Electronic version of an article published as *Parallel Processing Letters, Volume 23, Issue 3, 2013*, Pages 1350010 DOI: 10.1142/S0129626413500102 © World Scientific Publishing Company, http://www.worldscientific.com/toc/ppl/23/03

PARALLEL ALGORITHM FOR SECOND–ORDER RESTRICTED WEAK INTEGER COMPOSITION GENERATION FOR SHARED MEMORY MACHINES

DANIEL R. PAGE*†

Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada R3T 2N2

Published article can be found at http://www.worldscientific.com/doi/abs/10.1142/S0129626413500102

Received December 2012
Revised April 2013
Communicated by I. Stojmenovic

ABSTRACT

In 2012, Page presented a sequential combinatorial generation algorithm for generalized types of restricted weak integer compositions called second–order restricted weak integer compositions. Second–order restricted weak integer compositions cover various types of restricted weak integer compositions of n parts such as integer compositions, bounded compositions, and part–wise integer compositions. In this paper, we present a parallel algorithm that derives from our parallelization of Page's sequential algorithm with a focus on load balancing for shared memory machines.

Keywords: parallel algorithm, combinatorial algorithm, generation algorithm, computational number theory, weak integer compositions, restricted weak integer compositions

1. Introduction

In 2012, Page had introduced a new class of restricted weak integer compositions called second-order restricted weak integer compositions (SORWICs), and an algorithm for their effective generation (see [6]). Restricted types of integer compositions such as SORWICs have a variety of applications in fields such as computational chemistry, and statistical analysis. Let us define SORWICs. For $n \in \mathbb{Z}^+$, $s \in \mathbb{Z}^+ \backslash \{0\}$, a weak n–*composition of* s is an n–sequence $(a_0, a_1, \ldots, a_{n-1})$ of non–negative integers with $\sum_{i=0}^{n-1} a_i = s$. For a family $R_n^2 = (R_0^1, R_1^1, \ldots, R_{n-1}^1)$ of sets of non–negative integers called a *second–order restricted set* , a *SORWIC* $C_{s,n}^{R_n^2}$ is an n–composition of s, $(a_0, a_1, \ldots, a_{n-1})$, where $a_i \in R_i^1$, for all i, $0 \leq i \leq n-1$. For example, given $s = 10$, $n = 4$, and $R_4^2 = (\{0, 1, 2\}, \{3, 6\}, \{1, 3, 5, 6, 7\}, \{1\})$,

*Author Address: Box 4 Group 555 RR5 Winnipeg, Manitoba, Canada R2C 2Z2.
†Email Address: drpage@pagewizardgames.com

$$C_{10,4}^{R_4^2} = \{(2,6,1,1),(0,6,3,1),(1,3,5,1),(0,3,6,1)\}.$$

SORWICs have the potential to be highly output sensitive. Since SORWICs cover most types of restricted compositions or restricted weak compositions, the output can be of size $0 \leq |C_{s,n}^{R_n^2}| \leq \binom{n+s-1}{n-1}$. The upper bound is the number [7] of weak integer compositions (WIC). When the second–order restricted set is totally unrestricted, the SORWIC is a WIC.

The algorithm we will present is designed for the shared memory architecture as we wish to exploit global memory, and make use of threads which do not require much synchronization as we construct the sought SORWIC. Our construction is geared towards making use of only a small number of threads in the system, but these threads are to share the tasks required by the algorithm in a balanced manner.

There is no parallel algorithm yet in the literature which constructs this combinatorial object directly. Other parallel algorithms [1, 3, 5, 8, 9] do exist for less general definitions which are covered by the definition of a SORWIC. Unfortunately, these parallel algorithms would require pruning a potentially large set following execution, which is undesirable. In this paper we will present a parallel algorithm to construct any SORWIC based on Page's sequential algorithm in [6]. To begin, we need to modify the original sequential generation algorithm for effective parallelization.

2. Streamlining the Sequential Algorithm

We would like to modify Page's algorithm for generating second–order restricted weak integer compositions (SORWIC) found in [6] in such a way that we reduce the structure of the sequential algorithm, and exploit data locality. This means we would like to streamline this algorithm so the construction does not rely on s for building the queues. When we design the parallel algorithm, this property could help design a partitioning scheme to balance the load of work since there would be no data dependencies between any of the sequences. Next, we shall show how we can modify the algorithm to suit this property without modifying the computational time–complexity of the original procedure.

In the original sequential algorithm in [6], the algorithm uses a single array of queues $Q[0],\ldots,Q[s-1]$ to store sequences $(a_0, a_1, \ldots, a_{n-1})$ as they are being constructed, and $Q[s]$ to store completed sequences which are a part of the SORWIC $C_{s,n}^{R_n^2}$. Each queue $Q[i]$ contains all the non–negative integer sequences which sum to i in the current round. For example, suppose $s = 20$. A sequence $(1,3,2,0,0,0)$ may not be complete yet if this were $round = 2$ in the algorithm, but is stored in $Q[6]$ for the next induction round where the fourth position would be determined for any copy of this sequence. For any instance the algorithm operates on, there are exactly $(s+1)$ queues, which is not ideal for a large s. This is a data dependency we wish to resolve to streamline the algorithm.

To solve this problem, we know all the sequences in the original algorithm in [6]

are of length n, where n is the number of parts for all the sequences that construct the desired SORWIC. In the algorithm sequences that form the set are represented as integer arrays. Instead of having only n positions, we will increase the length by one where we will store the sum of the sequence of first n non-negative integers. From our previous example, the same sequence could be represented as $(1, 3, 2, 0, 0, 0, 6)$ without needing a specific queue which stores all sequence of sum six. Instead of *entering* sequences into particular queues, place the queue number from the original algorithm into this new position. With such, we remove the data dependency on the parameter s in the algorithm's line of queues, and can be streamlined into three queues for our modified sequential algorithm. These three queues are $Q[0][0]$, $Q[1][0]$, and Q_f.

The first two queues defined are called the *workbench*, and Q_f is the *finished* queue. For the *workbench*, the sequences stored in $Q[(round{+}1)\ mod\ 2][0]$ are copied based on the edge–sets in the original algorithm and *entered* into $Q[round\ mod\ 2][0]$ with our sum update in the final index of each given sequence. If the last index equals s, instead we *enter* the array representing the sequence into the *finished* queue Q_f. Unlike the original algorithm, the base step must also check this last index to see if it may be *entered* into Q_f. The variable $carryTill$ remains the same as in the original algorithm. This variable is used to look ahead to ensure invalid sequences which meets a sum s are not *entered* into Q_f. The variable $carryTill$ must be less than or equal to zero for the base step to *enter* any sequence into Q_f which is not a check needed in the original algorithm. At the end of the algorithm, we output the first n positions of each array before termination in Q_f. The steps of this modified procedure remain identical to the original algorithm except for our modifications.

There is one drawback to our modified sequential algorithm. Since we have removed the need for queues which represent their corresponding sums, we no longer can predict their contents without checking all the sequences. Our modified algorithm does not have optimizations such as the reachability test which can be found in the original algorithm. This is a drawback because for very sparse restricted sets, the final induction round's reachability test deletes all the contents of the queues which never reach $Q[s]$. Next, let us use the streamlined version of Page's algorithm to design a parallel algorithm to generate any SORWIC.

3. Parallelizing the Streamlined Sequential Algorithm

Let us parallelize the streamlined variation of Page's generation algorithm for SORWIC from the previous section. If we are to successfully parallelize this algorithm, then the induction step must have no two threads needing to lock to write to the same place in global memory. Since this algorithm can generate sets at most size $\binom{n+s-1}{n-1}$, critical sections in the algorithm would become a massive bottleneck and degrade the performance. A lock–free construction requiring as little time for synchronization of threads between induction step rounds is vital. This requires a near–optimal load balancing scheme as we construct sequences between threads.

We will show such is possible for any number of threads t. Also, another property we would like to exploit are the loops in the original algorithm that are used in the streamlined procedure. We have two goals in mind for this algorithm. Primarily, we wish to design an algorithm that acheives good absolute speedup with a small number of threads. Lastly, we desire an algorithm which provides near–optimal load balancing so threads are active in the system with as little forking, and joining required by the shared memory machine.

3.1. *Lock–free Induction Step and Load Balancing Scheme*

For our parallel algorithm, we want all the threads to generate sequences independent of one another without any critical sections between induction step rounds. Let us extend the *workbench* so each thread has its own row of t queues. In the streamlined sequential algorithm we had $Q[0][0]$, and $Q[1][0]$ (see Figure 1). Instead of having only two queues, let us extend each second dimension to now represent a $t \times t$ array of queues (see Figure 2). Also, the algorithm will have exactly t *finished* queues $Q_f[0], Q_f[1], \ldots, Q_f[t-1]$ which we will denote as $finished[0], \ldots, finished[t-1]$. Each thread can *enter* valid non–negative integer sequence that sum to s into its own corresponding queue.

$$\left[\ Q[0][0] \ \middle|\ Q[1][0]\ \right]$$

Fig. 1. Modified sequential algorithm *workbench* construction

Now, this leads to two questions. First, how is each row in the *workbench* used in parallel by threads in each induction step round? Furthermore, how does this construction attempt to balance the number of sequences in each row of queues for each thread?

$$\left[\begin{array}{cccc|cccc}
Q[0][0][0] & Q[0][0][1] & \cdots & Q[0][0][t-1] & Q[1][0][0] & Q[1][0][1] & \cdots & Q[1][0][t-1] \\
Q[0][1][0] & Q[0][1][1] & \cdots & Q[0][1][t-1] & Q[1][1][0] & Q[1][1][1] & \cdots & Q[1][1][t-1] \\
\vdots & & \ddots & \vdots & \vdots & & \ddots & \vdots \\
Q[0][t-1][0] & Q[0][t-1][1] & \cdots & Q[0][t-1][t-1] & Q[1][t-1][0] & Q[1][t-1][1] & \cdots & Q[1][t-1][t-1]
\end{array}\right]$$

Fig. 2. Parallel algorithm *workbench* construction

Each thread operates on its own collection of non–negative integer sequences. As we had in the streamlined sequential algorithm, the algorithm switches between the two $t \times t$ arrays in the exact same manner in terms of the rounds in the first dimension. Threads in the induction step will use all the queues in a row based on its identication or *thread_id*. Let us define an integer array of length t called *distribution*. Initially for any thread z, $distribution[z] := z$. For any thread z,

the algorithm assigns $distribution[z] := (distribution[z] + 1)\ mod\ t$ when any new sequence *enters* into a queue. If any thread z generates a valid sequence which sums to s, then it may be *entered* into $finished[z]$.

Our method of balancing the load of sequences for each induction step round is to cyclically allocate sequences along each column z for thread z. Sequences *entering* into the opposing set of queues are placed by the value $distribution[z]$ which represents the row number. In the next induction step round, we operate on the opposing set of queues now along the rows where we have placed all the new incomplete sequences, as before. This technique spreads sequences along rows so threads have sequences to operate on even though a row may not have any sequences initially. Restricted sets can have the potential to be very sparse due to the general nature of this problem. For instance in the base step of this algorithm, sequences are distributed along the first column of the first set of queues using cyclic distribution. So threads initially may not have tasks to complete if $|R_0^1| < t$. This method permits even such cases to spread out sequences as the induction step progresses due to the initial values used for *distribution*.

For any non-negative thread number $z < t$ and queue number $i < t$, every row entry $Q[(round+1)\ mod\ 2][z][i]$ is unique to any $Q[round\ mod\ 2][distribution[z]][z]$. Consequently, every thread z will have its own row to *leave* sequences from t queues, and one column to *enter* sequences into t queues cyclically based on the value $distribution[z]$. Threads can operate independent of one another throughout the entire induction step with only synchronization at the end of each round. Therefore, there is no locking required by any thread in the entire induction step.

3.2. *Exploiting Loops*

As in the original algorithm by Page in [6], the streamlined sequential algorithm makes use of numerous *for* loops. We can parallelize each of these by making use of static block distribution of the iterations. Some loops in the algorithm are not worth parallelizing due to only having a small number of iterations, but others can become very time consuming for larger inputs. These loops include the base step, and the loops contained in the construction step.

In the base step, we require two critical sections to be effectively parallelized. In the base step we are *entering* newly created sequences where the first position is determined along the first column of the first set of queues. We are using $distribution[0]$ to carry this out, which needs to be updated each time. We do not want two threads to attempt to *enter* into *workbench* queue $Q[0][distribution[0]][0]$ at the same time, so we include this in a critical section along with the update to $distribution[0]$. Also, we must consider if $s \in R_0^1$. If we *enter* a sequence which has sum s already, it must be *entered* into the *finished* queues. If such is the case employ a critical section, and *enter* the sequence into $finished[0]$. Following the base step, $distribution[0]$ is reset back to be zero for its use in the induction step.

In the construction step we have two loops we wish to parallelize. These loops

are the construction of a two dimensional linked list for building the edge–sets, and the creation of the edge–sets. Since both of these steps have other loops inside with dependencies we can distribute statically the outermost loop iterations to the threads to complete only. With such let us summarize the parallel algorithm.

4. Parallel Algorithm

Now, we shall describe our derived parallel algorithm[a] as a whole from the previous section, and verify its correctness. In this section we will use Q, and *workbench* synonymously to represent the *workbench* construction.

4.1. *Algorithm*

As inputs the parallel algorithm has four parameters. The first parameter is R_map which is a $n \times (s+1)$ boolean two–dimensional array. Each row v represents membership to position v. When a bit is set to one at column j in row v, then $j \in R_v^1$ or simply j is allowed in position v for any possible valid sequence contained in the sought second–order restricted weak integer composition (SORWIC). For example, if $s = 9$, $n = 4$, and restricted set $R_4^2 = (\{0,1,4,6\},\{1,3,5,8\},\{0,1\},\{0,9,2\})$, then,

$$R\_map := \begin{bmatrix} 1\,1\,0\,0\,1\,0\,1\,0\,0\,0 \\ 0\,1\,0\,1\,0\,1\,0\,0\,1\,0 \\ 1\,1\,0\,0\,0\,0\,0\,0\,0\,0 \\ 0\,0\,1\,0\,0\,0\,0\,0\,0\,1 \end{bmatrix}.$$

Next, the second parameter is s, the sum or order of the SORWIC. The third parameter is n, the number of parts or simply the number of positions allowed in every sequence of the sought SORWIC. Lastly the last parameter, a positive integer *num_threads* is the number of threads to be used by the parallel algorithm.

As in the original sequential algorithm in [6], the parallel algorithm has five sections:

(i) Preliminary construction,
(ii) Trivial cases,
(iii) Base step,
(iv) Construction step,
(v) Induction step.

Now, we describe our parallel algorithm for shared memory machines to generate any sought SORWIC (Figure 3). Each corresponding component of the algorithm is summarized throughout this section and can be referred to in more detail as referred below in Appendix A.

[a]The algorithm is described in object–oriented style pseudo–code with C++ OpenMP compiler specific directives.

```
void SORWICGeneration(bool[][] R_map,int s, int n,int num_threads){
Algorithm 1.1: Preliminaries
Algorithm 1.2: Trivial Cases
//generating beyond two positions is required...
if((n>2)&&!empty){
 Algorithm 1.3: Base Step
 Algorithm 1.4: Construction Step
 Algorithm 1.5: Induction Step
//output the finished queue line (only valid sequences will exist).
#pragma omp single
for(i:=0;i<num_threads;i++){
 while(finished[i].size()>0){
  int[] sequence:=finished[i].front();
  printSequence(sequence,n);//print actual sequence
  free(sequence);
  finished[i].leave();
 }
}
};//SORWICGeneration
```

Fig. 3. Algorithm 1: Parallel algorithm for SORWIC generation

In the preliminary construction (see Figure A1), the algorithm builds the *workbench*, the line of $finished$ queues, *distribution*, and rules out parameter combinations which will yield an empty set for the SORWIC. If $n = 1$, or $n = 2$, then the algorithm carries out each respective trivial case (see Figure A2), and moves directly to outputting the resulting SORWIC. For $n = 1$, we only check if the sum itself can be placed into a single position. When $n = 2$, the algorithm checks all pairs of non–negative integers and observe if such a sequence is valid or not. In either trivial case, the algorithm *enters* any valid sequences into $finished$[0] sequentially in the same manner as the original algorithm in [6]. If $n > 2$, then the procedure begins the base step (refer to Figure A3). In the base step, the algorithm creates our first sequences based on the first row of *R_map*, and places them along the first column in the first set of queues as described in Section 3.1 for the base step. Next, the construction step (see Figure A4) builds $(n-2)$ edge–sets that dictate the possible values a sequence can generate into new copies based on position n of each sequence. Recall the last position n stores the current sum of any given sequence, while positions zero through $(n-1)$ are the actual non–negative integer sequence. Once the edge–sets are constructed, the algorithm in the induction step (refer to Figure A5) inductively builds sequences based on the sum of each the sequence. Sequences *leave* the active queues of the *workbench* and are copied to create new sequences based on the edges. After this, any sequences created are *entered* into the non–active *workbench* queues or $finished$ queues following the technique we described in the previous section (see Section 3.1). For any $round < (n-2)$, for every sequence the procedure carries out the main induction round step (see Figure A6). Otherwise, the algorithm determines the last two positions in the final induction round step (see Figure A7). Finally, after the induction step for $n > 2$, the algorithm outputs the sought SORWIC.

4.2. *Correctness*

In the parallel algorithm, the algorithm shares almost most of its structure from the original algorithm in [6]. One particular significant modification was the use of the *workbench*, and *finished* queues instead of the structure of $(s+1)$ queues representing corresponding sums. For preliminary portions of the algorithm and base case, it is the simple manner of replacing terms or parallelizing loops. The construction step is identical to the original sequential algorithm only with loop parallelism involved. The portion where we see a difference in the algorithm is in the induction step, which is the workhorse of this algorithm. By division of parts, we will focus on showing the correctness by proving the correctness of the induction step. If the induction step is correct, then by composition, the entire algorithm is correct.

Lemma 4.1. *The sequential streamlined generation algorithm we developed from the original generation algorithm by Page in [6] in Section 2 is correct.*

Proof. Since we know the original algorithm is correct by [6], we wish to apply a reduction to our sequential streamlined generation algorithm to show it is also correct.

To replicate the behaviour of the original sequential algorithm we take both queues Q[0][0], and Q[1][0] and sort each by sum. Recall that every sequence has a current sum which can be found in the last position of the integer array storing the sequence. Since the algorithm doesn't take order of sequences into account in its output, we only care about if a sequence is a newly generated sequence or not. Once $Q[0][0]$, and $Q[1][0]$ are sorted, partition each queue into s queues by grouping each respective queue's contents by sum into $Q[0][0]_0, \ldots, Q[0][0]_{s-1}$, and $Q[1][0]_0, \ldots, Q[1][0]_{s-1}$. Each queue will now contain the sequences which belong to originally either $Q[0][0]$, or $Q[1][0]$ by their respective sum. Count the total number of sequences for the $(round+1)$ mod 2 set of queues and store these values in an array. For any given induction step round, link each pair of queues by their sums in a particular order. For the two queues that represent all the sequences that sum currently to i, $Q[(round+1)\ mod\ 2][0]_i$ is *entered* first, and $Q[round\ mod\ 2][0]_i$ is *entered* last. Call this queue $Q[i]$. Now at any given iteration of the induction step, the array that stores the counts of sequences tells the algorithm how many sequences are left to operate on. To be traversed, we can start at $Q[s-1]$, then decrement down to zero inclusively. Lastly, label Q_f as $Q[s]$. This is the same as in the original algorithm in [6]. Therefore, by this reduction, the sequential streamlined generation algorithm we developed from the original is correct. □

Theorem 1. The parallel algorithm for SORWIC generation terminates printing $C_{s,n}^{R_n^2}$.

Proof. For $t \geq 1$, let $A(t)$ be the statement, $A(t)$: For t threads, the parallel algorithm for SORWIC generation terminates printing $C_{s,n}^{R_n^2}$.
Base Step ($t = 1$): The statement A(1) says for one thread the parallel algorithm for SORWIC generates terminates printing the sought SORWIC $C_{s,n}^{R_n^2}$. When we have one thread, this is the streamlined variation of the original algorithm. By Lemma 4.1, A(1) holds.
Base Step ($t = 2$): For $A(2)$ to hold, the parallel generation algorithm must correctly terminate for two threads. For $t = 2$, we have a two 2×2 arrays of queues in the *workbench*. For two threads, thread one operates in any particular round along the queues in row one, and *enters* the results into the queues of column one cyclically unless the sequence sums to s, where the algorithm will enter into $finished[0]$. In a similar manner, thread two operates for any round along row two, and *enters* sequences into the queues of column two of the opposing set of queues, unless the sum is s for a sequence where we *enter* into $finished[1]$. Link all the queues in the first set of queues together in any particular order through *enter* operations. Similarily, do the same for the second set of queues in the *workbench*. Link both finished queues $finished[0]$, and $finished[1]$ in any particular order. Label the three new queues we have constructed $Q[0][0]$,$Q[1][0]$, and Q_f. By Lemma 4.1, $A(2)$ holds.
Inductive Step ($A(k) \rightarrow A(k+1)$): Fix some $k \geq 2$, let $A(k)$ be the statement, $A(k)$: For k threads, the parallel algorithm for SORWIC generation terminates printing $C_{s,n}^{R_n^2}$. Assume $A(k)$ holds. Now we wish to show $A(k+1)$ holds as well. Using the induction hypothesis, since $A(k)$ holds, then for $k+1$ threads, we have an additional row, or column in each set of queues. From $A(k)$ we have constructed $Q[0][0]$, $Q[1][0]$, and Q_f where we have the first k rows and columns entered from each set of queues respectively. In the first set of queues in the *workbench*, in any particular order *enters* the last column of queues, and the last row of queues (excluding the final one, to avoid overlap) into $Q[0][0]$. Do the same with the second set of queues, and *enter* these queues into $Q[1][0]$. We have one additional $finished$ queue, enter $finished[k]$ into Q_f. By Lemma 4.1, $A(k+1)$ holds.

Therefore, by the principle of mathematical induction, for $t \geq 1$, $A(t)$ holds. □

5. Results

We considered two properties in experimentation. These two properties are load balancing, and speedup. All of our experiments were executed on a single node of a shared memory machine called Helium at the University of Manitoba. With sixteen processors, each processor is a Dual Core AMD Opteron Processor 885 with processor speed of 1 GHz. Each processor has a cache of size 1024 KB. Let us first consider load balancing.

5.1. *Load Balancing*

Theoretically, the balance of the number of sequences between threads is the most important property to be met. In our results we used the largest amount of work the algorithm could process, a weak integer composition of n–parts (WIC). Since one can predict the number of sequences in each round, it serves as a useful benchmark. As we have an exact number, we can compare the algorithm's balance of number of sequences per thread v_{alg} by the *ideal* number of sequences processed by each thread v_{per}. To measure this error, we use the *approximation error*. In this circumstance, the approximation error is, $error = \dfrac{v_{alg} - v_{per}}{v_{per}}$.

A case we had selected is when we have a WIC where $n = 12$, $s = 20$, and number of threads is eight. It is important to recall that this combinatorial object can be output sensitive. This test case will result in a set of size 84,672,315 at the end of the algorithm.

Table 1. Load balancing approximation error for a WIC case ($n = 12$, $s = 20$, $t = 8$).

round :	1	2	3	4	5	6	7	8	Final
Thread 1	2.8571	3.8961	1.0728	0.3388	0.3953	0.0414	0.0348	0.0825	0.0110
Thread 2	6.6667	1.2987	0.1920	0.4141	0.0745	0.0827	0.0427	0.0199	0.0185
Thread 3	6.6667	0.2597	1.1858	0.3576	0.0113	0.0730	0.0028	0.0211	0.0137
Thread 4	6.6667	0.2597	0.3727	0.1129	0.2552	0.0304	0.0455	0.0528	0.0392
Thread 5	27.6191	1.2987	1.4342	0.6211	0.0971	0.3150	0.0233	0.0385	0.0066
Thread 6	2.8571	1.8182	0.0113	0.3012	0.2868	0.1630	0.0208	0.0233	0.0003
Thread 7	0.9524	1.2987	2.2699	0.1129	0.1197	0.1423	0.0423	0.0432	0.0256
Thread 8	2.8571	4.4156	1.5246	0.0376	0.0610	0.0110	0.0489	0.0290	0.0137
Average	7.1429	1.8182	1.0079	0.2870	0.1626	0.1073	0.0326	0.0388	0.0161

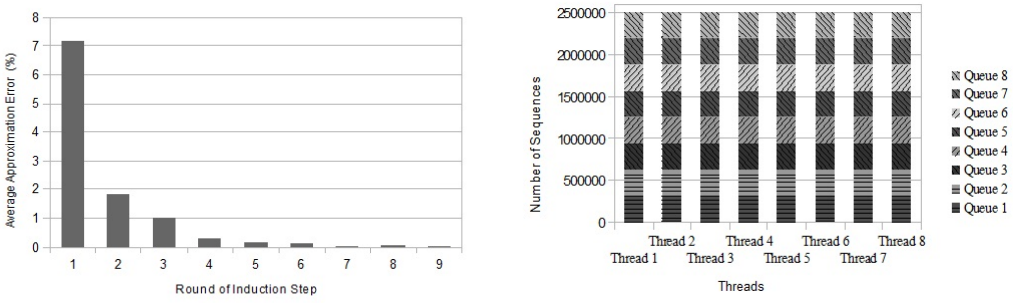

Fig. 4. (Left) The average approximation error as the algorithm processes through each induction step round. (Right) The balance of sequences before the final round of the induction step between threads and queues in the *workbench*.

Table 1 gives all the calculations for every induction step round for our particular instance. On average, as the algorithm progresses, the load balances uniformly across

all threads. From Figure 4, we can see when the algorithm will process its most work that the balance of number of sequences is near–optimal. We observed that on average the load balances already within one percent by $round = 3$ for many instances. Instances where it may take longer to reach near–optimal load balancing between threads are those where we have a very sparse restricted set. These results remained consistent for $s = 40$, and $s = 60$ as well. The *self-balancing* effects caused by cyclically allocating sequences through the *workbench* allows for eventually near–optimal load balancing between threads.

5.2. *Absolute Speedup*

In our experiments, we used three different types of second–order restricted weak integer compositions (SORWIC). We will focus on weak integer compositions (WIC) once again for our experimental results. For SORWICs, a feature to consider carefully is the choice of n, and s. Increasing n or s does not necessarily increase the cardinality of the sought SORWIC. Without this in mind, this may cause sporadic results as the cardinality can change dramatically. So we will present results only for weak integer compositions as these objects monotonically increase in size.

In exploring all our tests, a common effect observed was the rapid saturation of speedup partially due to Amdahl's law when the number of threads is increased to a larger value for a fixed problem instance [2]. Using Gustafson's law [4] to increase speedup, one may need to increase a proper combination of s, n and the number of threads which reflects in our results. We predict saturation and speedup degradation also occurs due to the exponential growth of accesses to global memory which can be costly if the bus is small in the shared memory machine. This is due to our lock–free construction when the number of threads is large, or when the output is very large. Since this algorithm is ideal for a small number of threads, this is not of concern. This algorithm theoretically has absolute speedup of t, the number of threads used by the algorithm as n grows large.

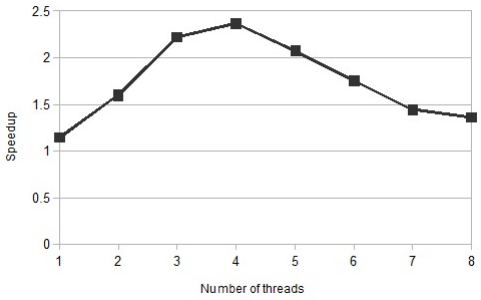

Fig. 5. Our speedup results for generating WIC ($n = 7$,$s = 60$) ($90,858,758$ sequences).

For our WIC tests, we observed $n = 1, \ldots, 7$, $s = 20, 40, 60$, and varied the number of threads between one and eight. For $n = 7$, experimentally we observed peak absolute speedups of 2.21 ($t = 8$), 2.01 ($t = 5$), 2.37 ($t = 4$). For our largest tests, we observed saturation following the peak absolute speedups, but the saturation decreased gradually never below one in our tests. Thus, for a small number of threads this algorithm can outperform the sequential algorithm, and make effective use of all the threads allocated to the algorithm.

6. Conclusions

We have presented a new parallel algorithm for shared memory machines that generates second–order restricted weak integer compositions efficiently for a small number of threads. To design our algorithm, we introduced a modified variation of Page's algorithm found originally in [6] for generating SORWIC that exploits data locality. With such, we introduced a load balancing technique that provides eventually near optimal result, and a lock–free algorithmic construction. Following the design of our algorithm, we proved the correctness of the parallel algorithm, and presented some of our experimental results. Our results show our parallel algorithm can outperform the sequential algorithm for large instances, and can provide an excellent load–balancing scheme. We hope this algorithm can open the exploration of other generalized classes of combinatorial objects where parallel computation has not yet been explored.

Acknowledgements

Thank you Dr. Parimala Thulasiraman for all you have taught me through the graduate course *Advances in Parallel Computing* at the University of Manitoba.

References

[1] S. G. Akl and I. Stojmenovic, Parallel algorithms for generating integer partitions and compositions, *The Journal of Combinatorial Mathematics and Combinatorial Computing*, **13** (1993), 107–120.

[2] G. Amdahl, Validity of the Single Processor Approach to Achieving Large–Scale Computing Capabilities, *AFIPS Conference Proceedings*. **30** (1967) 483–485.

[3] P. Gupta and G. P. Bhattacharjee, Parallel generation of permutations, *The Computer Journal*. **26**(2) (1983), 97–105.

[4] J. L. Gustafson, Reevaluating Amdahl's Law, *Communications of the ACM*. **31**(5) (1988), 532–533.

[5] Z. Kokosinski, Generation of integer compositions on a linear array of processors, *Proceedings of Second International Conference "Parallel and Distributed Processing Techniques and Applications" PDPTA96*. (1996) 56-64.

[6] D. R. Page, Generalized Algorithm for Restricted Weak Composition Generation, *Journal of Mathematical Modelling and Algorithms*. 10.1007/s10852-012-9194-4 (2012) 1–28, Springer Netherlands.

[7] E. M. Reingold, J. Nievergelt, and N. Deo, Combinatorial Algorithms: Theory and Practice, *Prentice-Hall Inc.*. 1977.

[8] H. Shen and J. Evans, A new method for generating integer compositions in parallel, *Parallel Algorithms and Applications.* **9** (1996), 101-109.
[9] I. Stojmenovic, Listing combinatorial objects in parallel, *The International Journal of Parallel, Emergent and Distributed Systems.* **21**(2) (2006), 127–146.

Appendix A Algorithm 1.1 – Algorithm 1.7

All the contents of this section correspond to portions of pseudo–code for a more detailed description of the parallel algorithm given in Section 3.

```
int i,j,k;
i:=0;
j:=0;
k:=0;
//Preliminaries
unsigned long distribution[num_threads];//will hold the weights.
for(i:=0;i<num_threads;i++){
 //Load Balancing initial values
 distribution[i]:=i;//assigning initial allocation scheme
}
bool empty:=0;//assume the set is not empty first
int carryTill =-1;//used to check if we can enter yet into bank of Q_f.
if((num_threads<=0)||(n=0)||(n<0)||((n==0)&&(s>0))){
 empty:=true;//the resulting set is empty.
}
//Assume R_map is binary matrix
queue[][][] workBench:=new queue[2][num_threads][num_threads];//workbench
queue[] finished:=new queue[num_threads];//bank of finished queues
```

Fig. A1. Algorithm 1.1: Preliminary constructions for algorithm

```
//trivial cases for n=1 and n=2...
//if n=1, just check if the sum can be placed in a single position
if(n==1 && !empty){
 if(R_map[0][s]){
  int[] element:=new int[1];
  element[0]:=s;
  finished[0].enter(element);
 }
}
//if n=2, just check the pairs
if(n==2 && !empty){
 for(i:=0;i<=s;i++){
  if(R_map[0][i] && R_map[1][s-i]){
   int[] element:=new int[2];
   element[0]:=i;
   element[1]:=s-i;
   finished[0].enter(element);
  }
 }
}
```

Fig. A2. Algorithm 1.2: Trivial cases for parallel algorithm

```
//carryTill is calculated
//following this round (carryTill), zeroes can be placed in.
for(i:=(n-2);i>0;i--){
 if(R_map[i][0]==0 && carryTill == -1){
  carryTill:=(i-1);
 }
}
//Base Step
#pragma omp parallel for private(j) schedule(static)
for(i:=0;i<=s;i++){
 int[] element:=new int[n+1];
 for(j:=0;j<(n+1);j++){
  element[j]:=0;
 }
 if(R_map[0][i]==1){
  element[0]:=i;
  element[n]:=i;
  if(element[n]<s){
   #pragma omp critical
   {
    workBench[0][distribution[0]][0].enter(element);//enter into workbench
    distribution[0]:=(distribution[0]+1)%num_threads;
   }
  }
  else{
   if(carryTill<=0){//If we allow all zeroes in sequence, enter it!
    #pragma omp critical
    {
     finished[0].enter(element);
    }
   }
  }
 }
}
distribution[0]:=0;
```

Fig. A3. Algorithm 1.3: Base step for parallel algorithm

```
//Construction Step
//2D linked list for building edge-sets
LinkedList[][] linkEdges:=new LinkedList[n-2][s+1];
#pragma omp parallel for private(i,j) schedule(static)
for(k:=1;k<(n-1);k++){
 for(i:=0;i<s;i++){
  for(j:=s-i;j>=0;j--){
   if(R_map[k][j]){
    if((carryTill<=(k-1)) || (i+j != s)){
     linkEdges[k-1][i].insert_front(i+j);//Add to front of the list
    }
   }
  }
 }
}
//now copy edge-sets to a 3D array for constant access
int[][][] edges:=new int[n-2][s+1][s+1];
#pragma omp parallel for private(j,k) schedule(static)
for(i:=0;i<(n-2);i++){
 for(j:=0;j<s;j++){
  int sizeL:=linkEdges[i][j].size();
  for(k:=0;k<sizeL;k++){
   edges[i][j][k]:=linkEdges[i][j].front();
   linkEdges[i][j].delete_front();
  }
 }
}
```

Fig. A4. Algorithm 1.4: Construction step of parallel algorithm

```
int round:=1;//round begins at 1.
//Induction Step...
#pragma omp parallel private(round,i,j,k)
{
 int thread_id:=omp_get_thread_num();
 for(round:=1;round<=(n-2);round++){
  int workIn:=(round+1)%2;
  int workOut:=(round)%2;
  //for each thread, use a row of the work bench.
  for(k:=0;k<num_threads;k++){
   for(int countQDone:=0;countQDone<num_threads;countQDone++){
    //work on each queue in our thread
    int workLeft:=workBench[workIn][thread_id][countQDone].size();
    for(i:=0;i<workLeft;i++){
     int[] element:=workBench[workIn][thread_id][countQDone].front();
     bool endReach:=0;
     workBench[workIn][thread_id][countQDone].leave();
     if(round< (n-2)){
      Algorithm 1.6:  Main round induction
     }
     else{
      Algorithm 1.7:  Final round induction
     }
     free(element);//free the sequence.
    }
   }
  }
  //now synchronize for next induction step round.
  #pragma omp barrier
 }
}
}
```

Fig. A5. Algorithm 1.5: Entire induction step of parallel algorithm

```
//main round induction
      for(j:=0;j<(s+1) && !endReach;j++){
       int value:=edges[round-1][element[n]][j];
       if(value!=-1){
        int[] copy:=copySequence(element,n+1);//copy all (n+1) positions
        copy[round]:=value-copy[n];
        copy[n]:=value;
        if(copy[n]==s){
         finished[thread_id].enter(copy);
        }
        else{
         workBench[workOut][distribution[thread_id]][thread_id].enter(copy);
        }
        distribution[thread_id]:=(distribution[thread_id]+1)%num_threads;
       }
       else{
        endReach:=1;
       }
      }
```

Fig. A6. Algorithm 1.6: Main round induction for parallel algorithm

```
//final round induction
for(int j:=0;j<(s+1) && !endReach;j++){
 int value:=edges[round-1][element[n]][j];
 if(R_map[n-1][s-value]){
  if(value!=-1){
   int[] copy:=copySequence(element,n+1);//copy the sequence and sum
   copy[round]:=value-copy[n];
   copy[n-1]:=s-value;
   copy[n]:=value;
   finished[thread_id].enter(copy);//done
  }
  else{
   endReach:=1;
  }
 }
}
```

Fig. A7. Algorithm 1.7: Final round induction for parallel algorithm