this results in a dramatic slowdown array size gets larger with a distinct jump when the array size is near 1000X1000.



**Figure 5: POWER 3 comparison of crossover**

In the next section we will see that the crossover selected for the Alpha Ev68 is 40. The Ouroboros crossover of 212 on the POWER 3 offers a modest speedup (from approximately 10% up to around 40%) over using the crossover of 40 as can be seen in Figure 6.



**Figure 6: Alpha Ev68 crossover on POWER 3**

Comparing Ouroboros with DGEMM on the POWER 3 we see that as the array size gets larger, Ouroboros begins to outperform the iterative BLAS 3 *gemm* library.

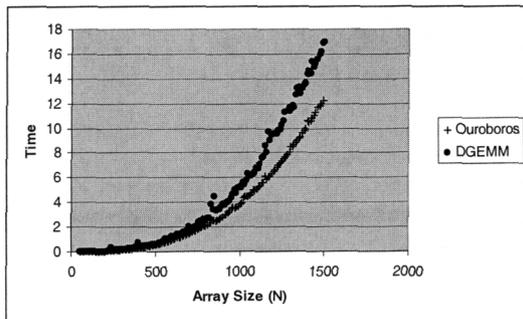


Figure 7: POWER 3 Ouroboros vs. DGEMM

### 5.2.2 Alpha Ev68

Figure 8 shows the performance of matrix multiply on the Alpha Ev68 for various crossover points. Ouroboros chose a crossover of 40. We can see that, as was the case for POWER 3, the Ouroboros-selected crossover corresponds very closely to the *gemmw* hard-coded crossover (32 for DEC Alpha). We also note that the crossover of 40 results in up to a 60% speedup over using the crossover of 212 that was chosen for POWER 3. This is a much more significant speedup than was seen using the the Alpha Ev68 crossover of 40 on the POWER 3. The Ouroboros crossover results in a two times speedup over the analytical lower-bound crossover of 8.

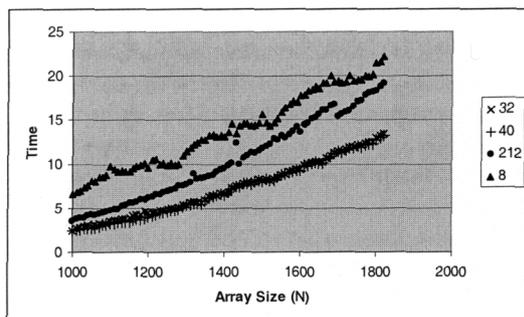


Figure 8: Alpha Ev68 crossover comparison

Figure 9 shows that speedup of the hybrid Ouroboros algorithm over the BLAS 3/*gemm* library. This is a more striking speedup in the array sizes tested. One would expect that the DGEMM curve would begin to level off as the array size gets larger.

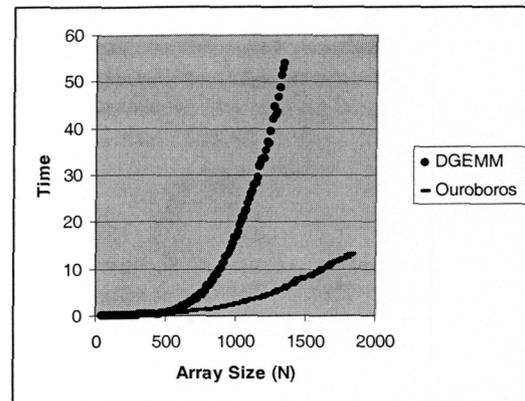


Figure 9: Alpha Ev68 Ouroboros vs. DGEMM

## 6 Conclusions and Future Work

We have presented that the use of a behavior-based model to construct hybrid algorithms automatically can produce high-performance results without requiring explicit knowledge of machine or algorithm structure. Our Ouroboros tool allows the algorithm designer to construct generic code that is then used by Ouroboros to instantiate a hybrid algorithm automatically using empirically derived performance data.

Since performance is machine dependent, it is a daunting task to create an algorithm that is sensitive to structural hardware nuances. Empirical approaches have been shown to be successful in addressing this challenge and there have been recent successes with empirically-tuned kernel libraries. However, for the algorithm designer, the challenge remains. The Ouroboros approach allows the designer the benefit of empirically-based hybrid algorithms without the need for specialized algorithm/architecture knowledge.

We are currently extending Ouroboros and its test suite to address Fourier Transform and All Pairs Shortest Path. We are also implementing a parallel version of Ouroboros capable of selecting from among parallel algorithm variants.

### References

1. G. Almasi, C. Cascaval, et al. "Demonstrating the Scalability of a Molecular Dynamics Application on a Petaflop Computer." *International Conference on Supercomputing*, Sorrento, 2001.
2. J. Bilmes, K. Asanovic, C-W. Chin, J. Demmel, "Optimizing Matrix

- Multiply using PHiPAC: a Portable, High-Performance, ANSI C Coding Methodology.”, *International Conference on Supercomputing*, Vienna, 1997.
3. J. Cao, D. J. Kerbyson, “Modelling of ASCI High Performance Applications Using PACE.” <http://www.dcs.warwick.ac.uk/~hpsg>.
  4. S. Chatterjee, A.R. Lebeck, et al., “Recursive Array Layouts and Fast Matrix Multiplication”, *IEEE Transactions on Parallel and Distributed Systems* 13(11): 1105-1123 (2002)
  5. S. Chatterjee, V.V. Jain, A. R. Lebeck, S., Mundhra, “Nonlinear Array Layouts for Hierarchical Memory Systems”, *International Conference on Supercomputing*, Rhodes, Greece, 1999.
  6. T. H. Cormen, C. E. Leiserson, et. al. *Introduction to Algorithms*. MIT Press. Boston. 2001.
  7. R.E. Crandall. *Projects in Scientific Computing*. Springer-Verlag. New York. 1994.
  8. A.C. Dusseau, D.E. Culler, et al. “Fast Parallel Sorting under LogP: Experience with the CM-5”, *IEEE Transactions on Parallel and Distributed Systems* 7(8): 791-805 (1996)
  9. I. Foster, C. Kesselman, eds., *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, San Francisco, 1998.
  10. I. Foster, C. Kesselman, Globus: “A Metacomputing Infrastructure Toolkit.” *International Journal of Supercomputer Applications*, vol. 11, no. 2, 1997.
  11. M. Frigo and S. G. Johnson, “FFTW: An Adaptive Software Architecture For The FFT.” *ICASSP*, vol. 3, 1998.
  12. M. Frigo and S. G. Johnson, “The Fastest Fourier Transform in the West.” *MIT-LCS-TR-728*, MIT, 1997.
  13. K.S. Gatlin, L. Carter, “Architecture-Cognizant Divide and Conquer Algorithms”, *Supercomputing*, Portland, 1999.
  14. K.S. Gatlin, L. Carter, “Faster FFTs via Architecture-Cognizance”, *International Conference on Parallel Architectures and Compilation Techniques*, Philadelphia, 2000.
  15. G. Golub, C. VanLoan, *Matrix Computations*, 3<sup>rd</sup> Ed. The Johns Hopkins University Press. Baltimore. 1996
  16. A. Hoisie, O. Lubeck, H. Wasserman “Performance and Scalability Analysis of Teraflop-Scale Parallel Architectures Using Multidimensional Wavefront Applications”, *The International Journal of High Performance Computing Application* 14(4): 330-347 (2002)
  17. J. Hong, H.T. Kung, “I/O Complexity: The Red-Blue Pebble Game”, *ACM Symposium on the Theory of Computing*, Milwaukee, 1981.
  18. E. Im, K. Yelick. “Optimization of Sparse Matrix Kernels for Data Mining”, <http://www.cs.berkeley.edu/~ejim/publication/icdm.ps>
  19. T. Katagiri, H. Kuroda, Y. Kanada, “A Method for Automatically Tuned Parallel Tridiagonalization on Distributed Memory Vector-Parallel Machines”, *Vector and Parallel Processing*, Porto, Portugal, 2000.
  20. K. Kennedy, M. Mazina, et. al., “Toward a Framework for Preparing and Executing Adaptive Grid Programs”, *International Parallel and Distributed Processing Symposium*, Fort Lauderdale, 2002
  21. A. R. Krommer, C.W. Ueberhuber, “Architecture Adaptive Algorithms”, *Parallel Computing*, 19, 1993.
  22. D. Mirkovic, S. L. Johnsson, “Automatic Performance Tuning in the UHFFT Library”, *International Conference on Parallel Computing*, 2001.
  23. A. Snively, N. Wolter, L. Carrington, “Modeling Application Performance by Convolution Machine Signatures with Application Profiles”,

*IEEE Workshop on Workload Characterization*, Austin, 2001.

24. S. S. Vadhiyar, G. E. Fagg, and J. Dongarra, "Automatically Tuned Collective Communications", *Supercomputing*, Dallas, 2000.

25. J. S. Vitter, "External Memory Algorithms and Data Structures", DIMACS Series in Discrete Mathematics and Theoretical Computer Science, 1991.

26. R. Vuduc, J.W. Demmel, "Code Generators for Automatic Tuning of Numerical Kernels: Experiences with FFTW", *Semantics Applications, and Implementation of Program Generation* Montreal, 2000.

27. R. Vuduc, J. W. Demmel, J. Bilmes, "Statistical Models for Automatic Performance Tuning", *International Conference on Computational Science* San Francisco, 2001.

28. R. C. Whaley, A. Petitet, J. J. Dongarra, "Automated Empirical Optimizations of Software and the ATLAS Project." *Parallel Computing*, 27(2), 2001.

29. J. Xiong, J. Johnson, et. al. "SPL: A Language and Compiler for DSP Algorithms", *Programming Language Design and Implementation*, Snowbird, 2001.