Space Exploration using Parallel Orbits: a Study in Parallel Symbolic Computing

Vladimir JANJIC [a], Christopher BROWN [a], Max NEUNHÖFFER [b], Kevin HAMMOND [a] Steve LINTON [a] and Hans-Wolfgang LOIDL [c]
[a] *School of Computer Science, University of St Andrews, UK.*
[b] *School of Mathematics, University of St Andrews, UK.*
[c] *School of Mathematical and Computer Sciences, Heriot-Watt University, UK.*

Abstract. Orbit enumerations represent an important class of mathematical algorithms which is widely used in computational discrete mathematics. In this paper, we present a new shared-memory implementation of a generic Orbit *skeleton* in the GAP computer algebra system [5,6]. By defining a skeleton, we are easily able to capture a wide variety of concrete Orbit enumerations that can exploit the same underlying parallel implementation. We also propose a generic cost model for predicting the speedups that our Orbit skeleton will deliver for a given application on a given parallel system. We demonstrate the scalability of our implementation on a 64-core shared-memory machine. Our results show that we are able to obtain good speedups over sequential GAP programs (up to 36 on 64 cores).

Keywords. Parallel symbolic computation, Orbit enumeration, GAP computer algebra system

1. Introduction

Orbit enumerations represent an important class of algorithms that is widely used in computational discrete mathematics, e.g. for the determinisation of finite state automata, processing formal languages (parsing) and natural languages (morphology), computer-aided verification via model checking and modelling of complex systems, for example in UML. Informally, an *Orbit enumeration* is an algorithm that computes the transitive closure for a given set of elements and a given list of *generator* functions that each map an element of the orbit to some (perhaps previously unknown) element. The enumeration terminates when no new elements can be added to the orbit using any of the generators on any element. Typically the number of reachable elements computed in this way is huge (in our test example 1.2 million points out of 10^{228} potential elements) and requires the usage of an efficient administrative data structure for cheap detection of duplicates, e.g. a hash table. While there have been some previous attempts to produce parallel implementations of Orbit enumerations, existing solutions usually do not scale well on larger systems or are tailored to a specific instance of the problem [7]. The two main problems that need to be overcome are that naive parallelisation yields ex-

cessively fine-grained parallelism, and that the shared administrative data structure needs to be implemented both efficiently and without creating a hotspot that will throttle the parallel computation.

This paper presents a new implementation of an *algorithmic skeleton* for Orbit enumerations using the GAP computer algebra system [5]. Algorithmic skeletons are implementations of common parallel patterns, parametrised over worker functions and other implementation-specific information (e.g. to guide task decomposition). A skeleton can be specialised to a specific problem by providing concrete instantiations of the worker function and other information. Orbit enumerations, as studied here, are specific instances of a general search space exploration problem, but with some specific properties arising from the application domain of computational discrete mathematics. The generator functions are typically very fine-grained, the parallel branching factor is often low (generators will often map to known points), and the task pool is both dynamic and self-pruning, with new tasks produced through feedback. Both pruning and task creation must be highly efficient, and perhaps use specialised memory representations and tests.

The specific research contributions of this paper are:

1. We design and implement a new *domain-specific parallel skeleton* for the Orbit algorithm using the GAP computer algebra system;
2. We present performance results for the Orbit skeleton for a realistic instance, demonstrating speedups over the sequential implementation of up to 36 on a 64-core shared-memory system;
3. We present a new abstract cost model for predicting the performance of specific instances of the Orbit skeleton on some parallel hardware, given some simple information about granularity and hardware characteristics;

GAP is the leading open source software for computational group theory. By using GAP, we are able to exploit a rich and complex library of group-theoretic algorithms, developed over two decades by expert mathematicians, and encoding detailed knowledge of many specialised applications in computational discrete mathematics.

2. The Orbit Algorithm

The orbit enumeration problem can be formally described in the following way.

Definition 1 (Orbit of an element)
Let X be a set of points, G a set of generators, $f : X \times G \to X$ a function (where $f(x, g)$ will be denoted by $x \cdot g$) and $x_0 \in X$ a point in X. The orbit *of x_0, denoted by $\mathcal{O}_G(x_0)$, is the smallest set $M \subseteq X$ such that*

1. $x_0 \in M$
2. *for all $y \in M$ and all $g \in G$ we have $y \cdot g \in M$.*

It is worth noting that the set X can be very large and is not given *a-priori*. The basic sequential orbit enumeration algorithm starts with two lists, $M = [x_0]$ (orbit points discovered so far) and $U = [x_0]$ (a set of unprocessed points), and a

hash table $T = \{x_0\}$. In each step, we remove the first point x from U, apply all generators $g \in G$ to x and add the resulting points $x \cdot g$ (that are not in M) to M, T and U. This eventually terminates (when all $g \in G$ have been applied to all $x \in M$), discovering the whole of the orbit $\mathcal{O}_G(x_0)$. The sequential algorithm is shown below.

```
   Data: x_0 ∈ X, g_1, g_2, ..., g_k : X → X
   Result: A list M containing the orbit of the element x_0
1  T := {x_0} (a hash table); M := [x_0] (a list); U = [x_0] (a list);
2  while U is not empty do
3  |  x := head of U; Remove x from U;
4  |  for i ← 1 to k do
5  |  |  y := x · g_i;
6  |  |  if y ∉ T then
7  |  |  |  Add y to T;
8  |  |  |  Add y to the end of U;
9  |  |  |  Append y to the end of M;
```

Figure 1. Sequential Orbit algorithm pseudocode

At a first glance, it may seem that the sequential Orbit algorithm can be parallelised by carrying out step 5 for all generators and all points in U independently (preserving the ordering of results). However, there is a synchronisation part (steps 6–9) where the duplicates need to be discovered. This requires communication among the tasks that apply generators to different elements of U. We describe our solution to this problem in Section 4.

3. The GAP Computer Algebra System

GAP is the leading free system for computational discrete algebra (`http://www.gap-system.org`). It is designed to be natural to use for mathematicians; to be powerful and flexible for experts and to be freely extensible so that it can encompass new mathematics. GAP has many features that are particularly suitable for large-scale Orbit computations in Computer Algebra. It supports very efficient linear algebra over small finite fields, multiple representations of groups, subgroups, cossets and different types of group elements, and backtrack search algorithms for permutation groups. It provides basic parallel programming primitives, plus higher-level automatic task placement, communication and load balancing techniques.

4. The Parallel Orbit Skeleton

Our Parallel Orbit skeleton uses two different kinds of threads:

- *Hash server* threads, that use hash tables to determine which points are duplicates; and

```
Data: the set G, the action function X × G → X, the number h of hash
      servers and a distribution hash function f : X → {1, ..., h}
while true do
    get a chunk C of points ;
    R := a list of length h of empty lists;
    for x ∈ C do
        for g ∈ G do
            y := x · g;
            append y to R[f(y)];
    for j ∈ {1, ..., h} do
        schedule sending R[j] to hash server j;
```

Figure 2. Pseudo code for a worker

```
Data: A chunk size s
initialise a hash table T;
while true do
    get a chunk C of points from our input queue;
    for x ∈ C do
        if x ∉ T then
            add x to T;
            if at least s points in T are unscheduled then
                schedule a chunk of size s points for some worker;
    if there are unscheduled points in T then
        schedule a chunk of size < s points for some worker;
```

Figure 3. Pseudo code for a hash server

- *Worker* threads that obtain chunks of points from hash servers and apply generators to them, so producing new points.

This means that we have explicit two-way communication between each hash server and each worker, but no communication between different hash servers or different workers. Where there are multiple hash servers, we use a distribution hash function to decide which hash servers is responsible for storing each point.

Figure 2 shows the pseudo code of a worker thread and Figure 3 the pseudo code of a hash server thread. Initially, all hash servers and input queues are empty and all workers are idle. We first feed a single point into one of the hash servers, which immediately creates some work. This hash server sends this work to a worker which, in turn, produces more points that are then sent to the corresponding hash servers. This bootstraps the whole computation. The number of points in the system increases and, eventually, all input queues become completely full and the system as a whole becomes busy. Towards the end of the computation, fewer new points are discovered. Eventually all generators will have been applied to all points and the computation terminates. Now all workers are idle again and the hash servers collectively know all the points in the orbit. In our GAP implementa-

tion we maintain one channel for the input queue of each hash server. Any worker can write to any of these channels. We also maintain a single shared channel for the work queue. Due to a chunking mechanism, this is not a bottleneck.

Cost Model: Our orbit enumeration algorithm has essentially two basic operations: the first, ACT, is acting with a generator g on a point x resulting in $x \cdot g$, and the second, LOOKUP, is looking up a point x in a hash table. Let A be the number of ACT operations a worker can do per second and L be the number of LOOKUP operations a hash server can do per second.

Assumptions 1
We assume the following idealising facts:

1. *All workers operate at the same speed described by A.*
2. *All hash servers operate at the same speed described by L.*
3. *The distribution hash function actually distributes the points in the orbit uniformly to the hash servers.*
4. *All points in the orbit are discovered equally many times during the computation*[1]*.*
5. *Any task can send data to any other task with the same bandwidth.*
6. *Concurrent communication between different pairs of tasks does not lower the bandwidth.*
7. *All latency of communication is successfully avoided by queueing.*
8. *Communication between tasks can happen concurrently to lookup (in the hash servers) and acting (in the workers), and does not slow down either one of the operations* LOOKUP *and* ACT*.*
9. *The rate of discovery of new points is fast enough to fill our queues.*

In our shared memory implementation all latency and communication assumptions are trivially fulfilled. In practical applications we easily find a distribution hash function for assumption 3.

Theorem 2 (A-priori cost estimate)
Let $\mathcal{O}_G(x_0)$ have size N and let G have size k. Let w be the number of workers and h be the number of hash servers. Assume that the Assumptions 1 hold. Then the runtime of our algorithm is approximately

$$\max\left\{\frac{kN}{wA}, \frac{kN}{hL}\right\}$$

5. Performance Results

In this section, we present a preliminary evaluation of our Parallel Orbit skeleton on a shared-memory machine consisting of 4 AMD Opteron 6376 processors, each comprising 16 cores (giving us a 64-core system). Each core runs at 1.4GHz and has 16Kb of private L1 data cache. In addition, each processor has 8x64Kb of L1 instruction caches and 8x2Mb of L2 caches (shared between two cores),

[1] This is usually fulfilled in the examples from the computational group theory

and 2x8Mb of L3 caches (shared between 8 cores). As an instance of an Orbit enumeration, we are considering the sporadic finite simple Harada-Norton group in its 760-dimensional representation over the field with 2 elements, acting on vectors. The set X of all possible points consists of about 10^{228} elements. The size of the orbit is 1.4 million points, and we are using 10 random generators. On each figure, we consider mean speedups over 5 runs of the same instance. We have also added error bars in the cases where error margin was notable.

Figure 4 shows the absolute speedups (when using the sequential Orbit algorithm executed with one thread as a base time) that we have obtained with 1 to 4 hash server threads, and 1 to 32 worker threads. We can observe very good speedups, up to 23 with 4 hash servers and 28 worker threads (so, using 32 cores in total). We can also observe that about 7 workers produce enough points to fully load one hash server. Also, for a fixed number of hash servers h, the speedups grow almost linearly with the increase in number of workers. Once the number of workers reaches $7h$, we do not observe further increase in speedups. This is exactly what our cost model predicts. Note further that, on our testbed, it is not possible to obtain linear speedups, due to frequency scaling of cores when multiple cores are used at the same time. Therefore, the figure also shows ideal speedups that could be obtained. The ideal speedup on n cores was estimated by taking the runtime of the sequential Orbit algorithm in one thread (and having $n-1$ parallel threads executing a dummy spin locking function that does not use any memory, to scale down the frequency) and then dividing it by n.

We were able to obtain similar results on other shared-memory architectures. Due to space limitations, we omit these results here. We refer reader to

`http://jv.host.cs.st-andrews.ac.uk/orbitResults.pdf`

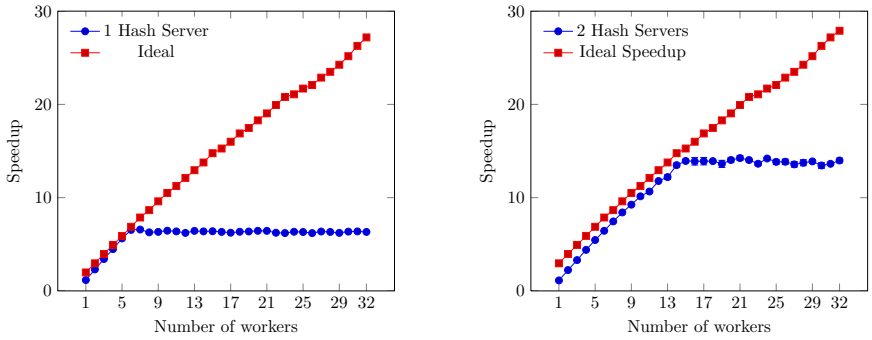
for additional results.

To test the scalability further, Figure 5 shows the speedups obtained on the same machine with 7 and 8 hash servers, and up to 57 workers, therefore using all 64 cores in the system. The speedups steadily increase in both cases from 18 to 35 (with 7 hash servers) and 36 (with 8 hash servers). The fact that the speedup obtained is suboptimal can be attributed both to the problem itself, since in these cases the bootstrapping and finishing phases (where not enough points are available to utilise all workers) become dominant, and to the hardware, since there are only 16 memory channels, which limits the memory access when more hash servers and workers are used.

6. Related Work

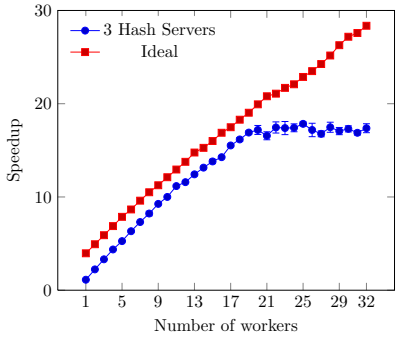
A more formal description of the Orbit algorithm can be found in [4]. There are unfortunately very few detailed descriptions of Orbit implementations, other than the one we have given here. One exception is a parallel C implementation using hash tables [7]. This implementation is, however, designed specifically for matrix groups. It is therefore not a general parallel skeleton, as described here.

Algorithmic skeletons [2], or patterns of parallel computation [8], have proven to be effective for isolating the parallel coordination from the computation aspects of an application. The most prominent generic pattern is Google's MapRe-

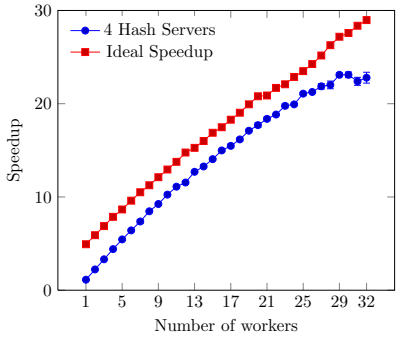


(a) 1 Hash Server

(b) 2 Hash Servers

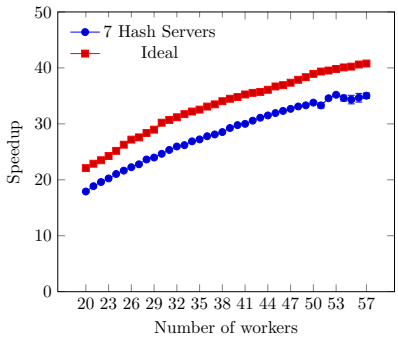


(c) 3 Hash Servers

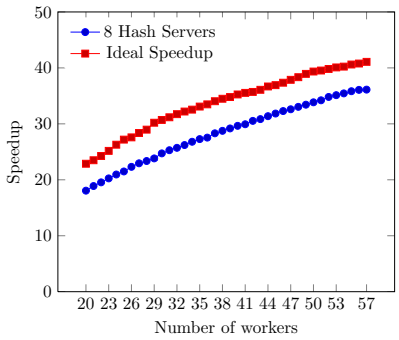


(d) 4 Hash Servers

Figure 4. Absolute speedups for the Parallel Orbit skeleton for the HN instance



(a) 7 Hash Servers



(b) 8 Hash Servers

Figure 5. Absolute speedups for the Parallel Orbit skeleton for the HN instance with 6 and 7 hash servers

duce [3], which has been applied to hundreds of concrete applications. Emerging practitioner-oriented textbooks on how to program multi-cores promote parallel pattern approaches [1,9]. In contrast to this work, we present a domain-specific pattern that accounts for the dynamic characteristics typical in computer alge-

bra, namely dynamic parallelism, and we apply it to foundational computational mathematics code, which itself has numerous instances in concrete application domains.

7. Conclusions and Future Work

We presented a domain-specific Orbit skeleton, achieving speedups of up to 36 on a 64-core shared-memory machine. In addition, we proposed a cost model for estimating the speedups that can be obtained with the Orbit skeleton on a given parallel machine.

By implementing a skeleton for a widely used computational pattern in computational discrete mathematics, we enable parallelism for a wide class of important algorithms, such as the determinisation of finite state machines. Being of foundational nature, these instances are applied in many different application domains, such as natural language processing. In developing the Orbit skeleton, we addressed several issues that are characteristic for parallel computational discrete mathematics algorithms: the parallelism in the application is highly dynamic, with the number of potential parallel tasks varying widely over time; it is crucial to gauge the parallelism for early peak utilisation and to coalesce individual computations to achieve suitable granularity.

As a future work, we plan to implement a distributed-memory version of the Orbit skeleton and to evaluate it both on a single cluster and on parallel systems comprising geographically-distributed clusters. We also plan to extend the cost model for this setting and to verify more rigorously that it can accurately predicts the speedups that will be obtained.

References

[1] C. Campbell, R. Johnson, A. Miller, and S. Toub. *Parallel Programming with Microsoft .NET — Design Patterns for Decomposition and Coordination on Multicore Architectures*. Microsoft Press, Aug. 2010. ISBN 978-0-7356-5159-3.

[2] M. Cole. *Algorithmic Skeletons: Structural Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[3] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, January 2008.

[4] B. E. Derek F. Holt and E. A. O'Brien. *Handbook of Computational Group Theory*. Chapman and Hall/CRC, 2005.

[5] The GAP Group. *GAP – Groups, Algorithms, and Programming, Version 4.6.2*, 2013.

[6] S. Linton, K. Hammond, A. Konovalov, C. Brown, P. W. Trinder, H. W. Loidl, P. Horn, and D. Roozemond. Easy Composition of Symbolic Computation Software Using SCSCP: A New Lingua Franca for Symbolic Computation. *J. Symb. Comput.*, 49:95–119, Feb. 2013.

[7] F. Lübeck and M. Neunhöffer. Enumerating large orbits and direct condensation. *Experiment. Math.*, 10(2), 2001.

[8] T. Mattson, B. Sanders, and B. Massingill. *Patterns for parallel programming*. Addison-Wesley Professional, first edition, 2004.

[9] M. McCool, J. Reinders, and A. Robison. *Structured Parallel Programming: Patterns for Efficient Computation*. Morgan Kaufmann Publishers, July 2012. ISBN 9780124159938.