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

David A. Bader[*] Kamesh Madduri
College of Computing
Georgia Institute of Technology, Atlanta, GA 30313
`{bader,kamesh}@cc.gatech.edu`

September 12, 2005

Abstract

Graph theoretic problems are representative of fundamental computations in traditional and emerging scientific disciplines like scientific computing, computational biology and bioinformatics, 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 portable parallel implementation of this benchmark in C using POSIX thread libraries 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 [9] 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) [21] benchmark suite. These synthetic benchmarks are envisioned to emerge as alternatives to current scalable micro-benchmarks and complex real

[*]This work was supported in part by 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; and DARPA Contract NBCH30390004.

applications to measure high-end productivity and system performance. Each SSCA benchmark is composed of multiple related kernels which are chosen to be represent workloads within real HPC applications, to test for instance, ease of use of the system, memory access patterns, communication and I/O characteristics. The benchmarks are not chosen be numerically rigorous and are small enough to permit productivity testing. They can be programmed in reasonable time and can scale down to run on a simulator, or scale up to run on a large system. They are also described in sufficient detail to drive novel HPC programming paradigms, as well as architecture development and testing.

SSCA#2 [12] 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 simple benchmarks like LinPack [16], which looks 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 locality. 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.

SSCA#2 consists of four timed kernels which involve operations on a directed multi-graph with weighted edges. In addition, there is an untimed preliminary phase termed the Scalable Data Generation stage which takes some user parameters as input and generates the graph as tuples of vertex pairs and their corresponding weights. The first kernel constructs the graph from this edge tuple list, and the subsequent kernels perform various operations on these kernels, namely finding the maximum weighted edges, extracting sub-graphs and highly inter-connected clusters. The SSCA#2 specification document comes with a companion executable specification (a reference implementation for developers to validate their work) which is in MATLAB, with both serial and MATLAB-MPI [15] versions. The MATLAB code and our C/POSIX thread implementations differ significantly. The MATLAB code uses efficient vector operations to perform most functions, and sparse-matrix graph representations are used extensively. We represent the graph using array and linked list data structures, and most of the computation is loop-based and iterative. The MATLAB code runs an order of magnitude slower than our implementation because of the interpreting overhead. However, the MATLAB code is a straightforward implementation, well documented and easy to understand, and is a good starting point for the architecture-specific code developer.

This paper is organized as follows. Section 2 summarizes salient features of symmetric multiprocessors (SMP) such as Sun Enterprise servers and presents a cost-performance model for analyzing the four kernels. Sections 4 through 8 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 9 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.

1.1 Symmetric multiprocessors

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 [3, 4, 1, 2] that provide efficient and scalable performance on SMPs.

The generic SMP processor is a four-way super-scalar microprocessor with 32 to 64 hardware registers, and two levels of cache. The L1 cache is small (64 to 128 KB) and on-chip. It can issue as many words per cycle as the processor can fetch and latency is a few cycles. The size of the L2 cache can vary widely from 256 KB to 8 MB. Bandwidth to the processor is typically 8 to 12 GB per second and latency is 20 to 30 cycles. The processors are connected to a large shared memory (4 to 8 GB per processor) by a high-speed bus, crossbar, or a low-degree network.

Caching and prefetching are two main hardware techniques often used to hide memory latency. Caching takes advantage of spatial and temporal locality, while prefetching mechanisms use data address history to predict memory access patterns and perform reads early. While an SMP is a shared-memory architecture, it is by no means the PRAM used in theoretical work synchronization cannot be taken for granted, memory bandwidth is limited, and performance requires a high degree of locality. The significant features of SMPs are that the input can be held in the shared memory without having to be partitioned and they provide much faster access to their shared memory (an order of magnitude or more) than an equivalent message-based architecture.

To analyze SMP performance, we use a complexity model similar to that of Helman and JáJá [6] which has been shown to provide a good cost model for shared memory algorithms on current symmetric multiprocessors [6, 7, 3, 4]. 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.

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. We list them in this section, but explain their usage in the later sections.

- *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.
- *maxCliqueSize* – the maximum size of a clique in the graph. Clique sizes are uniformly distributed in the interval [1, *maxCliqueSize*].
- *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*]
- *probUnidirectional* – probability that the connections between two vertices will be unidirectional as opposed to bidirectional
- *probInterClEdges* – the probability of inter-clique edges
- *percIntWeights* – percentage of edges assigned integer weights
- *maxIntWeight* – the maximum integer weight
- *maxStrLen* – maximum number of characters in the string weight
- *subGrEdgeLength* – maximum edge length in sub-graphs generated by Kernel 3
- *maxClusterSize* – maximum cluster size generated by the cuts in Kernel 4
- α – Kernel 4's point to start cluster search within *maxClusterSize*

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 untimed, we parallelize the main steps, as its computational complexity (and correspondingly the running time) is greater than any of the subsequent kernels.

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. We discuss Kernel 4 in detail later on, but the salient point to note here is that the intra-clique and inter-clique edge generation determines the quality of results in Kernel 4.

We can divide the data generation phase into five main steps :

- Assigning the graph vertices to cliques of random sizes
- Forming intra-clique edges
- Probabilistically creating inter-clique edges
- Generating edge weights
- Reordering/shuffling the edge lists to remove any locality that might aid subsequent kernels

Since we are using a shared-memory programming model and targeting SMP architectures, we do not need to explicitly divide the problem into sub-problems for various processors, take care of vertex ordering, etc. Random number generation is a critical component of this stage, and we use the SPRNG [17] library for portable and thread-safe pseudo-random number generation. For reporting results and debugging purposes, we would need to generate the same data stream independent of the number of processors involved, and this can be easily done using the same random seed for the SPRNG stream initialization in all cases.

The first step of generation of random cliques sizes can be trivially parallelized as there are no dependencies, and the size of the last clique is modified so that the graph has exactly n vertices. Now we need to assign the n vertices to cliques, ensuring that a vertex belongs to only one clique. One simple way to do this would be to assign the vertices to cliques in sequential order. For example, if clique 1 is of size 3, it is assigned vertices 0, 1 and 2, and clique 2 (of size, say 2) is assigned vertices 3, 4 and so on. The problem with this approach is that it introduces spatial locality in the clique data, which would skew the results of Kernel 4 and defeat the benchmark's purpose. We can overcome this problem by permuting the vertices after edge and weight generation, and then optionally sorting them by their start vertex. We use a fast parallel radix sort algorithm for this step.

While generating intra-clique edges, we need to keep track of the maximum number of possible edges between two vertices, given by the input parameter *maxParalEdges*. The intra-clique edges can be generated in parallel, since the cliques are by definition, independent sets. We use simple dynamic arrays to represent the partial edge lists, and then merge them into one contiguous array after inter-clique generation stage. The inter-clique edge generation phase is very critical to the correct intended usage of Kernel 4, and we generate edges as per the the pseudo-code given in the benchmark specifications. It is possible that the data generator may occasionally generate a clique which is disconnected from the rest of the graph. Alg. 1 summarizes the edge generation steps. Note that both of these steps can be easily parallelized on the SMP, and since we are assuming that the clique sizes are generated according to a uniform random distribution, we will have reasonably good load-balancing on the SMP.

```
Input: array of randomly chosen clique sizes – $cliqueSize$, array storing first element
       in the clique – $firstVsInClique$. We have $\sum |cliqueSize[i]| = totVertices$ and
       $|cliqueSize[i]| \leq maxCliqueSize$
Output: edge vertex pairs stored in $startVertex$, $endVertex$

for $cliqueNum = 1$ to $numCliques$ in parallel do
    for $i = 1$ to $cliqueSize[cliqueNum]$ do
        for $j = 1$ to $cliqueSize[cliqueNum]$ do
            if $i \neq j$ then
                $u \leftarrow firstVsInClique[cliqueNum] + i$;
                $v \leftarrow firstVsInClique[cliqueNum] + j$;
                $numParalEdges \leftarrow$ U[1, $maxParalEdges$];
                Generate $numParalEdges$ edges between $u$ and $v$, randomly omitting
                edges in either the forward or backward direction, depending on the
                input parameter value $probUnidirectional$;
synchronize processors;
for $i = 1$ to $totVertices$ in parallel do
    for ($d = 2$; $d < totVertices$; $d = d * 2, p = p/2$) do
        j $\leftarrow (i + d)$ mod $totVertices$;
        r $\leftarrow$ U[0,1];
        $p \leftarrow probInterClEdges$;
        if ($r \leq p$) and ($i$ and $j$ are not in the same clique) then
            $numParalEdges \leftarrow$ U[1, $maxParalEdges$];
            Generate $numParalEdges$ edges between $i$ and $j$;
synchronize processors;
Merge the partial tuple lists to create a contiguous list in shared memory;
```

Algorithm 1: Clique edge generation

Integer and string weight generation can be done while generating the intra and inter-clique edges, or can be done separately. This step can again be trivially parallelized, and the percentage of edges of each weight type is determined by the input parameter *percIntWeights*. All the character strings are assumed to be of fixed length, given by *maxStrLength*, which is permissible as per the specifications. The integer weights are uniformly distributed in the interval [1, *maxIntWeight*]. The next step after edge generation is to permute and sort the vertices. We do this as detailed in Alg. 2. We need to use locking for the swaps to prevent race conditions. For debugging and reporting results, we permute the vertices sequentially, to generate the same results for all runs. Since the data generator is not timed, we sort the permuted vertex list by the start vertex using a parallel radix sort, which helps us in Kernel 1's graph construction. The vertex numbers and weights are currently 64-bit (can be

```
**Input**: $startVertex$ and $endVertex$ edge lists
**Output**: Permuted $startVertex$ and $endVertex$ edge lists
for $i = 1$ to $totVertices$ in parallel do
    $P[i] \leftarrow i$;
synchronize processors;
for $i = 1$ to $totVertices$ in parallel do
    $t \leftarrow i + U[0, totVertices - i]$;
    if $t \neq i$ then
        swap $P[i]$ and $P[t]$;
synchronize processors;
for $i = 1$ to $totVertices$ in parallel do
    swap $startVertex[i]$ and $startVertex[P[i]]$;
    swap $endVertex[i]$ and $endVertex[P[i]]$;
synchronize processors;
```
Algorithm 2: Generating Vertex Permutations

extended if necessary) and the code is highly parallel, and so the data generator is designed to be *scalable*. Memory usage is currently the bottleneck to scaling on the machines where our experiments run. We could however get around this problem by writing the data generator output to disk, at the expense of affecting Kernel 1 timing.

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 are not provided and needs to be determined in this kernel. It is also suggested that statistics be collected on the graph (which is not timed) to aid verification.

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
- 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 specifications state that multiple edges, edge directions and edges weights have to be ignored. This complicates

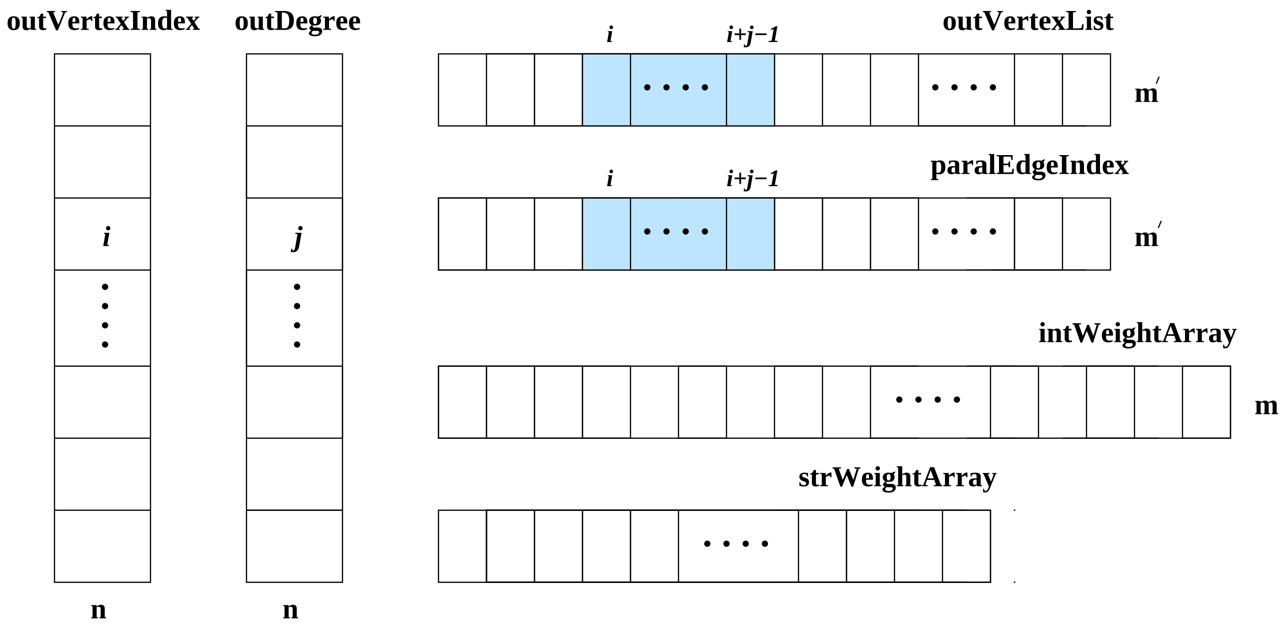

Figure 1: The data layout for representing the directed graph – Kernel 1

things – 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 has to come up with a data 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 suited for dense graphs. In this case, however, the generated graph would be sparse and a matrix representation would be very inefficient in memory usage. Another common method for 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* [18] with auxiliary arrays.

Note that multiple parallel edges between two vertices can be ignored for Kernels 3 and 4, according to the benchmark specification. So we do not store them explicitly in the adjacency lists, but have another array to keep track of the parallel 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 parallel edge information. The data layout used is illustrated in Fig. 1.

With the implied edges, we have two choices – we can either store them in this kernel by searching the directed graph and determining the edges, or just do the searches in Kernel 4 and incur the time overhead there. In the current version of the implementation, we explicitly construct the implied edge list in Kernel 1 itself. An edge $\langle u, v\rangle$ in the *outVertexList* is added to the implied edge list, if there is no directed edge $\langle v, u\rangle$ in the original list. Constructing the implied edge list is the dominant step in Kernel 1, as it involves a large number of non-contiguous memory accesses. We use the adjacency array layout to store the implied edges. We could also use a linked list for small problem scales.

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 an 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 book-keeping 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 edge generation steps are summarized in Alg. 3. 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 the results section.

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.

5.1 Implementation

This kernel can be implemented in a straight-forward manner as described in Alg. 4. 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 start vertices for the elements in S_I can be determined by a fast binary search 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 and $T_C = O(m/p + \log p)$ for this kernel.

```
Input: $startVertex$ and $endVertex$ edge lists of size $numEdges$
Output: $G(V,E)$ represented by $outVertexIndex$, $outVertexList$, $paralEdgeIndex$,
        $impliedEdgeIndex$ and $impliedEdgeList$ arrays
for $i = 1$ to $numEdges$ in parallel do
    $u = startVertex[i]$;
    inspect $endVertex$ array to determine unique vertices originating from $u$;
    update $outDegree[u]$, $outVertexIndex[u]$ and $paralEdgeIndex[u]$;
synchronize processors;
for $i = 1$ to $numVertices$ in parallel do
    add unique vertices adjacent to $i$ in $endVertex$ array to $outVertexList$, which is
    indexed by $outVertexIndex[i-1]$;
synchronize processors;
for $i = 1$ to $numVertices$ in parallel do
    for $j$=$outVertexIndex[i]$ to $j$ =$outVertexIndex[i]$+$outDegree[i]$ do
        if $i$ is not in $j$'s $outVertexList$ then
            update $impliedEdgeIndex[i]$ and $impliedEdgeDegree[i]$;
            add $(i, j)$ to local implied edge list;
synchronize processors;
merge the local implied edge lists to a sorted contiguous edge list;
```

Algorithm 3: Kernel 1 - Edge construction

```
Input: $G(V,E)$ - $outVertexIndex$, $outDegree$, $intWeight$ arrays
Output: Start set for Kernel 3 – $S_I$
$maxWt = -1$;
for $i = 1$ to $numEdges$ in parallel do
    if $intWeight[i]$ >$maxVal$ then
        $maxWt \leftarrow intWeight[i]$;
        find index $j$ for which $paralEdgeIndex[j] \leq i$ and $paralEdgeIndex[j+1] \geq i$;
        store $j$ and $outVertexIndex[j]$;
synchronize processors;
determine global MAX from the $p$ local $maxWt$ values and update $S_I$;
synchronize processors;
foreach element $t$ in $S_I$ do
    in parallel, find index $i$ for which $outVertexIndex[i] \leq j$ and
    $outVertexIndex[i+1] \geq j$;
    $S_{I_t}.startVertex \leftarrow i$;
    $S_{I_t}.endVertex \leftarrow outVertexList[j]$;
synchronize processors;
```

Algorithm 4: Kernel 2 - finding maximum weighted edges

6 Kernel 3 - Extracting sub-graphs

Starting from each of the vertex pairs in the sets S_I and S_C, this kernel produces subgraphs which consist of the vertices and edges along all paths of length less than *subGraphEdgeLength*. 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 *subGraphEdgeLength*. Now *subGraphEdgeLength* 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 algorithm is detailed in Alg. 5. The queue ADT we use in this algorithm has been 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.

```
Input: $G(V, E)$, $S_I$, $S_C$, $t = |S_I| + |S_C|$
Output: $t$ subgraphs $G(V_i, E_i)$, with $V_i$ given by elements in a queue $Q$ populated
        by the BFS, and $E_i \subseteq E$ the corresponding edges in outVertexList

forall end vertices in $S_I$ and $S_C$ in parallel do
    forall $v \in V$ do
        unmark the vertex v;
    mark the source vertex;
    Enqueue(Q, v);
    while while $k <$ subGraphEdgeLength do
        v $\leftarrow$ Dequeue(Q);
        foreach u adjacent to v do
            if u is unmarked then
                mark vertex u;
                Enqueue(Q, u);
        k $\leftarrow$ k+1;
synchronize processors;
```

Algorithm 5: Kernel 3 – Sub-graph extraction

64 Vertices
455 intraclique, 95 interclique edges
14 cliques, maxCliqueSize = 8

Chaco Clustering

2 3 15 26 28 37 39 64	1 8 17 49 53 55 57 59
13 20 21 29 30 44 56 61	4 7 12 19 40 43 47 51
5 6 33 42 50 52 58 62	16 22 25 34 41 46 48 60
10 11 18 23 31 32 38 63	9 14 24 27 35 36 45 54

Chaco partitions the graph into clusters of size 8
Original clique vertices are colored (in shades of blue)

Original Cliques

32	36
42 62	31 63
27 54 9	10 11 23 38
25 34 41 48	16 18 46 60
14 24 35 45 52	5 6 22 33 50 58
2 3 15 26 28 37 39 64	1 8 17 49 53 55 57 59
13 20 21 29 30 44 56 61	4 7 12 19 40 43 47 51

Figure 2: Graph Clustering using Chaco

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 those clusters. The edge directions, multiple edges, and their 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 [13], with all the edge costs considered equal. Sangiovanni-Vincentelli, Chert, and Chua [19, 20] have earlier applied this work for solving circuit problems. The maximal clique problem [5] is a well-studied NP-complete problem, and several heuristics have been proposed to solve this [10]. 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 [8] and METIS [11] to solve this kernel. These tools use a variety of algorithms and are highly refined, but they are primarily used to partition nearly-regular graphs into *equal-sized blocks*, while minimizing edge cut. See Fig. 2 for an illustration of graph partitioning results using Chaco. Identification of smaller clusters in the equal-sized partitions given out by these tools would be an additional step. There are other minor problems in using these tools *as-is* – they require the graph to be in a specific input format, the source code has to be modified to get the output in the format we desire etc.

The specification suggests an algorithm (Alg. 6) for solving this Kernel, which is a variant of a graph clustering algorithm given by Koester [14]. 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 largest of the clusters greater than $\alpha * maxClusterSize$ which has the minimal adjacent set size is selected. 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 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.

The reference greedy algorithm is however inherently sequential. Cliques of size less than $\alpha * maxClusterSize$ with inter-clique edges may not be extracted correctly. We propose a new parallel greedy algorithm (Alg. 7) 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. Now consider the first p vertices in the sorted list. Each processor colors the vertex and its adjacencies (both the out-vertices and the implied edges) with a unique number, given by $i *$ *current iteration number*, where i is the processor index. If two or more vertices which belong to the same clique happen to fall in this group of p vertices, multiple processors will attempt to color a set of vertices simultaneously. Although this will not lead to a race condition, one or more processors might be left with no work to do in the next step. So we suggest a staggered coloring scheme, wherein the first processor colors the first unmarked vertex in the list and its adjacencies and fires off, the second processor waits till the first one is done and then chooses the next unmarked vertex, and so on. This step has a computational complexity of $T_C = O(p)$, and requires a constant number of non-contiguous memory accesses.

In the second step, each processor has a tentative cluster to work with. 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 computation complexity $T_C =$ O(*maxClusterSize*)

```
**Input**: $G(V, E)$
**Output**: List of *cliques* and their corresponding *cut-sets*

for some vertex v remaining in the graph do
    $i \leftarrow 1$;
    while true do
        remove $v$ from the graph;
        $cluster.member[i] \leftarrow v$;
        $adjacencySet[i]$ = all remaining links adjacent to cluster;
        add $v$ to current cluster;
        update the corresponding $adjacencySet[i]$;
        if ($adjacencySet[i]$ == $\phi$) or ($i$ ==maxClusterSize) then
            break;
        find vertex $a$ in $adjacencySet[i]$ with minimum out-degree;
        $v \leftarrow a$;
        $i \leftarrow i + 1$;
    for $j = \alpha * maxClusterSize$ to $i$ do
        $t \leftarrow$ the value of $j$ for which size($adjacencySet[j]$) is minimal;
        store the first $t$ elements in the cluster;
        store the elements in the adjacency list in corresponding cut-set;
    restore other cluster members to the graph;
    remove the cut-set edges from the graph;
    add the selected cluster to clusterList and cut-set edges to cutSetList;
```

Algorithm 6: Reference greedy algorithm for solving Kernel 4

and we only require a constant number of non-contiguous memory accesses, dependent on the size of the cache line (as the adjacency lists are stored in a contiguous block).

These two steps constitute one iteration. The processors are synchronized at the end of each iteration, and the number of vertices assigned to clusters is updated. We continue to iterate in this manner unless we get to a stopping condition – if the assigned vertex number is greater than a certain threshold value, if the degree of vertices is less than $\alpha * maxClusterSize$, if the majority of processors are not being assigned a vertex to start with, and so on. The number of iterations required is $O(n/p)$. We then shift to a sequential algorithm similar to the reference algorithm. If we find uncolored vertices in this step with their degree greater than $\alpha * maxClusterSize$, we can conclude that they were most likely accidentally removed from a cluster (probably due to a high number of inter-clique edges associated with it). Its adjacent vertices are inspected and it is assigned to the cluster to which the majority of its adjacent vertices belong to. 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 clustering algorithm correctly extracts nearly all cliques of size greater than $\alpha *$ *maxClusterSize*. However, it is difficult to extract cliques 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 while 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.

Our implementation (and the reference MATLAB code) uses 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}
- $maxCliqueSize$: $2^{(SCALE/3)}$
- $maxParalEdges$: 3
- $probUnidirectional$: 0.3

```
**Input**: G(V,E), p processors - labelled 1 to p
**Output**: An array vStatus of size n storing the unique cluster number a vertex be-
        longs to, the edge cutSet

Apply parallel radix sort to sort vertices by their degree (= inDegree + outDegree).
Store the non-increasing sorted order in the array sortedList;
for i = 1 to numVertices in parallel do
    vStatus[i] ← 0;
verticesVisited ← 0;
iter ← 1;
pCutSetCounter ← 0;
while true do
    foreach processor j do
        v ← the first unassigned vertex in sortedList;
        foreach vertex t adjacent to v do
            vStatus[t] ← j ∗ iter;
    clusterSize ← 0;
    foreach processor j do
        foreach vertex t adjacent to v do
            adjCount ← 0;
            foreach vertex u adjacent to t do
                if vStatus[u] == j ∗ iter then
                    adjCount ← adjCount +1;
                else
                    add egde ⟨t, u⟩ to pCutSet;
                    pCutSetCounter ← pCutSetCounter +1;
            if adjCount > threshold value then
                update pCutSetCounter to previous value;
                clusterSize ← clusterSize +1;
            else
                vStatus[t] ← 0;
        if clusterSize < threshold value then
            vStatus[$v_j$] ← 0;
            foreach adjacent vertex t do
                vStatus[t] ← 0;
            Update pCutSetCounter;
    synchronize processors;
    verticesVisited ← verticesVisited + SUM(clusterSize);
    if stopping condition then
        break;
apply reference sequential algorithm on remaining vertices;
merge pCutSet lists to form cutSet;
```

Algorithm 7: Kernel 4 – parallel algorithm for graph clustering

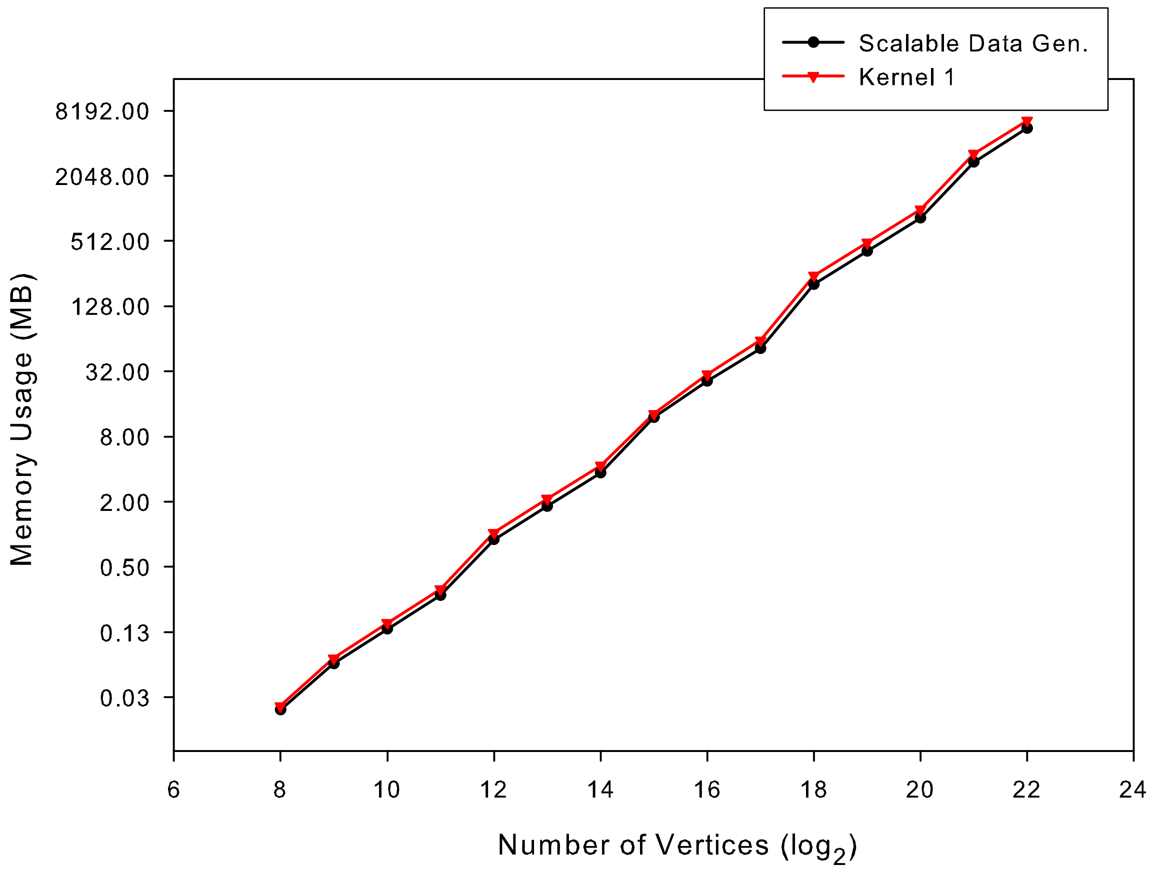

Figure 3: Memory Utilization

- *probInterClEdges* : 0.5
- *percIntWeights* : 0.7
- *maxIntWeight* : 2^{SCALE}
- *maxStrLen* : *SCALE*
- *subGrEdgeLength* : *SCALE*
- *maxClusterSize* : $2^{(SCALE/3)}$
- α : 0.5

Fig. 3 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, due to which the graph data structure utilizes 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 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.

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

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)

Figs. 4 and 5 give 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.

The running times for multi-processor runs are given in Fig. 6. 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.

Rigorous verification of full-scale runs is problematic, and so the benchmark specification suggests a statistical validation scheme. Tab. 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.

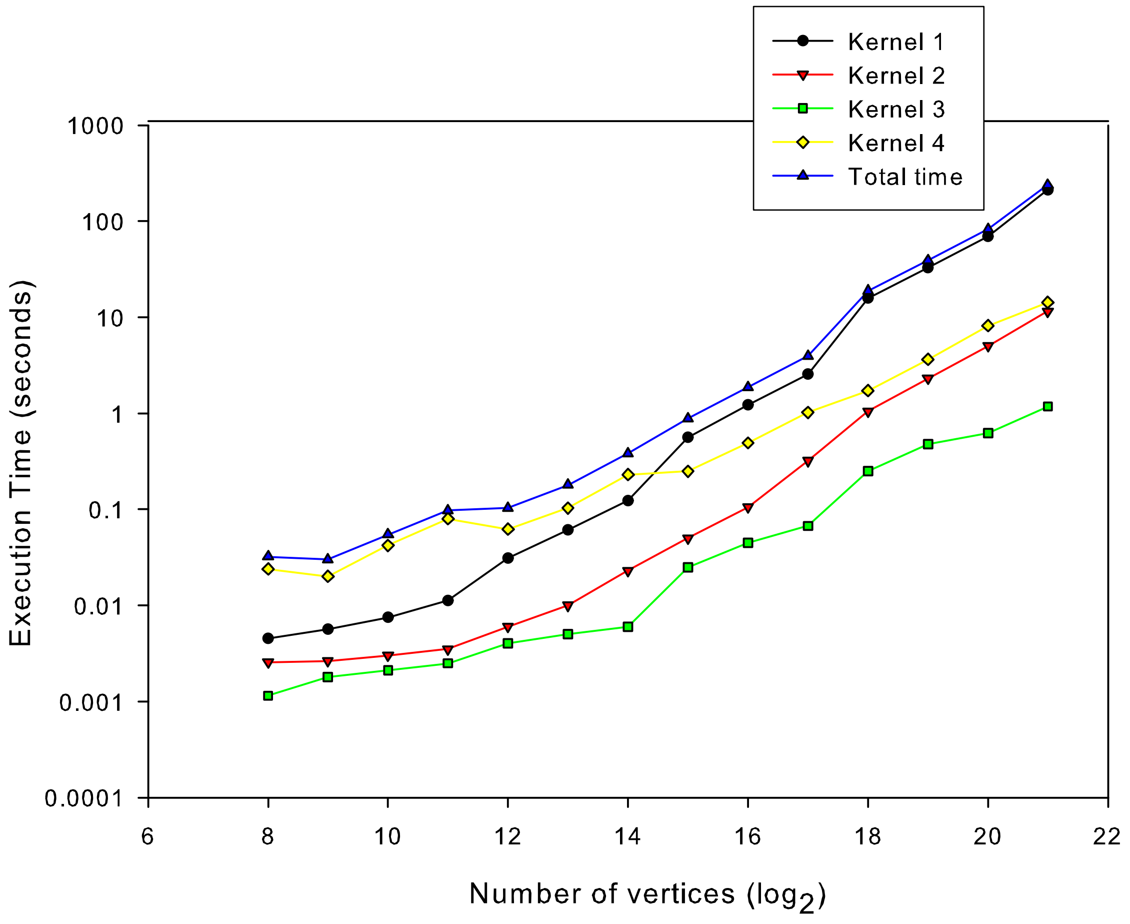

Figure 4: Execution times of kernels 1, 2, 3 and 4 on four processors

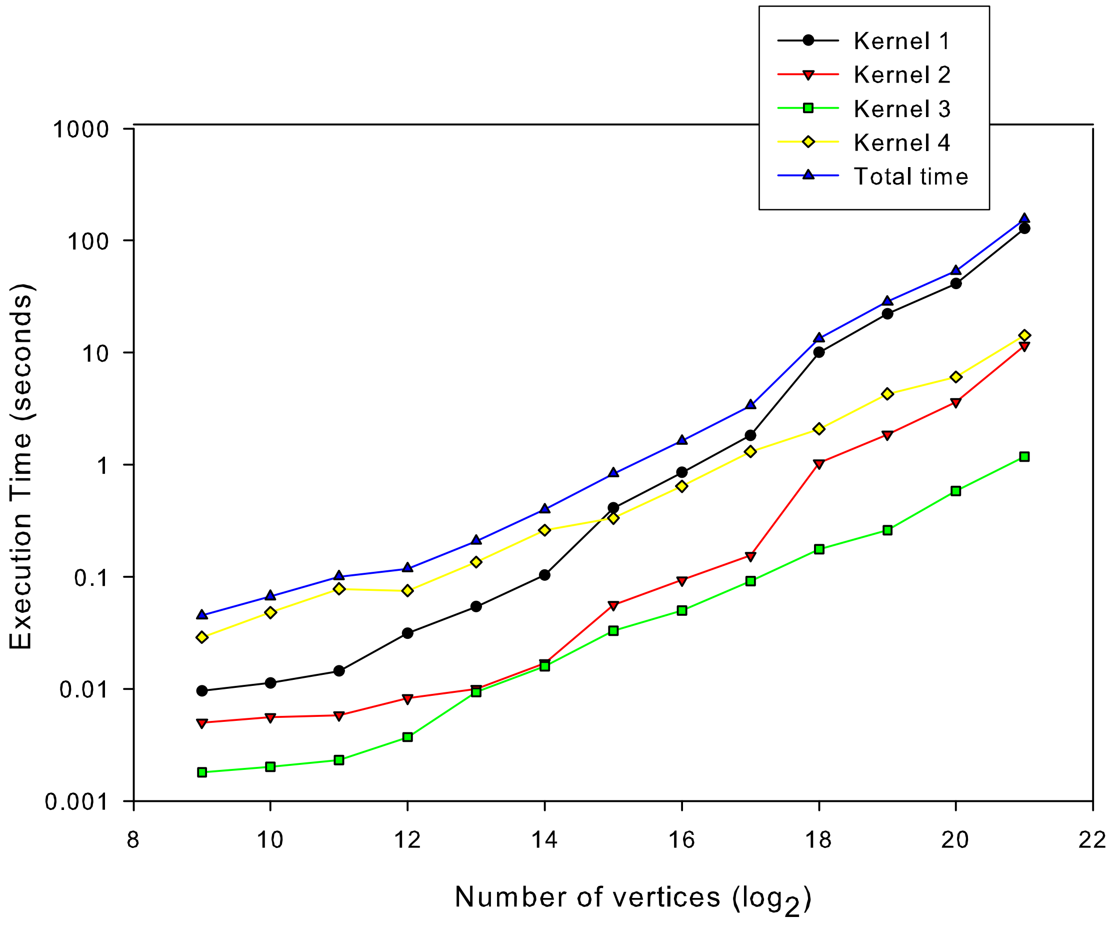

Figure 5: Execution times of kernels 1, 2, 3 and 4 on eight processors

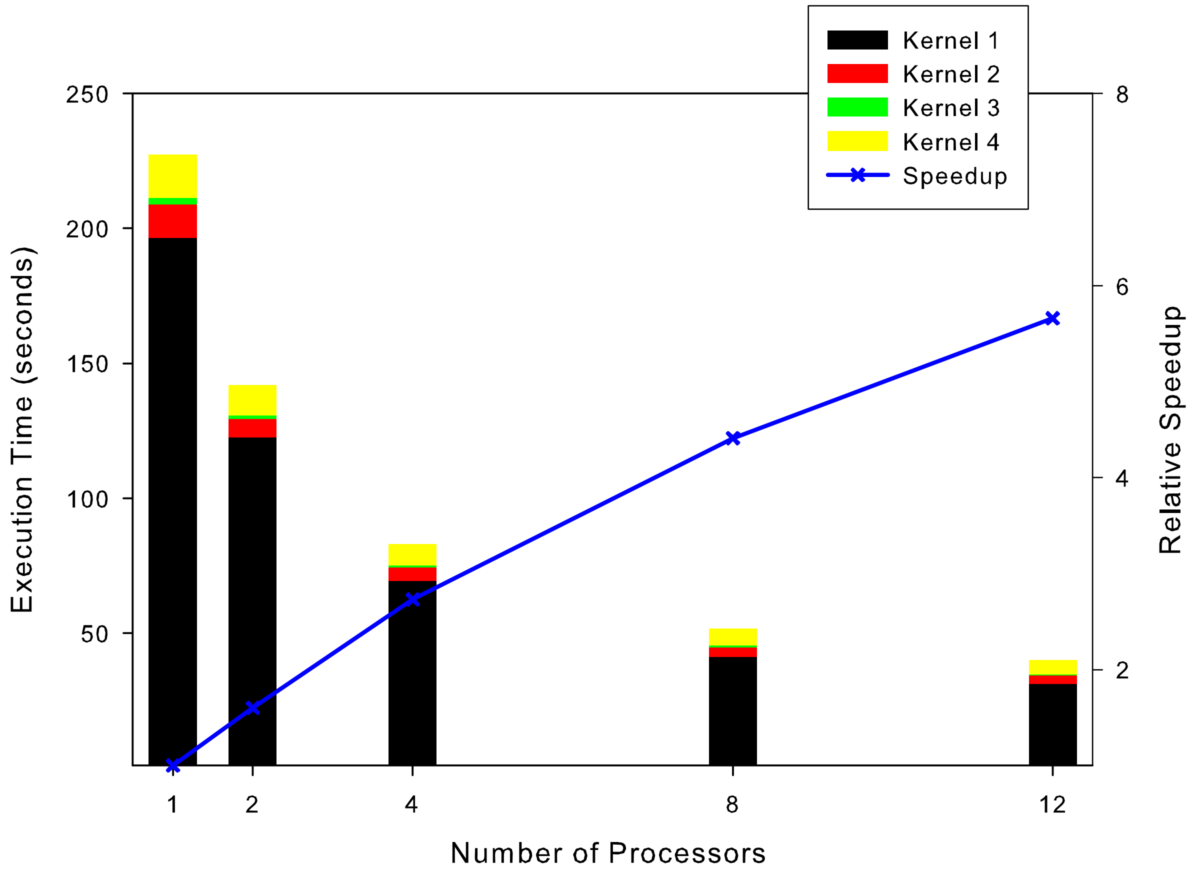

Figure 6: Execution time vs. Processors

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. Also, it cannot be modified by subsequent kernels. We are currently working on implementations of SSCA#2 on other shared-memory systems like the Cray MTA-2 and the Cray XD-1.

Acknowledgements

We thank Bill Mann, Jeremy Kepner, John Feo, David Koester, John Gilbert and other members of the HPCS working group for trying out early versions of our implementation, pointing out bugs in the code, and their valuable suggestions.

References

[1] D. A. Bader and G. Cong. A fast, parallel spanning tree algorithm for symmetric multiprocessors (SMPs). In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2004)*, Santa Fe, NM, April 2004.

[2] D. A. Bader and G. Cong. 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, April 2004.

[3] D.A. Bader, A.K. Illendula, B. M.E. Moret, and N. Weisse-Bernstein. Using PRAM algorithms on a uniform-memory-access shared-memory architecture. In G.S. Brodal, D. Frigioni, and A. Marchetti-Spaccamela, editors, *Proc. 5th Int'l Workshop on Algorithm Engineering (WAE 2001)*, volume 2141 of *Lecture Notes in Computer Science*, pages 129–144, Århus, Denmark, 2001. Springer-Verlag.

[4] D.A. Bader, S. Sreshta, and N. Weisse-Bernstein. Evaluating arithmetic expressions using tree contraction: A fast and scalable parallel implementation for symmetric multiprocessors (SMPs). In S. Sahni, V.K. Prasanna, and U. Shukla, editors, *Proc. 9th Int'l Conf. on High Performance Computing (HiPC 2002)*, volume 2552 of *Lecture Notes in Computer Science*, pages 63–75, Bangalore, India, December 2002. Springer-Verlag.

[5] I. Bomze, M. Budinich, P. Pardalos, and M. Pelillo. The maximum clique problem. In D.-Z. Du and P. M. Pardalos, editors, *Handbook of Combinatorial Optimization*, volume 4. Kluwer Academic Publishers, Boston, MA, 1999.

[6] D. R. Helman and J. JáJá. Designing practical efficient algorithms for symmetric multiprocessors. In *Algorithm Engineering and Experimentation (ALENEX'99)*, volume 1619 of *Lecture Notes in Computer Science*, pages 37–56, Baltimore, MD, January 1999. Springer-Verlag.

[7] D. R. Helman and J. JáJá. Prefix computations on symmetric multiprocessors. *Journal of Parallel and Distributed Computing*, 61(2):265–278, 2001.

[8] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Proc. Supercomputing '95*, San Diego, CA, December 1995.

[9] DARPA High Productivity Computing Systems Project. `http://www.darpa.mil/ipto/programs/hpcs/`.

[10] David J. Johnson and Michael A. Trick, editors. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, Workshop, October 11-13, 1993.* American Mathematical Society, Boston, MA, USA, 1996.

[11] G. Karypis and V. Kumar. *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 edition, September 1998.

[12] J. Kepner, D. P. Koester, and *et al.* HPCS SSCA#2 Graph Analysis Benchmark Specifications v1.0, April 2005.

[13] B.W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.

[14] D. P. Koester. *Parallel Block-Diagonal-Bordered Sparse Linear Solvers for Power Systems Applications.* PhD thesis, 1995.

[15] MIT Lincoln Laboratory. Parallel computing with MATLAB-MPI. `http://www.ll.mit.edu/MatlabMPI/`.

[16] LinPack and the TOP500 Project. `http://www.top500.org/lists/linpack.php`.

[17] M. Mascagni and A. Srinivasan. Algorithm 806: SPRNG: a scalable library for pseudorandom number generation. *ACM Trans. Math. Softw.*, 26(3):436–461, 2000.

[18] J. Park, M. Penner, and V.K. Prasanna. Optimizing graph algorithms for improved cache performance. In *Proc. Int'l Parallel and Distributed Processing Symp. (IPDPS 2002)*, Fort Lauderdale, FL, April 2002.

[19] A. Sangiovanni-Vincentelli, L.K. Chert, and L.O. Chua. An efficient heuristic cluster algorithm for tearing large-scale networks. *IEEE Trans. on Circ. and Syst.*, page 709717, 1977.

[20] A. Sangiovanni-Vincentelli, L.K. Chert, and L.O. Chua. A new tearing approach - node tearing nodal analysis. *Proc. IEEE Int. Symp. on Circ. and Syst.*, pages 143–147, 1977.

[21] HPCS Scalable Synthetic Compact Application (SSCA) Benchmarks. `http://www.highproductivity.org/SSCABmks.htm`.