Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors

David A. Bader* and Kamesh Madduri

College of Computing,
Georgia Institute of Technology, Atlanta, GA 30332, USA
{bader, kamesh}@cc.gatech.edu

Abstract. Graph theoretic problems are representative of fundamental computations in traditional and emerging scientific disciplines like scientific computing and computational biology, as well as applications in national security. We present our design and implementation of a graph theory application that supports the kernels from the Scalable Synthetic Compact Applications (SSCA) benchmark suite, developed under the DARPA High Productivity Computing Systems (HPCS) program. This synthetic benchmark consists of four kernels that require irregular access to a large, directed, weighted multi-graph. We have developed a parallel implementation of this benchmark in C using the POSIX thread library for commodity symmetric multiprocessors (SMPs). In this paper, we primarily discuss the data layout choices and algorithmic design issues for each kernel, and also present execution time and benchmark validation results.

1 Introduction

One of the main objectives of the DARPA High Productivity Computing Systems (HPCS) program [1] is to reassess the way we define and measure performance, programmability, portability, robustness and ultimately *productivity* in the High Performance Computing (HPC) domain. An initiative in this direction is the formulation of the Scalable Synthetic Compact Applications (SSCA) [2] benchmark suite. These synthetic benchmarks are envisioned to emerge as complements to current scalable micro-benchmarks and complex real applications to measure high-end productivity and system performance. Each SSCA benchmark is composed of multiple related kernels which are chosen to represent workloads within real HPC applications and is used to evaluate and analyze the ease of use of the system, memory access patterns, communication and I/O characteristics. The benchmarks are relatively small to permit productivity testing and

* This work was supported in part by DARPA Contract NBCH30390004; and NSF Grants CAREER ACI-00-93039, NSF DBI-0420513, ITR ACI-00-81404, DEB-99-10123, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654.

D.A. Bader et al. (Eds.): HiPC 2005, LNCS 3769, pp. 465–476, 2005.

programming in reasonable time; and scalable in problem representation and size to allow simulating a run at small scale or executing on a large system at large scale. They are also described in sufficient detail to drive novel HPC programming paradigms, as well as architecture development and testing.

SSCA#2 [3] is a graph theoretic problem which is representative of computations in the fields of national security, scientific computing, and computational biology. The HPC community currently relies excessively on single-parameter microbenchmarks like LINPACK [4], which look solely at the floating-point performance of the system, given a problem with high degrees of spatial and temporal locality. Graph theoretic problems tend to exhibit irregular memory accesses, which leads to difficulty in partitioning data to processors and in poor cache performance. The growing gap in performance between processor and memory speeds, the memory wall, makes it challenging for the application programmer to attain high performance on these codes. The onus is now on the programmer and the system architect to come up with innovative designs.

Symmetric Multiprocessors (SMPs) with modest shared memory have emerged as a popular platform for the design of scientific and engineering applications. SMP clusters are now ubiquitous in high-performance computing, consisting of clusters of multiprocessors nodes (e.g., IBM pSeries, Sun Fire, Compaq AlphaServer, and SGI Altix) inter-connected with high-speed networks (e.g., vendor-supplied, or third party such as Myricom, Quadrics, and InfiniBand). Current research has shown that it is possible to design algorithms for irregular and discrete computations [5,6,7] that provide efficient and scalable performance on SMPs. To analyze SMP performance, we use a complexity model similar to that of Helman and JáJá [8] which has been shown to provide a good cost model for shared memory algorithms on current symmetric multiprocessors [5,8,9]. The model uses two parameters: the problems input size n, and the number p of processors. There are two parts to an algorithm's complexity in this model: M_E, the maximum number of non-contiguous memory accesses required by any processor, and T_C, the computation complexity. This model, unlike the idealistic PRAM, is more realistic in that it penalizes algorithms with non-contiguous memory accesses that often result in cache misses.

This paper is organized as follows. Sections 3-7 discuss the scalable data generation stage and each of the four kernels in detail: we present the kernel specification, the design trade-offs involved in implementation, illustrations of our data layouts, and relevant algorithms. Section 8 summarizes the execution time and memory usage results, primarily on the Sun E4500 shared memory SMP. In the final section, we present our conclusions and plans for future work.

2 Preliminaries

2.1 Definitions

Let $G = (V, E)$ be a directed, weighted multi-graph, where $V = \{v_1, v_2, ..., v_n\}$ is the set of vertices, and $E = \{e_1, e_2, ..., e_m\}$ is the set of weighted, directed edges. An edge $e_i \in E$ is represented by the tuple $\langle u, v, w_i \rangle$, where $u, v \in V$,

w_i is either a positive integer from a bounded universe or a character string of fixed length, and the edge e_i is directed from u to v. There are no self loops in the SSCA#2 graph, i.e., for any edge $e_i = \langle u, v, w_i \rangle \in E$, we have $u \neq v$. Two vertices u, v are said to be *linked* if there exists at least one directed edge from u to v or v to u. We define a set of vertices $C \subseteq V$ to be a *clique*, if each pair of vertices $\{u, v\} \in C$ is *linked*. This means that a clique has edges between each pair of vertices, but not necessarily in both directions. A *cluster* $S \subseteq C \subseteq V$ is loosely described as a maximal set of *highly inter-connected* vertices.

2.2 Benchmark Input Parameters

Some user-defined constants are used for the data generation step and subsequent kernels.

1. *totVertices* : the number of vertices in the graph. We also use n to represent the number of vertices, and m the number of directed edges in sections of the paper.
2. *maxCliqueSize* : the maximum size of a clique in the graph. Clique sizes are uniformly distributed in the interval [1, *maxCliqueSize*].
3. *maxParalEdges* : the maximum number of parallel edges between two vertices. The number of edges between any two vertices are uniformly distributed in the interval [1, *maxParalEdges*]
4. *probUnidirectional* : probability that the connections between two vertices will be unidirectional as opposed to bidirectional
5. *probInterClEdges* : the probability of inter-clique edges
6. *percIntWeights* : percentage of edges assigned integer weights
7. *maxIntWeight* : the maximum integer weight
8. *maxStrLen* : maximum number of characters in the string weight
9. *subGrEdgeLength* : maximum edge length in graphs generated by Kernel 3
10. *maxClusterSize* : maximum cluster size generated by the cuts in Kernel 4

3 Scalable Data Generation

The Scalable Data Generation stage takes user parameters as input and generates the graph as tuples of vertex pairs and their corresponding weights. The intended graph has a hierarchical nature, with random-sized *cliques*, and inter-clique edges assigned using a random distribution. The edge weights can be integer values or randomly generated character strings. The scalable data generator need not be parallelized, and is not timed.

3.1 Implementation

This step's output should be an edge list with each element of the form $\langle u, v, w \rangle$, where the edge is directed from u to v, and w is a positive integer weight or a character string. Our implementation returns four one-dimensional array

constructs: two arrays corresponding to the start and end vertices, and the two other arrays representing the integer and string weights. Although this stage is not timed, we parallelize the main steps for practical considerations.

Note that the SSCA#2 graph has some very specific properties. It is essentially a collection of cliques (defined in the earlier section), with the *inter-clique edges* assigned using a hierarchical distribution, based on the distance between the cliques. The fourth kernel deals with extraction of highly inter-connected *clusters* from the graph, and we would like the extracted clusters to be as close as possible to the original cliques. The implementation details of the data generation stage are discussed in an extended version of this paper [10].

4 Kernel 1: Graph Generation

This kernel constructs the graph from the data generator output tuple list. The graph can be represented in any manner, but cannot be modified by subsequent kernels. The number of vertices in the graph is not provided and needs to be determined in this kernel. It is also suggested that statistics be collected on the graph to aid verification of subsequent kernels.

4.1 Details

There are many figures of merit for each kernel, including but not limited to memory use, running time, ease of programming, ease of incrementally improving, and so forth. Thus, a figure of merit for any implementation would be the total space usage of the graph data structure. Also, the graph data structure (or parts of it) *cannot be modified or deleted* by subsequent kernels. So we need to choose a data layout which can be created quickly and easily (since Kernel 1 is timed), is space efficient, and is optimized for efficient implementations of Kernels 2, 3 and 4.

Kernels 2 and 3 operate on the directed graph, but for Kernel 4, the specification states that multiple edges, edge directions, and edges weights, are to be ignored. This complicates the design and implementation – if we plan to use a separate graph layout for Kernel 4, we need to construct it in Kernel 1, and it cannot be modified in Kernels 2 and 3. The developer now must design a data structure and layout which considers all these competing optimization criteria, and this is the core challenge in the benchmark.

An adjacency matrix representation is easy to implement and well-suited for dense graphs. In this case, however, the generated graph is sparse and a matrix representation would be very inefficient in memory usage. Another common method of representing directed and weighted graphs is the adjacency list representation. This is easy to implement and also space efficient. However, repeated memory allocation calls while constructing large graphs, and irregular memory accesses in the subsequent kernels will hurt performance. For our current implementation, we follow an adjacency list representation, but using the more cache-friendly *adjacency arrays* [11] with auxiliary arrays.

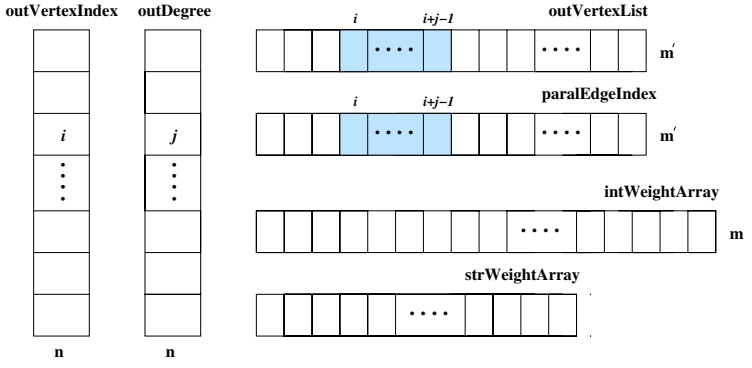

Fig. 1. The data layout for representing the directed graph – Kernel 1

Since multiple edges between two vertices can be ignored for Kernels 3 and 4, we do not store them explicitly, but have another array to keep track of these edges and to map an edge to its corresponding weight. We first construct the part of the data structure to store the directed graph information. We use two arrays of size *totVertices* to index and access the adjacencies corresponding to each vertex. The adjacency list (without multiple edges) is stored in a contiguous memory location, and so is the array storing the multiple edge information. The data layout used is illustrated in Fig. 1.

Graph construction (for our adjacency array representation) is inherently sequential, but since we have a sorted edge tuple list, we can extract some parallelism. First, the size of the graph can be easily determined by finding the maximum vertex number in the start vertex or the end vertex list. Assuming the tuple list is sorted by start vertex, the value can be determined in constant time by reading off the last element in the startVertex array. Otherwise we can determine the maximum value in parallel in $T_C = O(m/p + \log p)$ time. Processors then scan independent sections of the tuple list to determine the out-degree of each vertex. We have a parallel time overhead of $O(p)$ for bookkeeping purposes. In the next pass, we allocate memory for the *outVertexList* and *paralEdgeList* arrays and fill in entries in parallel in $O(m'/p + \log p)$ time, where m' is the number of unique directed edges (removing the parallel edges).

We construct the implied edge list by scanning the *outVertexList* in parallel. For each edge $\langle u, v \rangle$, we check if the *outVertexList* has the edge $\langle v, u \rangle$. If not, we add u to the implied edge list of v. This step has an asymptotic time complexity of $T_C = O(m'/p + \log p)$ and involves $m' + m/p$ non-contiguous memory accesses. We also need to use mutex locks to prevent race conditions, which affects performance. The integer and string weight arrays can be trivially constructed in constant time, since we retain the vertex ordering in the edge tuples. In sum, the computational complexity for Kernel 1 is given by $T_C = O(m/p + \log p)$, and $M_E = m' + 2m/p$. The asymptotic space requirements for the storing the tuple list and the graph data structure are both $O(m)$. The memory requirements in both these cases are further compared in Section 9.

5 Kernel 2: Classify Large Sets

The intent of this kernel is to determine vertex pairs with the largest integer weight and the specified string weight. Two vertex pair lists, S_I and S_C, are generated in this step and serve as start sets for graph extraction in Kernel 3. This kernel is timed.

To determine S_I, we first scan the integer weight list in parallel, determine local maxima, and store the corresponding end vertex. Then, we do an efficient reduction operation on the p values to determine the maximum weight in $O(\log p)$ time. The corresponding start vertices for the elements in S_I can be determined by a fast binary search in parallel on the *outVertexIndex* array. The set S_C can be similarly determined. As we have stored the edge weights in a contiguous block, we have the work equally distributed among all processors. Finding the maximum weighted edge is the dominant step in this stage and $T_C = O(m/p + \log p)$ for this kernel.

6 Kernel 3: Extracting Sub-graphs

Starting from each of the vertex pairs in the sets S_I and S_C, this kernel produces sub-graphs which consist of the vertices and edges along all paths of length less than *subGrEdgeLength*. The recommended algorithm for graph extraction in the specification is Breadth First Search.

6.1 Implementation

We use a Breadth First Search (BFS) algorithm starting from the *endVertex* of each element in S_I and S_C, up to a depth of *subGrEdgeLength*. Now *subGrEdgeLength* is typically chosen to be a small number, a constant value in comparison to the number of graph vertices. We also know that this graph is essentially a collection of cliques (whose maximum size is bounded), and so a BFS up to a constant depth would yield a subgraph $G' = (V', E')$ such that $|V'| \ll |V|$. Even though the BFS computational complexity is of the same order as the previous kernels ($T_C = O(m')$), we can expect this kernel to finish much faster. We have not implemented a fine-grained parallel BFS yet. Currently, we just distribute the vertices in S_I to the available processors and run BFS in parallel on each of these, which limits the concurrency to $|S_I| + |S_C|$. The queue ADT we use in this algorithm is implemented using a dynamic array, a linked list and a simple one-dimensional array. Since the extracted graph is quite small, we find that all three representations give similar results. Note that linked lists are easy to implement, space-efficient and could be used for small problem sizes, since we will not be performing any further operations with the extracted graph.

7 Kernel 4: Graph Clustering

The intent of this kernel is to partition the graph into highly inter-connected *clusters* and minimize the number of links between these clusters. Multiple edges,

edge directions and weights can be ignored. Since exact solutions to this problem are NP-hard, heuristics are allowed, provided they satisfy the kernel validation criterion. This kernel should not utilize any auxiliary information collected in the previous kernels or in the graph generation process.

7.1 Details

This kernel is based on the partitioning problem formulated by Kernighan and Lin [12], with all the edge costs considered equal. Sangiovanni-Vincentelli, Chert, and Chua [13,14] have earlier applied this work for solving circuit problems. The maximal clique problem [15] is a well-studied NP-complete problem, and several heuristics have been proposed to solve this [16]. Our problem is not as difficult as the maximal clique problem, because of the manner in which the graph is generated, and also due to the restriction on the maximum clique size.

We cannot apply popular multi-level graph partitioning tools like Chaco [17] and METIS [18] to solve this kernel. These tools use a variety of heuristics and are highly refined, but they are primarily used to partition nearly-regular graphs into *equal-sized blocks*, while minimizing edge cut. Graph partitioning results using Chaco are presented in [10]. The required partitioning in this problem, however, is highly irregular and cannot be found accurately using these tools.

The specification suggests an algorithm for solving this kernel, which is a variant of a graph clustering algorithm given by Koester [19]. This sequential algorithm iteratively forms a sequence of disjoint clusters, which are subgraphs no larger than *maxClusterSize* vertices. As each cluster is selected, its vertices are removed from further consideration. To select the vertices in a cluster, the algorithm starts with some remaining vertex (which forms the initial one-element cluster), and its links to any remaining vertices (which form the initial adjacent set). It then expands the cluster by repeatedly moving an adjacent set vertex to the cluster, and adding that vertex's non-cluster links to the adjacent set. The new vertex is chosen depending on how tightly it and its links are connected to the existing cluster, and how many links it adds to the adjacent set. The cluster is complete if the adjacent set is empty. Otherwise when the cluster reaches *maxClusterSize* vertices in size, the cluster elements are marked used, the cluster is added to the cluster list, and size of the adjacent set is added to the count of interclique links.

The reference implementation uses this algorithm for solving Kernel 4 and reports good results. The specification suggests statistical validation for assessing the quality of the clustering algorithm. One recommended empirical measure is to check if $interClusterLinkNum < refcutLinksNum$, where *refcutLinksNum* is given by $\frac{interCliqueLinkNum}{\sqrt{(maxClusterSize/maxCliqueSize)}}$ and *interCliqueLinkNum* refers to the number of inter-clique vertex pairs connected by at least one directed edge. Algorithms with *interClusterLinkNum* within 5% of the value *refCutLinksNum* are acceptable. It is also suggested that for small problem sizes, the algorithm correctness be checked rigorously, and parallel results be verified against serial results.

This algorithm is however inherently sequential. Cliques of size less than *maxClusterSize* with inter-clique edges may not be extracted correctly. We propose a new parallel greedy algorithm (pseudo-code is given in [10]) to extract clusters. The quality of results is comparable to the reference algorithm, and some results are presented in the next section.

Our parallel algorithm works as follows. We first sort the vertices in parallel in the decreasing order of their degree. The parallel radix sort uses a linear-time counting sort for a constant number of iterations. A shared array *vStatus* of size n is maintained to keep track of the status of each vertex – whether it is unassigned yet, or assigned to a unique cluster. Each processor chooses a vertex from the top of the queue, colors the vertex and its adjacencies (both the out-vertices and the implied edges) with a unique number, given by $i \times$ *current iteration number*, where i is the processor index. The adjacencies of each vertex in the cluster are inspected, and if more than a certain threshold of them are similarly colored, it is accepted. Otherwise it is rejected and the vertex is unmarked. We also update the *edgeCut* simultaneously — if we decide that an originally colored vertex does not belong to the cluster, we add all the inter-clique edges to the cut-set. The vertex degree is bounded by $O(maxClusterSize)$. The clustering algorithm runs in linear time in the worst case (a single clique of size $O(n)$), with M_E given by $O(n/p)$. If *maxClusterSize* is chosen to be a constant value, $T_C = M_E = O(n/p)$.

The heuristic correctly extracts nearly all cliques, except for those of very small sizes (with 3-4 elements), as it is tough to define acceptance thresholds. We have two choices in such cases: either classify these vertices as clusters of smaller sizes (say 1 or 2), or add these vertices to existing clusters. The former approach is a more conservative method of forming clusters and *false positives* (vertices wrongly assigned to a cluster) are avoided, but it would also lead to an inflated number of extracted clusters and inter-cluster edges. We thus have a trade-off between graph clustering *specificity* (corresponds to exact clique extraction) and *sensitivity* (correlates to minimization of intra-cluster links) in this case. We can define the threshold values for accepting a vertex into a cluster according to what our primary optimization criterion is — retaining specificity, or minimizing inter-clique edges and increasing sensitivity. The suggested validation scheme for this kernel is to compare the inter-clique links with the inter-cluster links, and so we optimize for the inter-cluster edges when reporting the results in Section 9.

8 Experimental Results

This section summarizes the experimental results of our SSCA#2 implementation, tested on the Sun E4500, a uniform-memory-access (UMA) shared memory parallel machine with 14 UltraSPARC II 400MHz processors and 14 GB of memory. Each processor has 16 Kbytes of direct-mapped data (L1) cache and 4 Mbytes of external (L2) cache.

We use a binary scaling heuristic *SCALE* to uniformly express the input parameter values. The following values have been used for reporting results in this section: $totVertices = 2^{SCALE}$, $maxCliqueSize = 2^{(SCALE/3)}$, *maxParalEdges*

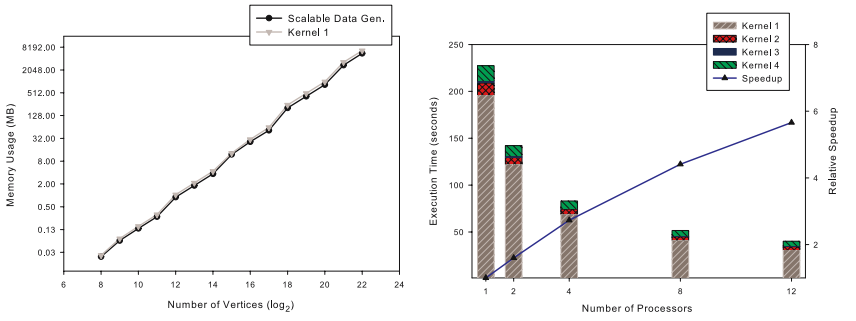

Fig. 2. Memory Usage (left) and Execution Time (right)

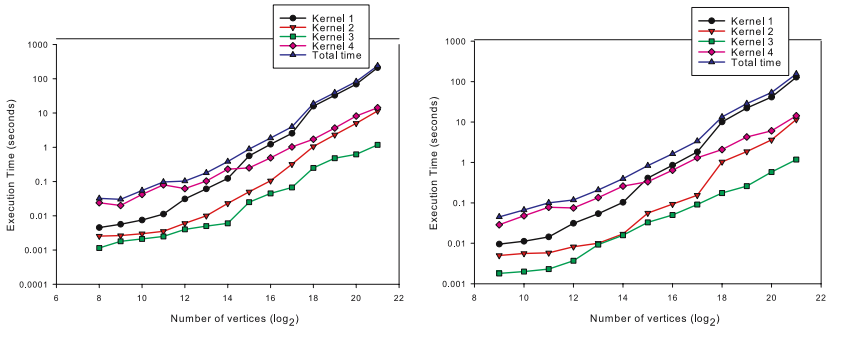

Fig. 3. Execution time of Kernels 1, 2, 3, and 4, on four and eight processors, in the left and right plots, respectively

$= 3$, $probUnidirectional = 0.3$, $probInterClEdges = 0.5$, $percIntWeights = 70$, $maxIntWeight = 2^{SCALE}$, $maxStrLen = SCALE$, $subGrEdgeLength = SCALE$, and $maxClusterSize = 2^{(SCALE/3)}$.

Fig. 2 compares memory utilization of the data generator and our graph layout (described in Section 5). Note that we explicitly store implied edge information in Kernel 1, causing the graph data structure to use slightly more memory than the data generator output. One of the figures of merit of the implementation is the largest problem size that can be solved on a given architecture. On the Sun E4500, memory proves to be the bottleneck to scaling. The largest problem size that can be handled with these parameters is 2^{21} vertices, which generates 156M edges for the above input parameters. We could further solve a problem size of 2^{22} vertices, by writing the data generator output to disk.

The running times for multi-processor runs are also given in Fig. 2. The execution time is dominated by graph generation, which scales reasonably with the number of processors for various problem sizes. We use a locking scheme to construct the implied edge list in parallel, which leads to a moderate slowdown of Kernel 1. There is also limited parallelism in Kernel 3 dependent on the size of the Kernel 2 start sets.

Table 1. Kernel 4 – Graph Clustering Results. (intra and inter-clique edges include parallel edges; a link is defined as a vertex pair connected by at least one directed edge).

SCALE	12	16	20
No. of Vertices	4096	65536	1048576
No. of intra-clique edges	40850	361114	39511513
No. of inter-clique edges	8472	72365	645787
No. of cliques	486	3990	32167
Avg. clique size	8.42	16.42	32.6
No. of extracted clusters	383	3142	25201
Avg. cluster size	10.69	20.85	41.6
No. of inter-clique links	5230	49907	422292
No. of inter-cluster links	1968	18892	185250

Fig. 3 gives the running times of the four kernels for various problem scales, on four and eight processors respectively. Note that the number of non-contiguous memory accesses $M_E = O(m')$ and $T_C = O(n/p + \log p)$ for Kernel 1, and so the benchmark execution time is dominanted by graph construction. Since $maxClusterSize = 2^{SCALE/3}$, we find a sharp rise in Kernel 1 execution time for $SCALE = 9$, 12, 15, and 18, as the number of edges generated in these cases is comparatively higher than the previous value. The dominant step in Kernel 1 is construction of the implied edge list. Kernel 3 takes the least time, as the search depth value is very small.

Rigorous verification of full-scale runs is prohibitive, and so the benchmark specification suggests a statistical validation scheme. Table 1 summarizes validation results for Kernel 4. The number of clusters extracted and the number of inter-cluster links are reported for three different problem sizes (for a four-processor run). The quality of the results is chiefly dependent on two input parameters: *probUnidirectional* and *probInterClEdges*. We have tested the correctness of our implementation on small graph sizes. We also find the clustering results to be consistent across multi-processor runs, as we do not use locking in this kernel. Note that in cases when the graph has a high percentage of inter-clique edges, we have a trade-off between exact clique extraction and minimization of inter-cluster edges, as discussed in the previous section.

9 Conclusions

In this paper, we present the design and implementation of the SSCA#2 graph theory benchmark. This benchmark consists of four kernels with irregular memory access patterns that chiefly test a system's memory bandwidth and latency. Our parallel implementation uses C and POSIX threads and has been tested on the Sun Enterprise E4500 SMP system. The dominant step in the benchmark is the construction of the graph data structure, which limits scaling on the

Sun E4500. We are currently working on implementations of SSCA#2 on other shared-memory systems such as the Cray MTA-2 and the Cray XD1.

Acknowledgments

We thank Bill Mann, Jeremy Kepner, John Feo, David Koester, John Gilbert, Ram Rajamony, and other members of the HPCS working group for trying out early versions of our implementation, discussions of the benchmark specifications, and their valuable suggestions.

References

1. DARPA Information Processing Technology Office: High productivity computing systems project (2004) `http://www.darpa.mil/ipto/programs/hpcs/`.
2. Kepner, J., Koester, D.P., *et al.*: HPCS Scalable Synthetic Compact Application (SSCA) Benchmarks (2004) `http://www.highproductivity.org/SSCABmks.htm`.
3. Kepner, J., Koester, D.P., *et al.*: HPCS SSCA#2 Graph Analysis Benchmark Specifications v1.0. (2005)
4. Dongarra, J., Bunch, J., Moler, C., Stewart, G.: LINPACK Users' Guide. SIAM, Philadelphia, PA. (1979)
5. Bader, D., Sreshta, S., Weisse-Bernstein, N.: Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In Sahni, S., Prasanna, V., Shukla, U., eds.: Proc. 9th Int'l Conf. on High Performance Computing (HiPC 2002). Volume 2552 of Lecture Notes in Computer Science., Bangalore, India, Springer-Verlag (2002) 63–75
6. Bader, D.A., Cong, G.: A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In: Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004), Santa Fe, NM (2004)
7. Bader, D.A., Cong, G.: Fast shared-memory algorithms for computing the minimum spanning forest of sparse graphs. In: Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004), Santa Fe, NM (2004)
8. Helman, D.R., JáJá, J.: Designing practical efficient algorithms for symmetric multiprocessors. In: Algorithm Engineering and Experimentation (ALENEX'99). Volume 1619 of Lecture Notes in Computer Science., Baltimore, MD, Springer-Verlag (1999) 37–56
9. Helman, D.R., JáJá, J.: Prefix computations on symmetric multiprocessors. Journal of Parallel and Distributed Computing **61** (2001) 265–278
10. Bader, D.A., Madduri, K.: Design and implementation of the HPCS graph analysis benchmark on symmetric multiprocessors. Technical report, Georgia Instutite of Technology (2005)
11. Park, J., Penner, M., Prasanna, V.: Optimizing graph algorithms for improved cache performance. In: Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2002), Fort Lauderdale, FL (2002)
12. Kernighan, B., Lin, S.: An efficient heuristic procedure for partitioning graphs. The Bell System Technical Journal **49** (1970) 291–307
13. Sangiovanni-Vincentelli, A., Chert, L., Chua, L.: A new tearing approach: Node tearing nodal analysis. In: Proc. IEEE Int'l Symp. on Circ. and Syst., Phoenix, AZ (1975) 143–147

14. Sangiovanni-Vincentelli, A., Chert, L., Chua, L.: An efficient heuristic cluster algorithm for tearing large-scale networks. IEEE Trans. Circuits and Systems (1977) 709–717
15. Bomze, I., Budinich, M., Pardalos, P., Pelillo, M.: The maximum clique problem. In Du, D.Z., Pardalos, P.M., eds.: Handbook of Combinatorial Optimization. Volume 4. Kluwer Academic Publishers, Boston, MA (1999)
16. Johnson, D., Trick, M., eds.: Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, October 11-13, 1993. Volume 26 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science. American Mathematical Society (1996)
17. Hendrickson, B., Leland, R.: A multilevel algorithm for partitioning graphs. In: Proc. Supercomputing '95, San Diego, CA (1995)
18. Karypis, G., Kumar, V.: MeTiS: A Software Package for Partitioning Unstructured Graphs, Partitioning Meshes, and Computing Fill-Reducing Orderings of Sparse Matrices. Department of Computer Science, University of Minnesota. Version 4.0 edn. (1998)
19. Koester, D.P.: Parallel Block-Diagonal-Bordered Sparse Linear Solvers for Power Systems Applications. PhD thesis, Syracuse University, Syracuse, NY (1995)