DOSA: Design Optimizer for Scientific Applications [*]

David A. Bader[1] and Viktor K. Prasanna[2]

[1]College of Computing
Georgia Institute of Technology
Altanta, GA 30332
bader@cc.gatech.edu

[2]Department of EE-Systems
University of Southern California
Los Angeles, CA 90089
prasanna@ganges.usc.edu

Abstract

In this paper we briefly introduce our new framework, called "Design Optimizer for Scientific Applications" (DOSA) which allows the programmer or compiler writer to explore alternative designs and optimize for speed (or power) at design-time and use a run-time optimizer. The run-time system is a portable interface that enables dynamic application optimization by interfacing with the output of DOSA. As an illustration we demonstrate speed up for two applications: Parallel Exact Inference and Community Identification in large-scale networks.

1. Introduction

High-performance computing (HPC) systems are taking a revolutionary step forward with complex architectural designs that require application programmers and compiler writers to perform the challenging task of optimizing the computation in order to achieve high performance. In the past decade, caches and high-speed networks have become ubiquitous features of HPC systems, canonically represented as clusters of multiprocessor workstations. To efficiently run application codes on these systems, the user must carefully lay out data and partition work so as to reduce communication, maintain a load balance, and expose locality for better cache performance. Realizing the gap between processor and memory performance, several HPC vendors, such as IBM and Cray, are incorporating into their next-generation systems innovative architectural features that alleviate this *memory wall*. These new architectural features include hardware accelerators (e.g., reconfigurable logic such as FPGAs, SIMD/vector processing units such as in the IBM Cell Broadband Engine processor, and graphics processing units (GPUs)), adaptable general-purpose processors, run-time performance advisors, capabilities for processing in the memory subsystem, and power optimizations. With these innovations, the multidimensional design space for optimizing applications is huge. Software must be sensitive to data layout, cache parameters, and data reuse, as well as dynamically changing resources, for highest performance.

Until recently, design-time analysis and optimizing compilers were sufficient to achieve high-performance on many HPC systems because they often appeared as static and homogeneous resources with a simple, well-understood model of execution at each processor. Today, techniques for load-balancing and job migration, readily-accessible grid computing, complex reconfigurable architectures, and adaptive processors, necessitate the requirement for run-time optimizations that depend upon the dynamic nature of the computation and resources. At run-time, an application may have different node architectures available to its running processes because it is executing in a distributed grid environment, and each component may require its own specific optimization to make use of the unique features available to it. HPC systems may have adaptable resources such as processors, and the run-time system gains new responsibility for requesting the appropriate configuration for the workload. Also, in a large, complex computing system, the available resources may change during iterations, and the run-time system must monitor, select, and tune new components to maintain or increase performance. Our goal is to design a dynamic application composition system that provides both high-performance computing and increased productivity. In this work, we discuss the Design Optimizer for Scientific Applications (DOSA), a semi-automatic framework for software optimization. DOSA will allow rapid, high-level performance estimation and detailed low-level simulation.

[*]This work is supported in part by the National Science Foundation, USA, under grant numbers CNS-0614915 and CNS-0613376.

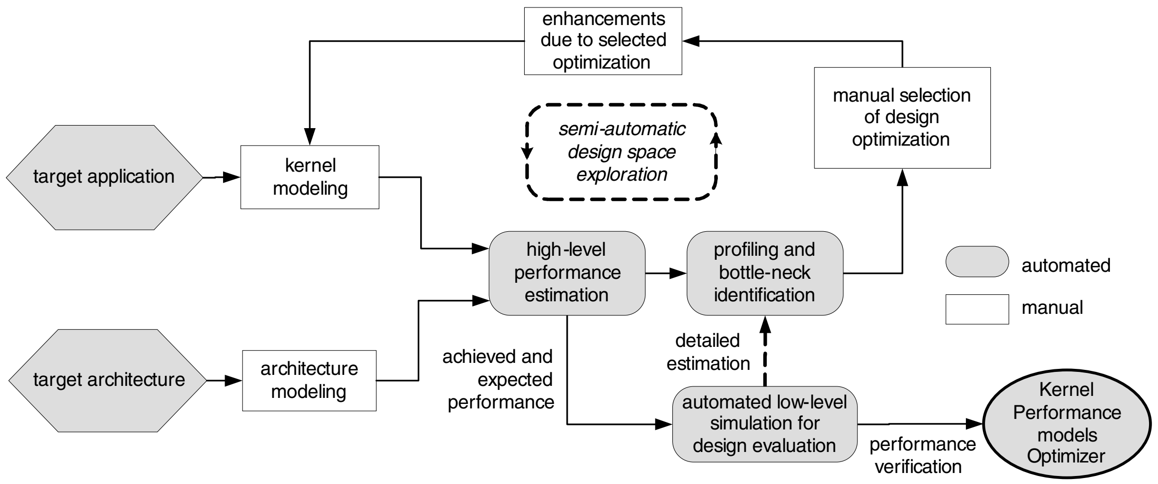
2. DOSA Framework

The design framework (DOSA) allows various optimizations to be explored based on the architectural features of the HPC platforms. In the DOSA framework, at design time, the designer models the kernels and architectures of her/his application. This modeling process includes performance estimation, simulation, identification of bottlenecks, and optimization. Through this modeling process, the designer identifies sets of components in the component library that can be utilized for the tasks in the kernels of her/his application. We will implement a high-level estimator that will estimate bandwidth utilization, memory activity, I/O complexity, and performance. Our approach is hierarchical: the framework will be used to perform a coarse exploration to identify potential optimizations, followed by detailed simulations to validate the performance improvements of the optimizations. The detailed simulations can also expose additional optimizations. This design-time exploration is manual. Thus our framework uses a semi-automatic design space exploration tool. The framework outputs a representation of the kernel, the performance models, and a run-time optimizer that can use these for efficient execution of the kernel.

The run-time system is a portable, high-level advisor that interfaces between the underlying execution system (operating system and architecture) and the application. During a kernel's execution, the execution system supplies performance information that is used by the run-time system to determine if the run-time optimizer should be called and to update the values of the parameters in the performance models. The run-time optimizer uses the updated performance models to select appropriate component(s) from the set determined during design-time optimization for executing the current task. Together, the run-time system and run-time optimizer make the run-time optimization decisions as a dynamic application composition system.

The design flow consists of two phases(refer to Figure 1): configuration and exploration. In the configuration phase, the framework is configured based on the target kernel and the architecture. The kernel developer (designer) initially defines a structural model of the target architecture. The structural model captures the details of the architecture. The designer uses appropriate micro-benchmarks and model and feedback interpreters to automatically perform cycle-accurate architecture simulation using integrated simulators (such as Mambo or SimpleScalar) to estimate the values of these parameters and update the model. We also plan to create a library of architecture models because it is likely that a designer would optimize several kernels for the same target architecture. If architecture models are available for the target architecture, then the designer can choose to skip the configuration phase and use the library instead. Along with the architecture, the designer also models the kernel. Kernel modeling involves describing the kernel as a task graph and specifying the components that implement these tasks. We will create a library of common components, such as matrix multiplication, FFT, and graph operators [1, 2]. If the desired component is not in the library, the designer can develop it from within the framework. We will write model interpreters that will generate executable specifications from a task graph. Once a kernel is modeled, the designer uses the integrated simulators to automatically generate a memory access trace for each code segment and associate it with the model.

In the exploration phase, the performance models (architecture and kernel models) defined in the configuration phase are used to perform design space exploration. Initially, the designer uses the high-level performance estimator to rapidly estimate the performance of the design and generate a profile of memory access. Based on the estimate and the profile, the designer identifies appropriate optimizations such as I/O complexity optimizations, data layout optimizations and data remapping optimizations, and modifies the models to include components supporting these types of optimizations. For example, in the case of in-memory processing for data remapping, the kernel model will be modified to indicate that a component for this type of processing is available and the estimator will account for concurrent in-memory processing during performance estimation and also suitably modify the memory access profile to reflect the remapping. The designer also can perform automatic low-level simulation to verify design decisions. Note that the low-level simulation is optional. By using the high-level models and the estimations, the design time can be reduced significantly. However, low-level simulations can provide additional insights with respect to optimizations to be considered. The designer continues to perform estimation, optimization, and low-level simulation until desired performance is reached or optimal processor-memory traffic is achieved. Similarly, for memory energy optimizations, the designer can identify data placement schemes (blocking, tiling, etc.) and memory/bank activation schedules and use the performance estimator to evaluate reduction in energy dissipation. Simplifying assumptions made to enable rapid estimation may induce errors in the estimates. Therefore, the DOSA framework will support specification of multiple candidate designs with estimated performance close to the desired performance. The framework will then output a representation of the kernel, the performance models, and a run-time optimizer that will use the representation and the performance models for efficient execution of the kernel.



enhancements due to selected optimization
manual selection of design optimization
semi-automatic design space exploration
target application
kernel modeling
high-level performance estimation
profiling and bottle-neck identification
automated
manual
target architecture
architecture modeling
achieved and expected performance
detailed estimation
automated low-level simulation for design evaluation
performance verification
Kernel Performance models Optimizer

Figure 1. Design flow using the DOSA framework

3 Illustrative Applications

3.1 Exact Inference

A. Inference in Bayesian Networks: A *Bayesian network* exploits conditional independence to represent a joint distribution more compactly. A Bayesian network is defined as $\mathrm{B} = (\mathbb{G}, \mathbb{P})$ where $\mathbb{G}$ is a *directed acyclic graph* (DAG) and $\mathbb{P}$ is the parameter of the network. The DAG $\mathbb{G}$ is denoted as $\mathbb{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V} = \{A_1, A_2, \ldots, A_n\}$ is the node set and $\mathcal{E}$ is the edge set. Each node A_i represents a random variable. If there is an edge from A_i to A_j i.e. $(A_i, A_j) \in \mathcal{E}$, A_i is called a *parent* of A_j. $pa(A_j)$ denotes the set of all parents of A_j. Given the value of $pa(A_j)$, A_j is conditionally independent of all other preceding variables. The parameter $\mathbb{P}$ represents a group of *conditional probability tables* (CPTs) which are defined as the conditional probability $P(A_j|pa(A_j))$ for each random variable A_j. Given the Bayesian network, a joint distribution $P(\mathcal{V})$ can be rewritten as $P(\mathcal{V}) = P(A_1, A_2, \cdots, A_n) = \prod_{j=1}^{n} Pr(A_j|pa(A_j))$.

The *evidence variables* in a Bayesian network are the variables that have been instantiated with values e.g. $\mathrm{E} = \{A_{e_1} = a_{e_1}, \cdots, A_{e_c} = a_{e_c}\}$, $e_k \in \{1, 2, \ldots, n\}$. Given the evidence, we can inquire the distribution of any other variables. The variables to be inquired are called *query variables*. The process of *exact inference* involves propagating the evidence throughout the network and then computing the updated probability of the query variables.

It is known that traditional exact inference using Bayes' rule fails for networks with undirected cycles[9]. Most inference methods for networks with undirected cycles convert a network to a cycle-free hypergraph called a *junction tree*. A junction tree is defined as $\mathrm{J} = (\mathbb{T}, \hat{\mathbb{P}})$ where $\mathbb{T}$ represents a tree and $\hat{\mathbb{P}}$ denotes the parameter of the tree. Each vertex $\mathcal{C}_i$ (known as a clique) of $\mathbb{T}$ is a set of random variables. Assuming $\mathcal{C}_i$ and $\mathcal{C}_j$ are adjacent, the *separator* between them is defined as $\mathcal{C}_i \cap \mathcal{C}_j$. All junction trees satisfy the *running intersection property* (RIP)[7]. $\hat{\mathbb{P}}$ is a group of *potential tables* (POTs). The POT of $\mathcal{C}_i$, denoted as $\psi(\mathcal{C}_i)$, can be viewed as the joint distribution of the random variables in $\mathcal{C}_i$. For a clique with w variables, each taking r different values, the number of entries in the POT is r^w.

In a junction tree, exact inference proceeds as follows: Assuming evidence is $\mathrm{E} = \{A_i = a\}$ and $A_i \in \mathcal{C}_j$, E is *absorbed* at $\mathcal{C}_j$ by instantiating the variable A_i, then renormalizing the remaining constituents of the clique. The effect of the updated $\psi(\mathcal{C}_j)$ is propagated to all other cliques by iteratively setting $\psi^*(\mathcal{C}_x) = \psi(\mathcal{C}_x)\psi^*(\mathcal{S})/\psi(\mathcal{S})$ where $\mathcal{C}_x$ is the clique to be updated; $\mathcal{S}$ is the separator between $\mathcal{C}_x$ and its neighbor that has been updated; ψ^* denotes the updated POT. After all cliques are updated, the distribution of a query variable $Q \in \mathcal{C}_y$ is obtained by $P(Q) = \sum_{\mathcal{R}} \psi(\mathcal{C}_y)/Z$ where $\mathcal{R} = \{z : z \in \mathcal{C}_y, z \neq Q\}$ and Z is a constant with respect to $\mathcal{C}_y$. This summation sums up all entries with respect to $Q = q$ for all possible q in $\psi(\mathcal{C}_y)$. The details of sequential inference are proposed by Lauritzen et al.[7]. Pennock[9] has proposed a parallel algorithm for exact inference on Bayesian network, which forms the basis of our work.

B. Parallelization: Given an arbitrary Bayesian network $\mathrm{B} = (\mathbb{G}, \mathbb{P})$, it can be converted into a junction tree $\mathrm{J} = (\mathbb{T}, \hat{\mathbb{P}})$ by five steps: moralization, triangulation, clique identification, junction tree construction and potential table construction. In this section, we discuss the parallel algorithms used by us in our implementation.

When the evidence variable is present at the root, the root absorbs the evidence by instantiating the evidence variable in its POT. Then, the pointer jumping technique is used to propagate the evidence throughout the complete tree. When the evidence variable is not present at the root, we extend the parallel tree rerooting technique [9] to make the clique that contains the evidence variable as the root of a new junction tree. Additional details can be found in our conferece paper [10].

Experiments: We ran our implementations on the DataStar Cluster at the San Diego Supercomputer Center (SDSC) [4] and on the clusters at the USC Center for High-Performance Computing and Communications (HPCC) [6]. Results can be found in Figure 2 and Figure 3. The DataStar Cluster at SDSC employs IBM P655 nodes running at 1.5 GHz with 2 GB of memory per processor.

The USC HPCC is a Sun Microsystems & Dell Linux cluster. A 2-Gigabit/second low-latency Myrinet network connects most of the nodes. The HPCC nodes used in our experiments were based on Dual Intel Xeon (64-bit) 3.2 GHz with 2 GB memory.

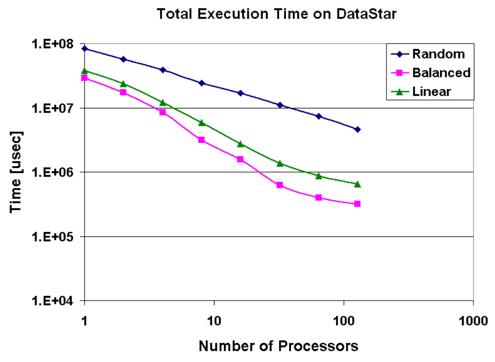


Figure 2. The execution time of exact inference on DataStar.

3.2 Large-scale Graph Analysis

Graph-theoretic applications have emerged as a prominent computational workload in the petascale computing era, and are representative of fundamental kernels in biology, scientific computing, and applications in national security. Real-world systems such as the Internet, socio-economic interactions, and biological networks typically exhibit common structural features a low graph diameter, skewed vertex degree distribution, self-similarity, and dense sub-graphs and are broadly referred to and modeled as small-world networks. Due to their large memory footprint, fine-grained computational granularity, and non-contiguous concurrent accesses to global data structures with low degrees of locality, massive graph problems pose serious challenges on current parallel machines. In recent work, we present new parallel algorithms and implementations that enable the design of several high-performance complex graph analysis kernels for small-world networks. In this report, we discuss our parallel approaches for the community identification problem [3].

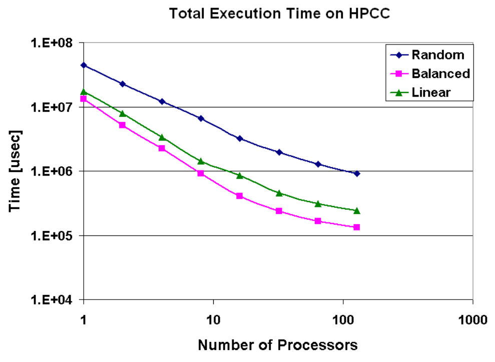


Figure 3. The execution time of exact inference on HPCC.

A. Community Identification in Small-world networks: A key problem in social network analysis is that of finding communities, dense components, or detecting other latent structure. This is usually formulated as a graph clustering problem, and several indices have been proposed for measuring the quality of clustering. Existing approaches based on the Kernighan-Lin algorithm, spectral algorithms, flow-based algorithms, and hierarchical clustering work well for specific classes of networks (e.g., abstractions from scientific computing, physical topologies), but perform poorly for small-world networks. Newman and Girvan recently proposed a divisive algorithm for small-world networks based on the edge betweenness centrality metric [8], and it has been applied successfully to a variety of real networks. However, it is compute-intensive and takes $O(n^3)$ time for sparse graphs (n denotes the number of vertices). We design three new parallel clustering approaches (two hierarchical agglomerative approaches: pMA and pLA, and one divisive clustering algorithm: pBD) [3] that exploit typical topological characteristics of small-world networks and optimize for a new clustering metric called modularity from the social network analysis.

B. Parallelization: Our parallel community identification approaches are primarily designed to exploit fine-grained thread level parallelism in graph traversal. We apply one of the following two paradigms in the design of parallel kernels: *level-synchronous* graph traversal, where vertices at each level are visited in parallel; or *path-limited searches*, in which multiple searches are concurrently executed and aggregated. The level-synchronous approach is particularly suitable for small-world networks due to their low graph diameter. Support for fine-grained efficient synchronization is critical in both these approaches. We try to aggressively reduce locking and barrier constructs through algorithmic changes, as well as implementation optimizations.

A second effective optimization we apply is to vary the granularity of parallelization in our clustering algorithms. In the initial iterations of the pBD algorithm, before the graph is split up into connected components of smaller sizes, we parallelize computation of approximate betweenness centrality. Once the graph is decomposed into a large number of isolated components, we can switch to computing exact centrality. We can then exploit parallelism at a coarser granularity, by computing centrality scores of each component in parallel.

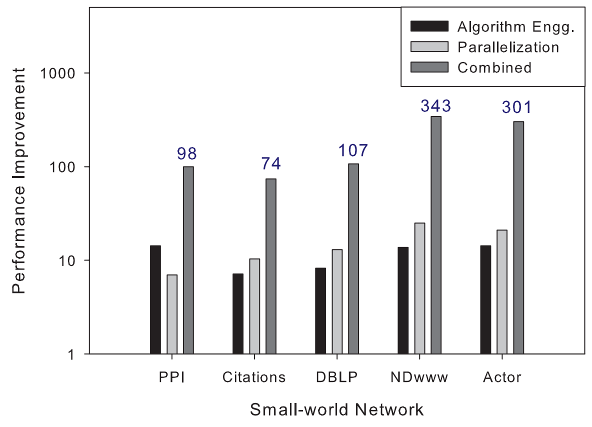


Figure 4. Speedup achieved by our parallel approach (pBD) over the Girvan-Newman (GN) algorithm. The bar labels indicate the ratio of execution time of GN to that of pBD.

C. Experiments: Our test platform for reporting parallel performance results in this report is the Sun Fire T2000 server, with the Sun UltraSPARC T1 (Niagara) processor. This system has eight cores running at 1.0 GHz, each of which is four-way multithreaded. The cores share a 3 MB L2 cache, and the system has a main memory of 16 GB. We compile our code with the Sun C compiler v5.8 and the default optimization flags.

We demonstrate that our parallel schemes give significant running time improvements over existing modularity-based clustering heuristics for a collection of small-world networks gathered from diverse application domains: a protein-interaction network from computational biology (`PPI`), a citation network (`Citations`), a web crawl (`NDwww`), and two social networks (`Actor`, `DBLP`). For instance, our novel divisive clustering approach based on approximate edge betweenness centrality (pBD) is *more than two orders of magnitude* faster than the Girvan-Nirvan (GN) algorithm on the Sun Fire T2000 multicore system, while maintaining comparable clustering quality (see Figure 4).

4. Concluding Remarks

The goal of our research is to develop a framework that supports design-time optimizations of applications in high-performance computing and whose output can then be used to perform further optimizations at run-time. To illustrate our ideas, we give two example kernels, one for parallel exact inference in Bayesian networks and the other for community identification in small-world networks. In each case, we briefly describe the design-time modeling of the algorithms and give several performance optimizations enabled by the DOSA framework.

References

[1] D. Bader and G. Cong. Fast shared-memory algorithms for graph theoretic problems. *Journal of Parallel and Distributed Computing*, 66(11):1366–1378, 2006.

[2] D. Bader and G. Cong. Efficient parallel graph algorithms for shared-memory multiprocessors. In S. Rajasekaran and J. Reif, editors, *Handbook of Parallel Computing: Models, Algorithms, and Applications*. CRC Press, 2007.

[3] D. Bader and K. Madduri. SNAP, small-world network analysis and partitioning: an open-source parallel graph framework for the exploration of large-scale networks. In *Proc. 22nd Int'l Parallel and Distributed Processing Symp. (IPDPS)*, Miami, FL, Apr. 2008.

[4] Datastar. `http://www.sdsc.edu/us/resources/datastar/`.

[5] Generic modeling environment. `http://www.isis.vanderbilt.edu/Projects/gme/`.

[6] HPCC. `http://www.usc.edu/hpcc/`.

[7] S. Lauritzen and D. Spiegelhalter. Local computation with probabilities and graphical structures and their application to expert systems. *J. Royal Statistical Society B.*, 50(6):157–224, 1988.

[8] M. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113, 2004.

[9] D. Pennock. Logarithmic time parallel Bayesian inference. In *Proc. of the 14th Ann. Conf. on Uncertainty in Artificial Intelligence*, pages 431–438, Oct. 1998.

[10] Y. Xia and V. Prasanna. Parallel exact inference. In *Proc. of the 11th Int'l Conf. on Parallel Computing (ParCo '07)*, Sept. 2007.