HPC-GAP Case Study: Finding Matrices of Invariant Bilinear Forms in the Representation Theory of Hecke Algebras

Patrick Maier

07 Oct 2013

1 Problem

The context is the representation theory of Hecke algebras. Given generators $M_1, \ldots, M_m$, where each M_i is a square matrix of polynomials in $\mathbb{Z}[x]$, the problem is to find a (non-trivial) symmetric square matrix Q of Laurent polynomials in $\mathbb{Z}[x, x^{-1}]$ such that the product of Q with each generator is itself symmetric, i. e., such that the following equation holds for all k:

$$Q \cdot M_k = M_k^{\mathsf{T}} \cdot Q \tag{1}$$

Note that this implies that the dimension of Q equal that of the generators. In Hecke algebras, such matrices Q exist and are unique up to scalar multiplication. Moreover, the theory provides us with tight bounds on the upper and lower degree of the Laurent polynomials occuring in Q.

Depending on the Hecke type E_m ($m = 6, 7, 8$) there are several *representations* of generators. For E_6, there are 23 representations, with dimensions varying from 6 to 90, and a spread between the upper and lower degree varying between 29 and 53. E_7 has 58 representations, with dimensions varying from 7 to 512, and degree spread varying from 45 to 95. Finally, E_8 has 110 representations of dimensions between 8 and 7168, and degree spread ranging from 65 to 185.

2 Algorithm

In principle, the problem can be solved by treating equation (1) as a system of linear equations and solving for the entries of Q. However, solving linear

systems over $\mathbb{Z}[x, x^{-1}]$ is not very efficient, and there is no hope of obtaining solutions with this direct method for the higher dimension representations of E_7 and E_8.

Instead, we solve the problem by interpolation. We view each entry of Q as a Laurent polynomial with $u-l+1$ unknown coefficients, where $u-l+1$ is the spread between lower degree bound l and upper degree bound u. Evaluating Q at enough values will provide enough information to compute the unknown coefficients by solving linear systems over the rationals instead over $\mathbb{Z}[x, x^{-1}]$. To avoid dealing with very large rational numbers (due to polynomials of high degree), we will perform most of the computations in homomorphic images over small prime fields and use Chinese Remainder Theorem to recover any information lost.

The algorithm takes as input generators $M_1, \ldots, M_m$ (of dimension n), lower and upper degree bounds l and u, and a finite set of primes P. From the degree bounds, we construct a set V_{lu} of $u-l+1$ small integers (excluding zero) to be used as values for interpolation. The primes in P must be chosen large enough not to divide any of the values in V_{lu}. The algorithm runs in three phases:

1. For each prime $p \in P$ and value $v \in V_{lu}$, *generate* a solution Q_{pv} of (1) by instantiating the unknown x with v and solving the resulting system modulo p.

2. *Reduce* all Q_{pv} by Chinese remaindering and obtain linear systems for the coefficients of the entries of Q. (Since Q is symmetric, there are $\frac{1}{2}n(n+1)$ such systems, each of dimension $u-l+1$.) Solve these systems over the rationals and construct Q.

3. For all k, *check* that the resulting Q does satisfy equation (1) over $\mathbb{Z}[x, x^{-1}]$.

Note that the theory of Hecke algebras admits a particularly efficient way to compute Q_{pv} in the *generate* phase. Instead of solving a linear system, Q_{pv} is generated by *spinning* the basis e of a (pre-determined) one-dimensional sub-space. Here, *spinning* the basis means multiplying this basis vector e with n (pre-determined) products of the generators M_k, generating the n rows of Q_{pv}.

Typically, many of the $\frac{1}{2}n(n+1)$ entries of Q are duplicate. Therefore, the *reduce* phase includes a step filtering duplicates to avoid solving the same linear systems several times over. Often (but not always) this reduces the workload of the reduce phase by a factor of 5 to 10.

2.1 Improvements to sequential implementation

Parallelising Daria's GAP code has resulted in a number of improvements to the sequential implementation, some with dramatic effect. The three most important improvements are listed here.

1. Storing the generators in an array of lambda abstractions (rather than as a single lambda abstraction of arrays). This change improved the sequential runtime for the complete batch of E_6 representations by a factor of 12.

2. Sparse block matrix representation of generators. This change improved the memory consumption dramatically. Without sparse matrix representation, even just reading the generators for E_7 consumed more than 4 GB of memory; in sparse matrix representation, the generators for E_7 fit into about 80MB of memory.

3. Spinning the basis by computing repeated vector/matrix products (rather multiplying the basis vector to a matrix product of generators). This change improved the sequential runtime for the complete batch of E_6 representations by a factor of by a factor of 28.

All in all, the improvements resulted in a sequential runtime (on a Intel Core i7 at 2.8 GHz) for the complete batch of E_6 representations of about 17 minutes, down from about 100 hours, a reduction by a factor of about 350.

3 Parallelisation

Each of the three phases the above algorithm can be parallelised. However, how to parallelise depends to some extent on the architecture of the parallel system, as its structure and latencies dictate to some extent what is useful/easy to parallelise and what is not.

Figure 1 shows the architecture of SymGridPar2 (SGP2), which was used for this case study. SGP2 is a middleware coordinating GAP servers over the network. In this case study, SGP2 coordinated a single master GAP server and n worker GAP servers, distributed over m nodes. All GAP servers run standard (single threaded) GAP 4.6. Experiments were run on two platforms, a Beowulf cluster comprising 8-core nodes located at Heriot-Watt, and HECToR, the UK national super computer comprising 32-core nodes.

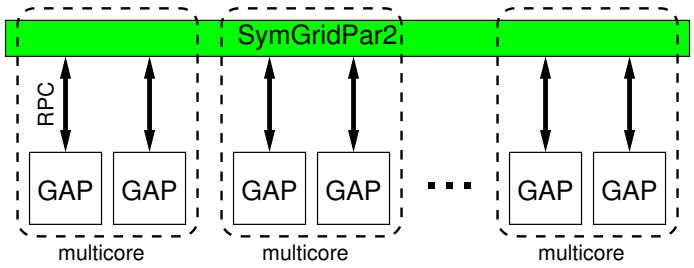

Figure 1: SGP2 Architecture.

On the Beowulf cluster, SGP2 coordinated $n = 7(m-1)+1$ workers, i. e., 7 single-threaded workers per 8-core node in general, but only 1 worker on the root node. One core per node remained reserved for the SGP2 middleware. The reason for limiting the number of workers on the root node to one is that the root node also hosts the master GAP server, which typically requires much more memory than the workers. Similarly, on HECToR SGP2 coordinated $n = 31(m-1)+1$ workers, i. e., 31 single-threaded workers per 32-core non-root node, plus 1 worker on the root node.

3.1 Parallelisation of the Generate phase

The *generate* phase produces independent homomorphic images, one per prime/value pair $(p, v) \in P \times \{l, \ldots, u\}$. That is, this phase generates $|P|(u-l+1)$ independent tasks to compute the homomorphic solutions Q_{pv}. All these tasks can be run in parallel. Moreover, the input to each task is very small, only the pair of integers (p, v), so the tasks are easily distributed across the network. However, the output of each task is Q_{pv}, an $n \times n$ matrix over $\mathsf{GF}(p)$, so the amount of data to be communicated back to the master GAP server grows quadratically with the dimension of the generators.

Note that SGP2 uses a `parBufferTryReduce` skeleton to constantly feed matrices Q_{pv} back to the master GAP server, rather than gathering all matrices and returning them back at once. This allows the master GAP to commence pre-processing for the *reduce* phase even while the *generate* phase is still active.

Possible improvements: Each row of Q_{pv} is generated independently by spinning a basis vector. Hence, the generate phase could produce up to $n|P|(u-l+1)$ independent parallel tasks, with a total cost of communica-

tion almost as low as when computing Q_{pv} sequentially. This improvement isn't worthwhile for the small dimensions seen in E_6 and E_7 since the granularity of tasks would become too small, but it may prove useful for the E_8 representations, many of whose dimension is above 1000.

3.2 Parallelisation of Reduce phase

The *reduce* phase produces up to $\frac{1}{2}n(n+1)$ independent linear systems (over the rationals) to determine the coefficients of the entries of Q. Each linear system is of dimension $u - l + 1$. Fortunately though, only the right-hand side vector of the system needs to be communicated since the matrix can be generated on the fly. Thus both the input and the output of reduce tasks is are vectors of rationals of size $u - l + 1$.

Unfortunately, solving these linear systems isn't all that the *reduce* phase is doing. The rational vectors determining the systems are computed from the Q_{pv} matrices in a complex pre-processing step, including filtering out duplicates and Chinese remaindering. After solving the linear systems, a post-processing step actually constructs the matrix Q, reversing the previous elimination of duplicates. The pre- and post-processing steps are currently sequential. Their complexity increases with n and with the number of unique entries in Q; for the 20 biggest cases of E_7, the runtime of this sequential part of the reduction phase dominates the parallel runtime, largely obliterating parallel performance.

Possible improvements: The sequential pre-processing phase consists of several steps, at least some of which could be parallelised (as they appear to be simple loops). However, parallelising these loops would require to communicate vast amounts of data (essentially all the data stored in the Q_{pv} matrices, which can run to several GB). Moreover, the sequential code introduces synchronisation points at the beginning and end of each loop.

To obtain better performance, the *reduce* phase should be rewritten from scratch, with a view to removing synchronisation points. Alternatively, the code could be re-written to make use of the multithreading capabilities of HPC-GAP; then the master GAP server could be run on a large multicore machine, exploiting shared memory parallelism to speed up the pre- and post-processing.

3.3 Parallelisation of Check phase

Obviously, the equation (1) can be checked for each generator independently; that is, the *check* produces m parallel tasks. The communication costs for these tasks are high (at least in a distributed-memory setting) since each task requires the full matrix Q, and the size of Q is bounded by $n^2(u-l+1)$ in the worst case. The return value of each task is small, though: a single bit indicating whether or not Q satisfies equation (1) for generator M_k.

Possible improvements: Equation (1) could be checked row by row. That is, the *check* phase could produce nm parallel tasks. Unfortunately, each task would still require the full matrix Q, and transmitting the matrix to all GAP workers may consume too much bandwidth.

Alternatively, (1) could be checked entry by entry. That is, the *check* phase could produce $\frac{1}{2}n(n+1)$ parallel tasks, each checking that entry q_{ij} of Q satisfies equation (1) w. r. t. all m generators. This would only require the ith row and jth column of Q to be transmitted per task, hence per-task communication costs would be linear in the dimension n rather than quadratic. (If granularity of entry-by-entry checking is too low, Q could be divided into blocks, much like Gentleman's algorithm for matrix multiplication, retaining linear per-task communication costs.)

4 Results

This section presents the results of computing bilinear forms using SGP2. Most experiments were performed on a Beowulf cluster at Heriot-Watt University; the cluster features 8-core nodes, each with 12GB RAM. Due to the 12GB RAM limit becoming a problem, some experiments for E_8 had to be performed on Cantor, a 48-core server with 512GB RAM at Heriot-Watt University. A small set of experiments was performed on HECToR, which features 32-core nodes with 32GB RAM, mostly to demonstrate that SGP2 does run on HECToR.

In the following sections, we present a complete set of results for E_6 and E_7, including scaling experiments on all representations of E_6 and some representations of E_7. For E_8, we only managed to compute the first 9 representations, where the dimensions of the generators remain below 100. Lack of memory on the Beowulf cluster, and the dominance of sequential pre-/post-processing during the *reduce* phase prevented us going further.

We note that confidence in the measurements on the Beowulf cluster is not very high. Since the cluster lacks a batch scheduling system, runtimes

may have been distorted by competing experiments.

4.1 E_6 on HWU Beowulf cluster

Table 1 shows the runtimes of a scaling experiment for all 23 representations of E_6 on the Beowulf cluster at Heriot-Watt. Each row of the table corresponds to one representation, identified by the first column. The second and third columns show the dimension n of the generators and the spread between lower and upper degree bounds (which equals the number of values used for interpolation). The fourth and fifth columns display the number of Q_{pv} matrices (computed in the *generate* phase) and the number of unique entries q_{ij} of Q (computed in the *reduce* phase); these two figures equal the number of parallel tasks in the *generate* and *reduce* phases. The sixth column displays the time for computing Q sequentially in a single GAP server. The remaining three columns show the runtime of SGP2 using 4, 8 and 16 nodes of the cluster, respectively, which corresponds to using 22, 50 and 106 GAP workers, respectively. All runtimes are in seconds and exclude the time for starting up GAP.

Note that $\#Q_{pv}/|V_{lu}|$ equals the number of primes used. The primes were drawn in descending order from the set of primes less than 255. In this experiment, a minimal set of primes was chosen for each representation.

Figure 2 presents absolute speedup graphs derived from Table 1; the figure also plots the fill ratio of Q (in percent), i. e., the ratio of number of unique entries (column $\#q_{ij}$) divided by the number $\frac{1}{2}n(n+1)$ of entries of Q (taking symmetry into account).

As the figure shows, the E_6 representations do not scale very well, mostly because their sequential runtime is already too low to benefit much from parallelism. The figure also shows that spikes in the fill ratio correspond to spikes in speedup. This is because the higher the fill ratio, the more parallel tasks are generated during the reduce phase.

4.2 E_7 on HWU Beowulf cluster

Tables 2 and 3 shows the runtimes of all 58 representations of E_7 on 16 nodes (106 GAP workers) of the Beowulf cluster at Heriot-Watt. The first five columns of these tables are the same as for E_6, and the sixth column is the parallel runtime of SGP2. The remaining three columns display the average runtimes of the parallel tasks of the *generate*, *reduce* and *check* phases. From these three columns and the columns $\#Q_{pv}$ and $\#q_{ij}$ we can estimate the sequential runtime (or rather, the sequential equivalent to the

rep	n	$\lvert V_{lu}\rvert$	$\#Q_{pv}$	$\#q_{ij}$	t_{seq}	t_{22}	t_{50}	t_{106}
3	10	29	522	6	2.741	2.442	2.511	2.108
4	6	53	1961	12	17.715	12.255	8.814	8.426
5	6	53	2014	3	13.350	9.065	8.627	8.115
6	20	29	522	11	4.456	2.031	2.247	2.009
7	15	37	814	17	8.383	3.955	4.269	3.236
8	15	37	925	7	7.867	3.064	3.692	3.988
9	15	37	814	17	8.306	3.171	3.081	3.139
10	15	37	888	4	7.016	3.041	3.199	3.613
11	20	45	1395	79	48.355	7.373	5.903	6.041
12	20	45	1395	4	15.164	5.560	4.978	6.486
13	24	37	962	24	13.673	6.003	3.947	4.327
14	24	37	962	9	13.288	4.401	3.706	3.368
15	30	37	851	124	35.985	7.335	5.442	6.152
16	30	37	851	15	16.544	4.594	4.440	4.532
17	60	29	580	67	32.513	12.141	6.842	7.587
18	80	29	522	379	69.614	16.396	13.424	15.977
19	90	29	493	163	63.870	15.185	14.994	14.295
20	60	33	660	143	46.158	8.652	7.364	8.728
21	60	33	726	39	37.272	8.684	9.020	7.368
22	64	35	735	323	83.704	13.962	10.621	10.225
23	64	35	770	21	41.287	11.610	7.416	7.389
24	81	33	660	68	57.036	12.396	11.177	10.823
25	81	33	627	45	47.669	11.634	9.748	11.256

Table 1: Scaling E_6, runtimes on Beowulf cluster: sequential and parallel with 22 (4 nodes), 50 (8 nodes) and 106 (16 nodes) GAP workers.

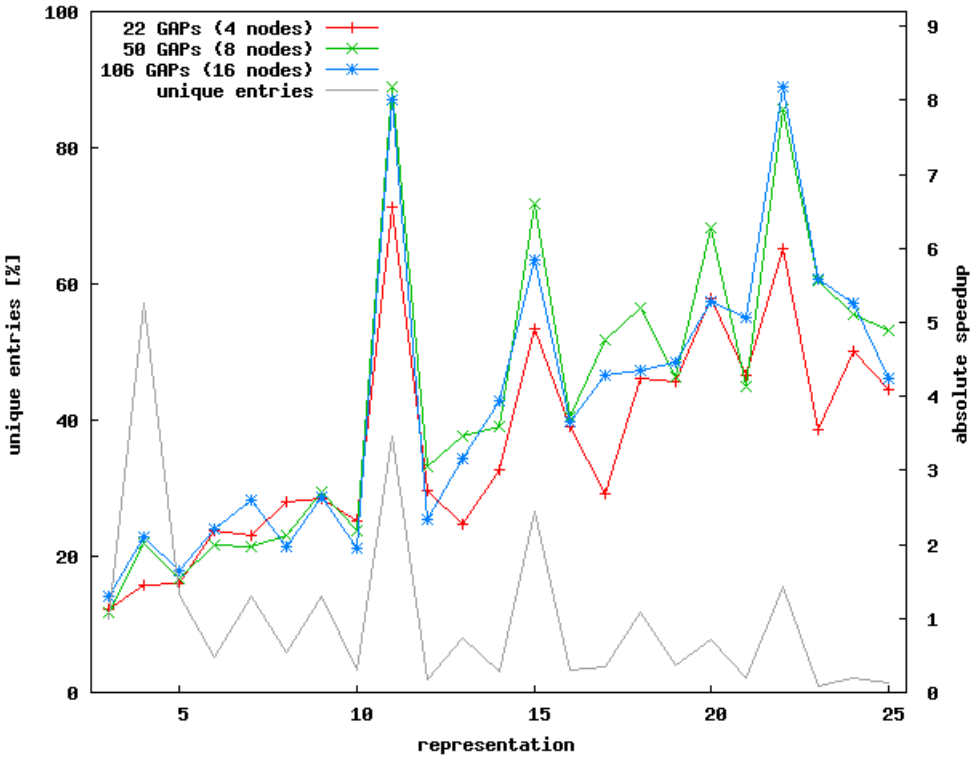

Figure 2: Scaling E_6, speedup on Beowulf cluster.

parallel tasks, which should be a lower bound on the sequential runtime).

We note that $\bar{t}_{\text{red}}$ is proportional to $|V_{lu}|$, and $\bar{t}_{\text{check}}$ is proportional to n^2. Column $\bar{t}_{\text{gen}}$ would be expected to be proportional to n^2, too, yet it also depends on the choice of prime numbers in a non-trivial way. For many representations of E_7, the set of primes less than 255 (which GAP computes with very efficiently) is not sufficient to compute Q. Hence for representations 3 to 38, we start from a set of 80 primes, consisting of all primes below 255 and some larger primes (less than 2^{16}). (How many primes are actually used depends on set of interpolation values because primes that divide some value are excluded.) For these representations, $\bar{t}_{\text{gen}}$ is indeed proportional to n^2.

For representations 39 to 54, we found that choosing 60 primes was sufficient to compute Q. For representations 55 to 60, even fewer primes (40 for rep 56, 33 for all others) are sufficient to compute Q. (Note that these numbers are not unlikely to be tight; we did not systematically search for minimal sets of primes.) Cutting down on the number of primes means generating fewer Q_{pv} matrices and thus saves memory (and pre-processing time) on the master GAP server. We note that reducing the number of primes was crucial for computing representations 59 and 60; without this reduction, the master GAP server would have hit the memory limit of the Beowulf cluster.

Figure 3 summarises tables 2 and 3 into runtime and speedup graphs. The runtime graph also plots the sequential equivalent of the parallel work; it is clearly seen dropping off from rep 39 and then again from rep 54 due to the smaller sets of primes employed. Moreover, the graphs also show that beyond about rep 20 spikes in the fill ratio do not only result in spikes in the parallel work but also in marked spikes in the the parallel runtime. From rep 30, this can be also observed as a marked dent in the speedup. This is due to the fact that the sequential pre-processing time in the *reduce* phase is proportional to the fill ratio, thus countering the speedup from increased parallelism. (Note that speedup here is computed relative to the parallel work since we do not have sequential runtimes for all reps of E_7.)

Figure 4 presents the speedups obtained in a scaling experiment on the Beowulf cluster, scaling representations 23 to 38 from 4 to 8 to 16 nodes. This confirms the above finding of the inverse link between fill ratio and speedup. The figure also shows reasonable scaling (speedups of around 50 with 106 GAP workers) for some of the representations.

rep	n	$\lvert V_{lu}\rvert$	$\#Q_{pv}$	$\#q_{ij}$	t_{106}	$\bar{t}_{\text{gen}}$	$\bar{t}_{\text{red}}$	$\bar{t}_{\text{check}}$
3	7	95	6175	23	94.316	0.014	28.269	0.018
4	7	95	6175	3	90.628	0.008	26.803	0.015
5	15	59	4130	17	14.695	0.020	2.100	0.037
6	15	59	4130	4	15.790	0.018	2.200	0.040
7	21	67	4623	7	23.109	0.022	4.170	0.069
8	21	67	4623	31	23.188	0.029	4.128	0.079
9	21	79	5372	59	43.695	0.031	9.678	0.084
10	21	79	5372	4	40.125	0.022	9.618	0.070
11	27	79	5372	4	41.518	0.031	9.777	0.119
12	27	79	5372	247	64.619	0.045	9.787	0.207
13	35	47	3337	21	12.279	0.057	0.681	0.218
14	35	47	3337	12	12.365	0.056	0.675	0.215
15	35	67	4623	5	26.288	0.056	4.186	0.233
16	35	67	4623	147	34.625	0.064	4.233	0.273
17	56	67	4623	808	96.114	0.182	4.232	1.022
18	56	67	4623	15	38.364	0.144	4.228	0.775
19	70	47	3337	207	37.106	0.263	0.654	1.560
20	70	47	3337	35	32.750	0.278	0.660	1.462
21	84	47	3337	82	46.720	0.438	0.659	2.403
22	84	47	3337	111	47.691	0.441	0.647	2.428
23	105	59	4130	1222	157.434	0.780	2.104	4.831
24	105	59	4130	31	85.342	0.725	2.105	4.084
25	105	55	3905	46	84.650	0.792	1.387	4.397
26	105	55	3905	470	102.479	0.854	1.364	4.690
27	105	55	3905	61	83.680	0.817	1.366	4.374
28	105	55	3905	72	82.103	0.742	1.343	4.316
29	120	59	4130	38	112.831	1.119	2.047	6.093
30	120	59	4130	2947	317.200	1.176	2.107	7.455
31	168	55	3905	76	238.068	2.986	1.382	16.396
32	168	55	3905	3766	523.125	2.953	1.369	18.220
33	189	47	3337	196	280.998	4.141	0.657	22.522
34	189	47	3337	842	324.863	4.172	0.647	23.615
35	189	55	3905	7093	1048.128	4.205	1.384	26.308
36	189	55	3905	92	326.568	3.991	1.391	23.239
37	189	55	3905	1815	472.842	4.158	1.356	24.525
38	189	55	3905	71	308.191	4.122	1.367	22.307

Table 2: E_7 (rep 3 to 38) runtimes and task sizes on Beowulf cluster (16 nodes, 106 GAP workers).

rep	n	$\lvert V_{lu}\rvert$	$\#Q_{pv}$	$\#q_{ij}$	t_{106}	$\bar{t}_{\text{gen}}$	$\bar{t}_{\text{red}}$	$\bar{t}_{\text{check}}$
39	210	55	2805	70	239.208	2.125	1.376	30.451
40	210	55	2805	1869	424.677	2.015	1.367	33.035
41	210	47	2397	795	298.763	2.251	0.648	32.668
42	210	47	2397	831	289.998	2.179	0.647	33.115
43	216	47	2397	2792	463.946	2.537	0.646	36.376
44	216	47	2397	552	287.135	2.458	0.656	35.864
45	280	47	2397	3355	839.829	4.948	0.651	75.988
46	280	47	2397	292	490.255	5.030	0.658	71.961
47	280	47	2397	535	548.570	5.013	0.658	73.603
48	280	47	2397	4120	1043.795	5.203	0.652	76.728
49	315	47	2397	17301	3840.132	6.836	0.654	112.332
50	315	47	2397	1701	865.875	7.075	0.659	106.128
51	336	47	2397	2407	1164.553	7.837	0.646	129.476
52	336	47	2397	709	836.436	7.690	0.650	126.009
53	378	47	2397	11715	3943.044	10.910	0.648	186.758
54	378	47	2397	1065	1240.623	11.202	0.652	180.267
55	405	47	1551	1587	1126.754	4.186	0.661	217.912
56	405	47	1880	18673	5600.334	3.847	0.643	232.453
57	420	47	1551	1454	1183.566	3.026	0.657	242.633
58	420	47	1551	7267	2660.525	4.034	0.655	250.170
59	512	45	1485	32221	13473.779	7.175	0.484	457.091
60	512	45	1485	32221	14796.846	7.354	0.489	451.358

Table 3: E_7 (rep 39 to 60) runtimes and task sizes on Beowulf cluster (16 nodes, 106 GAP workers).

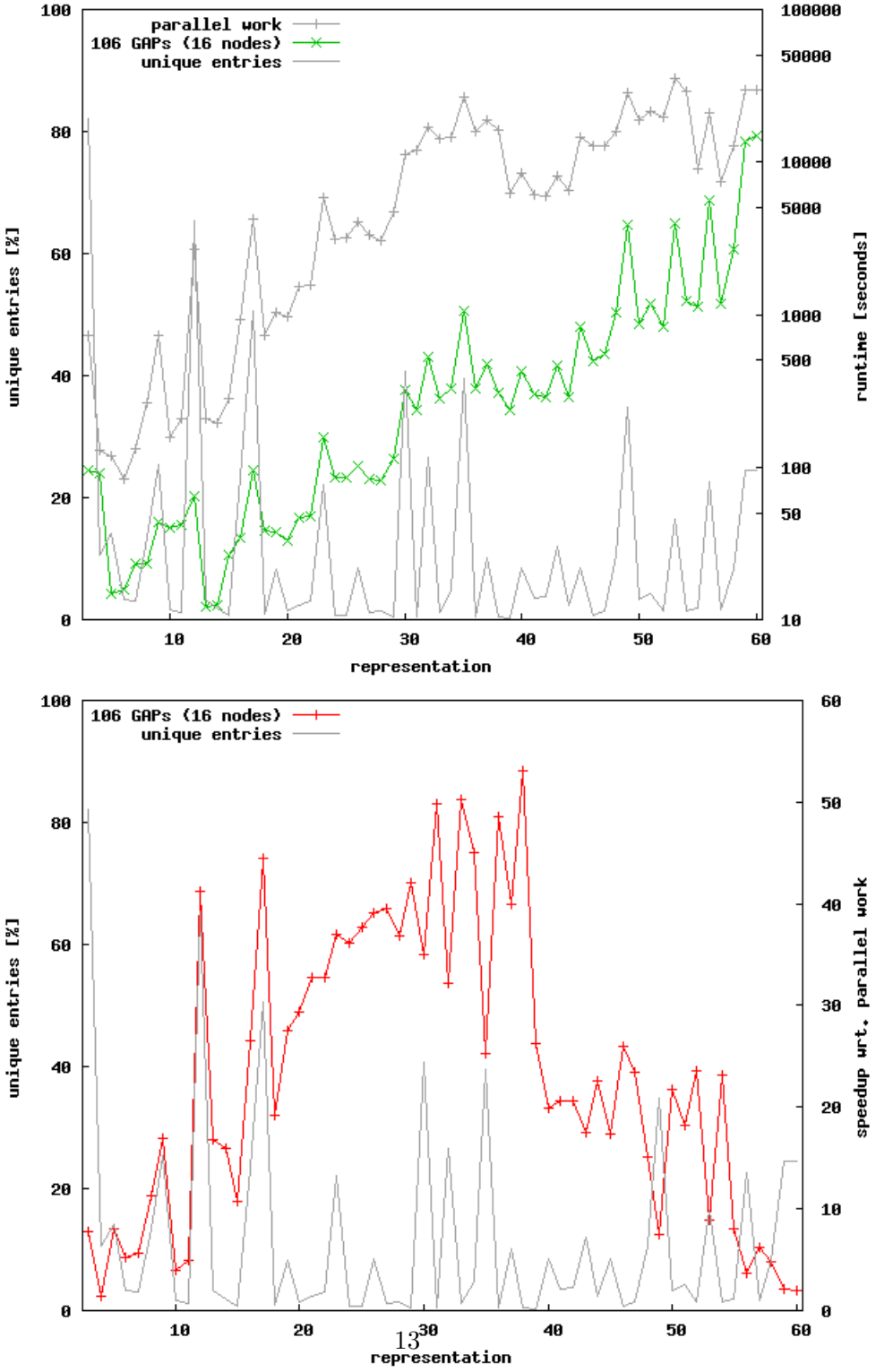

Figure 3: E_7, runtime and estimated speedup on 16 nodes (106 GAP workers) of Beowulf cluster.

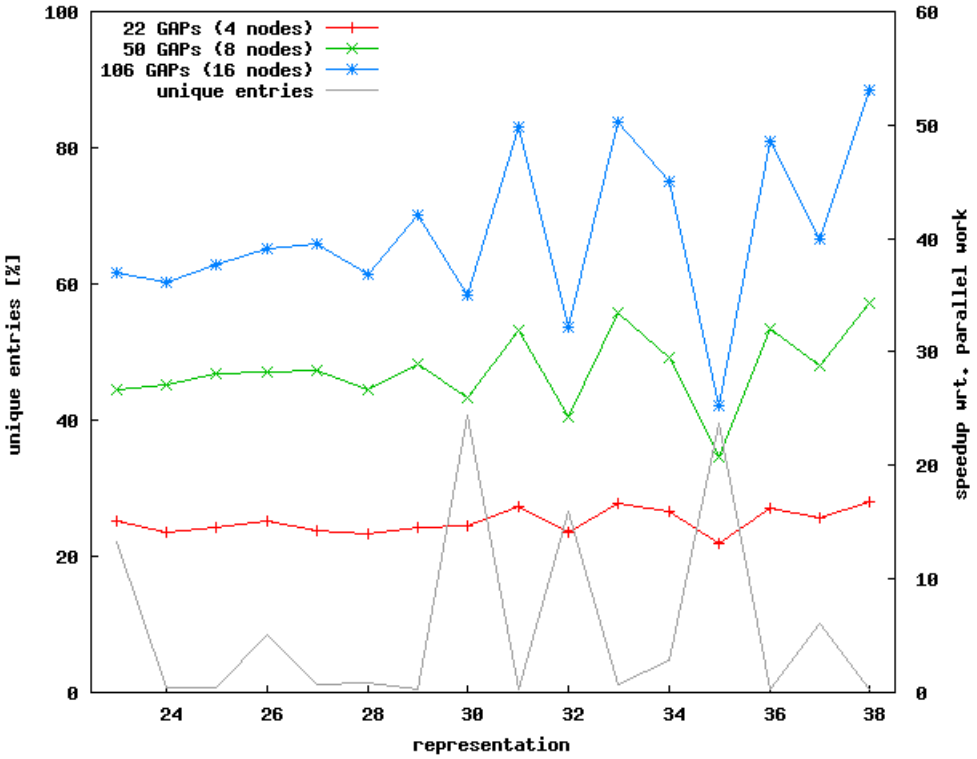

Figure 4: Scaling E_7, reps 23 to 38, speedup on Beowulf cluster.

rep	n	$\lvert V_{lu}\rvert$	$\#Q_{pv}$	$\#q_{ij}$	par_work	t_{94}	t_{218}	t_{466}	t_{962}
31	168	55	3905	76	18462.259	329.253	235.526	202.738	200.627
33	189	47	3337	196	21284.381	378.480	255.478	232.414	234.682
34	189	47	3337	842	24156.627	455.562	332.027	298.904	294.015
36	189	55	3905	92	25411.119	453.064	341.651	297.486	287.775
38	189	55	3905	71	25918.995	432.036	283.559	255.947	251.989

Table 4: Scaling selected reps of E_7 on HECToR: parallel work and parallel runtimes with 94 (4 nodes), 218 (8 nodes), 466 (16 nodes) and 962 (32 nodes) GAP workers.

4.3 Selected E_7 representations on HECToR

To test how far the SGP2 implementation for finding biliear forms scales, we set up a scaling experiment with hand picked representations of E_7. We picked the five reps that showed the biggest speedups on the Beowulf cluster and ran these computations again on 4, 8, 16 and 32 nodes of HECToR, i. e., a strong scaling experiment from 128 to 1024 cores. (Note however that the true limit of parallelism in our experiments is not the number of cores but the number of GAP workers.)

Table 4 shows the runtimes of these experiments, including a column with sum total of all parallel work (which is the baseline for our speedup computations). The figures already show that a scaling limit is hit between 8 and 16 nodes, a finding that is confirmed by the speedup graphs in Figure 5, with limit speedups of around 90 to 100. The reason for this behaviour is the sequential pre-/post-processing in the *reduce* phase.

4.4 E_8 representations 3 to 11 on HWU Beowulf cluster

Table 5 presents data on the first 9 reps of E_8, running on 16 nodes of the Beowulf cluster. While the dimensions of these reps is small, the min/max degree spread is twice as large as for E_7 (except for rep 7). Accordingly, there is more work because the reduce tasks are much larger. Figure 6 shows that those reps with a high fill ratio scale quite well on 16 nodes, reaching speedups of up to 90. However, where the fill ratio is low, there are only a handful parallel reduce tasks, each of which requires several minutes to complete, resulting in very poor speedups for those reps.

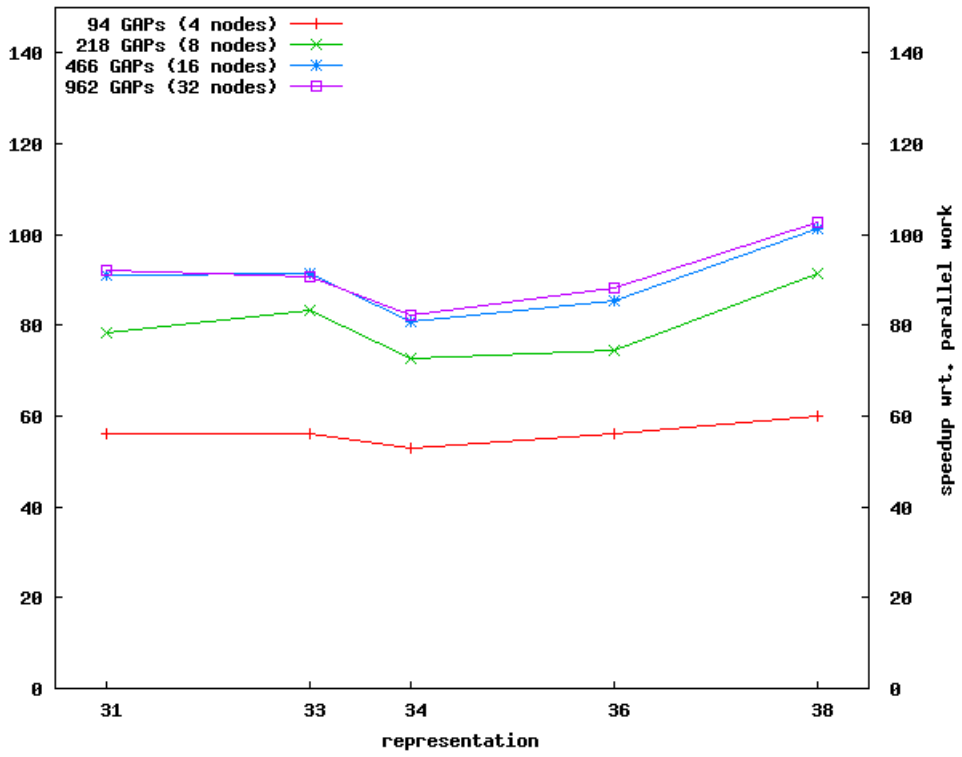

Figure 5: Scaling selected reps of E_7 on HECToR: estimated speedup

rep	n	$\lvert V_{lu}\rvert$	$\#Q_{pv}$	$\#q_{ij}$	t_{106}	$\bar{t}_{\text{gen}}$	$\bar{t}_{\text{red}}$	$\bar{t}_{\text{check}}$
3	28	133	13433	7	516.864	0.051	155.860	0.121
4	28	133	13433	45	539.171	0.063	165.692	0.138
5	35	153	15147	4	1124.900	0.077	354.093	0.208
6	35	153	15147	407	2042.302	0.115	379.535	0.410
7	70	65	7085	21	70.658	0.349	3.548	1.252
8	50	113	11752	5	237.668	0.191	59.512	0.577
9	50	113	11752	172	329.150	0.214	63.334	0.654
10	84	133	13433	8	632.725	0.648	155.602	2.173
11	84	113	13433	2103	4079.103	0.876	166.418	3.226

Table 5: E_8 (rep 3 to 11) runtimes and task sizes on Beowulf cluster (16 nodes, 106 GAP workers).

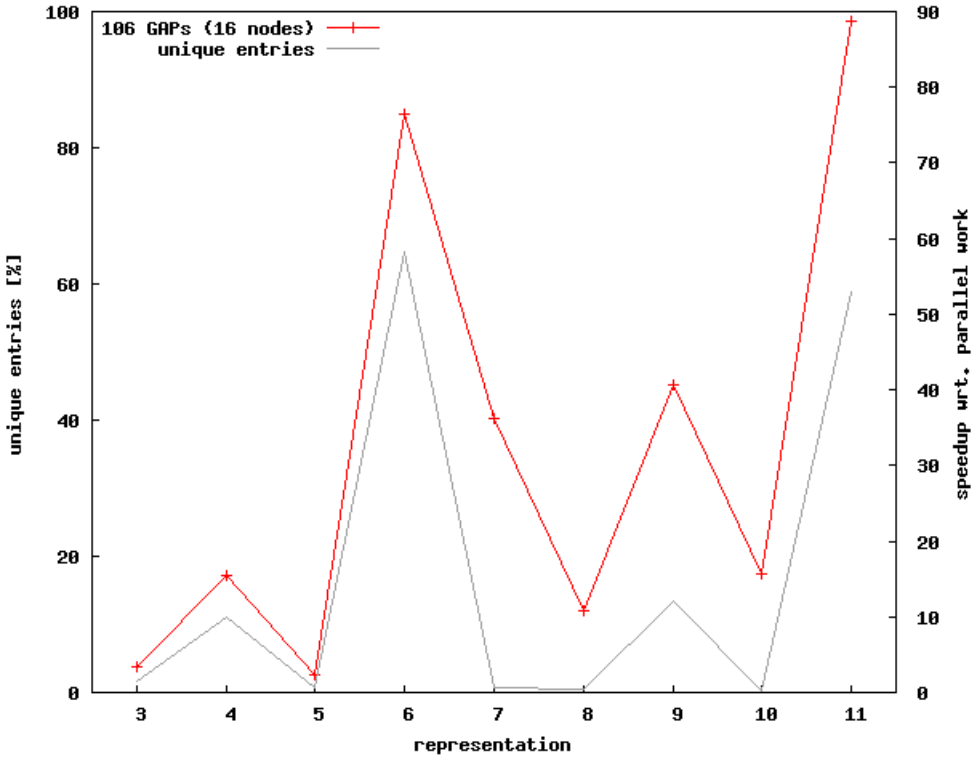

Figure 6: E_8, estimated speedup on 16 nodes (106 GAP workers) of Beowulf cluster.

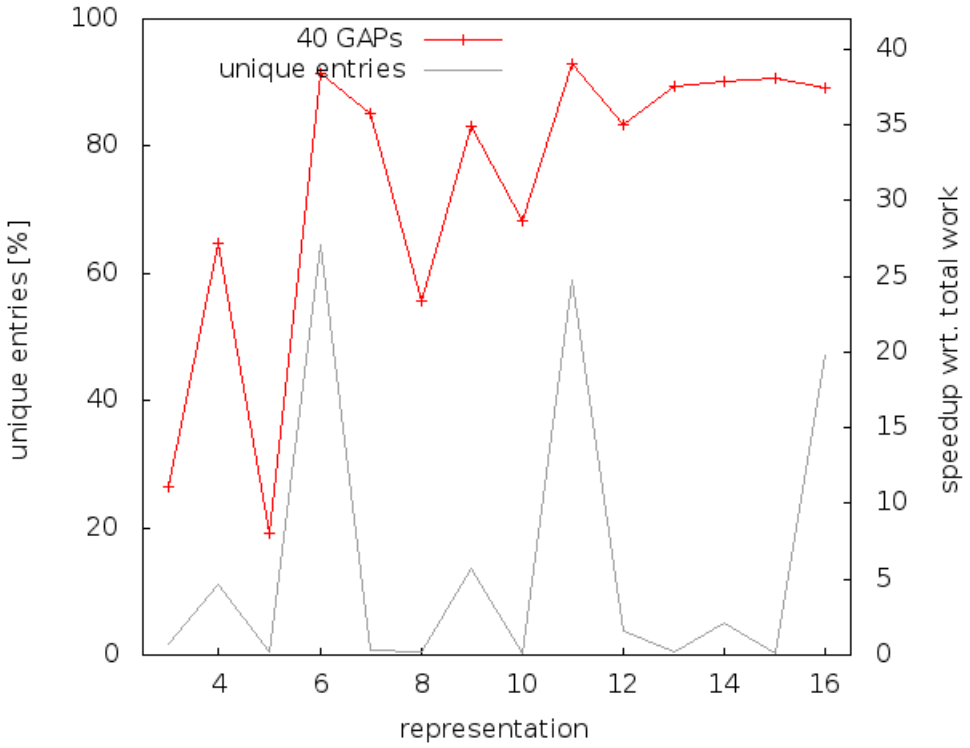

Figure 7: E_8, estimated speedup on Cantor (40 GAP workers).

rep	n	$\lvert V_{lu}\rvert$	$\#Q_{pv}$	$\#q_{ij}$	t_{40}	$\bar{t}_{\text{gen}}$	$\bar{t}_{\text{red}}$	$\bar{t}_{\text{check}}$	t_{seq}
3	28	133	13433	7	187.130	0.076	149.834	0.111	0.874
4	28	133	13433	45	454.898	0.082	250.194	0.132	3.851
5	35	153	15147	4	394.896	0.121	333.995	0.194	1.007
6	35	153	15147	407	6632.009	0.154	619.954	0.408	35.230
7	70	65	7085	21	190.336	0.949	3.478	1.166	2.184
8	50	113	11752	5	155.800	0.288	52.296	0.503	1.341
9	50	113	11752	172	525.910	0.307	85.602	0.602	11.386
10	84	133	13433	8	582.447	1.150	150.699	1.771	4.844
11	84	133	13433	2103	14731.808	1.442	264.075	2.734	185.640
12	168	65	4485	531	701.746	4.956	4.198	14.455	24.617
13	175	81	7047	56	1791.476	9.431	11.808	15.369	15.181
14	175	81	7047	775	3028.959	14.876	12.723	15.276	53.299
15	210	113	11752	43	5056.106	16.097	73.237	24.801	31.255
16	210	113	11752	10433	30168.769	18.176	87.495	32.466	1671.244

Table 6: E_8 (rep 3 to 16) runtimes and task sizes on Cantor (40 GAP workers).

4.5 E_8 representations 3 to 16 on HWU Cantor

Table 6 presents data on reps 3 to 16 of E_8, running on Cantor using 42 cores (40 for GAP workers, one for the GAP master and one for SGP2).[1] The dimensions of these reps are comparable to the middling dimensions of E_8 yet their bigger min/max degree spread causes more work in all phases: in the generate phase because it requires more primes; in the reduce phase because it increases the dimension of the linear systems for determining the coefficients; in the check phase because larger polynomials need to be multiplied and compared. Figure 7 shows that the reps with a high fill ratio or dimension above 100 scale quite well on Cantor, reaching speedups of close to 40.

Note that the last column of Table 6 lists the time spent in the sequential part of the reduce phase. This appears strongly correlated to $\#q_{ij}$, the number of unique entries in Q. While this sequential part isn't much of limit at the scale of Cantor, the E_7 reps have shown that it does limit scalability, and does so increasingly with increasing dimension (since $\#q_{ij}$ is likely to

[1] I am lacking some inputs (the "spinning words") for E_8 reps beyond 16. However, since dimensions rise quickly, I doubt that the current implementation could solve many of the outstanding reps.

rep	n	$\lvert V_{lu}\rvert$	$\#Q_{pv}$	$\#q_{ij}$	total_work	t_{115}	t_{992}
6	35	153	15147	407	1104894.352	13392.621	5308.296
11	84	133	13433	2103	2454855.937	22668.601	4475.932
12	168	65	4485	531	27970.581	398.142	240.351
13	175	81	7047	56	50861.235	679.120	389.389
14	175	81	7047	775	91272.945	1088.743	467.762
15	210	113	11752	43	179139.344	2373.011	1337.154

Table 7: Scaling selected reps of E_8 on HECToR: total work and parallel runtimes with 115 (4 nodes) and 992 (32 nodes) GAP workers.

increase with the dimension).

4.6 Selected E_8 representations on HECToR

Table 7 and Figure 8 show how some selected E_8 reps (those showing best speedup on Cantor) scale on HECToR using 4 nodes (115 GAP workers) and 32 nodes (992 GAP workers). While all of these reps scale well on 4 nodes, all appear to hit scaling limits on 32 nodes. However, rep 11, the one with the most amount of work and the longest runtime, does at least reach a speedup of about 550.

5 Summary

After improving the sequential code, E_6 was too small to be meaningfully parallelised.

We managed to parallelise E_7, reaching speedups of up to 100 on some reps. However, scalability of the largest reps was more limited due to sequential parts in the reduce phase of the algorithm. Memory also became an issue with the largest reps of E_7.

E_8 does show promise in the sense that the problems are so hard that they can scale much further than E_7. However, memory constraints and the remaining sequential parts of the algorithm prevented us from attempting more than the first 14 reps.

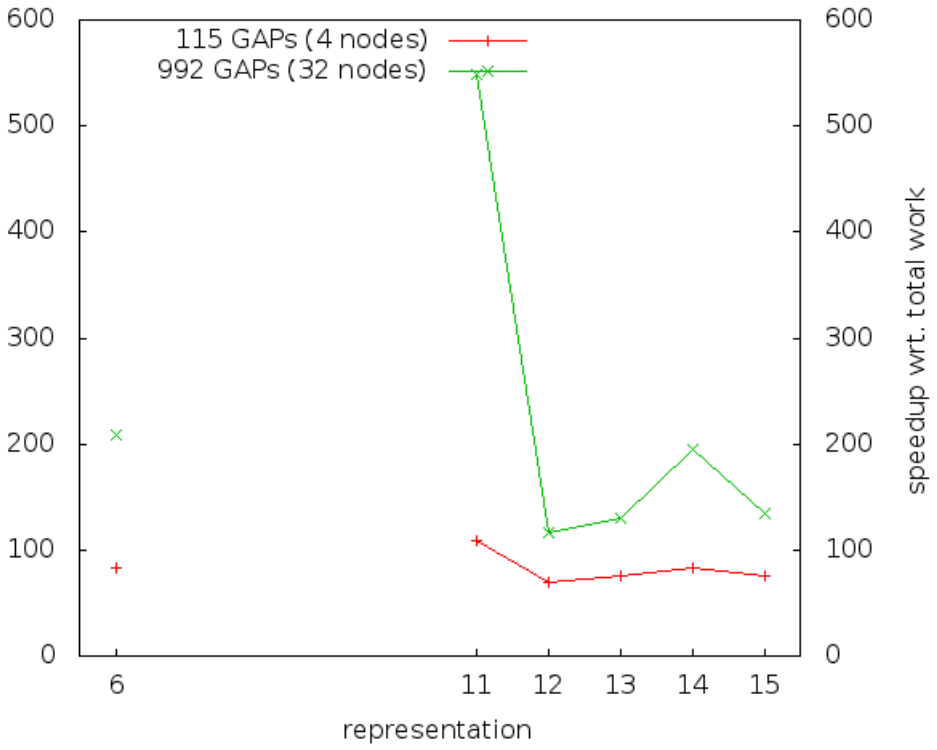

Figure 8: Scaling selected reps of E_8 on HECToR: estimated speedup