Developing an aCe Solution for Two-Dimensional Strip Packing

John E. Dorband
NASA Goddard Space Flight Center
Greenbelt MD 20771
John.E.Dorband@nasa.gov

Christine L. Mumford
Computer Science Department
Cardiff University, U.K.
C.L.Mumford@cs.cardiff.ac.uk

Pearl Y. Wang
Computer Science Department 4A5
George Mason University
Fairfax, VA 22030 U.S.A
pwang@cs.gmu.edu

Abstract

This paper describes the development of a fine-grained meta-heuristic for solving large strip packing problems with guillotine layouts. aCe, an architecture-adaptive environment, and the aCe C parallel programming language are used to implement a massively parallel genetic simulated annealing (GSA) algorithm. The parallel GSA combines the temperature schedule of simulated annealing with the crossover and mutation operators that are applied to chromosome populations in genetic algorithms. For our problem, chromosomes are normalized postfix expressions that represent guillotine strip packings. Preliminary results for some benchmark data sets are reported and indicate that the parallel GSA method holds promise as a technique for solving the strip packing problem.

1. Introduction

In recent years, PC clusters and networks of workstations have been increasingly utilized as cost-effective alternatives to traditional multiprocessors for solving a broad range of problems, including scientific applications and combinatorial optimization problems. The most popular approaches to programming these cluster systems usually assume a coarse-grained, control parallel programming model where C or Fortran programs are commonly written as SPMD codes that employ the MPI message passing interface.

While this approach has been successfully used to solve a number of challenging applications, MPI programmers are restricted to the control parallel model when designing solutions to their problems and often need to employ techniques that address issues such as load imbalance and/or task termination detection. As a result, systems such as Treadmarks [1] and the UPC [20] model have emerged to support the distributed shared memory paradigm in order to ease the programmer's burden.

In contrast to the shared-memory paradigm, we propose the use of *aCe*, an "architecture-adaptive computing environment" [5, 6] which provides the programmer with both data-parallel and control-parallel paradigms for developing application algorithms. With aCe C, *clusters* can be defined where the elements within a cluster execute common threads of execution while different clusters can execute different threads concurrently. Moreover, aCe is architecture-adaptive because it also provides the capability to design and utilize virtual architectures while hiding the underlying physical architecture from the programmer. (The aCe C compiler used in our work produces executable code that runs on a Beowulf cluster.)

aCe C has been used to teach parallel programming to students as reported in [7]. In addition, a number of test programs, including FFTs, finite element computation, bitonic sorting, and Conway's Game of Life have been coded and tested on single and multiple PC's at the NASA Goddard Space Flight Center, Bucknell University, and George Mason University.

In this paper, we report on our current efforts to develop a fine-grain evolutionary algorithm for solving an optimization problem known as the two-dimensional strip packing problem. The solution method, referred to as a genetic simulated annealing (GSA) approach, has been successfully applied to a number of difficult optimization problems. The GSA combines the temperature schedule that forms the basis of simulated annealing (SA) with the crossover and mutation operations that are applied to chromosome populations in genetic algorithms (GAs). In this case, the popula-

tions represent solutions to the two-dimensional strip packing problem. The aCe paradigm is particularly suited for implementing a parallel version of the GSA algorithm and our preliminary testing has indicated that an aCe GSA algorithm can produce good, and sometimes optimal, solutions.

In the remaining sections of the paper, we summarize some of the features of the aCe C programming language, describe the parallel GSA approach for strip packing and outline its implementation in aCe C, and present the preliminary results we have obtained for some benchmark data sets. We conclude by discussing the work that remains to be completed in order to fully investigate and evaluate our aCe GSA algorithm.

2. The aCe C programming language

As a parallel programming language, aCe C allows programmers to explicitly specify what can be performed concurrently by the means of *threads*. aCe assumes that every algorithm is based on an inherent virtual architecture consisting of one or more *bundles* of threads. While an aCe program is being executed, the same instruction stream is applied to all threads within a bundle. However, an individual thread does not have to execute all the instructions of the bundle's instruction stream. Using a conditional control structure, a thread may not be required to execute all of the instructions. Similarly, different bundles of threads may likely be performing different instruction streams.

aCe C is executed on the basis of thread activations: there is always a `MAIN` thread that is automatically created in every aCe program and threads are also declared globally. Information about executing threads can be obtained through various pre-defined system constants. For example, `$$i` is associated with each thread's unique identifying rank (ID) within its bundle. Further, a thread can communicate with another thread in the same bundle or with one in a different bundle. aCe also provides a feature called *thread alignment* that allows the user to specify the mapping of the threads onto physical processors. Details about thread declaration and usage are discussed in [7] and in the aCe Tutorial [8].

One of the interesting features of aCe is that it allows programmers to specify the communication paths between the threads in a bundle. Many different forms of path descriptions can be used, such as absolute addressing, universal addressing, relative addressing, and reduction addressing. Further, aCe provides the programmer with the ability to define precomputed path descriptions (i.e. virtual topologies) in order to minimize the cost of routing messages at run-time.

Figure 1 illustrates an aCe code segment that predefines the paths from one thread to its neighbors for a grid topology. Each thread has a total of four neighbors, and the code

```
#include <stdio.aHr>

threads A[100][100];

int main() {
  A.{
     int value, total = 0;

     /*define communication paths*/
     path(A) N, S, E, W, NW, NE, SW, SE;
     N = .A[-1][0].;
     S = .A[1][0].;
     W = .A[0][-1].;
     E = .A[0][1].;

     total = @N.value + @S.value + @W.value + @E.value;
   }
}
```

Figure 1. aCe example of a grid topology

```
#include <stdio.aHr>
#include <stdlib.aHr>

threads A[16];

int main() {
   A. {
      int value, sum;
      int X = $$i;

      /* Define neighbors */
      path(A) cube0, cube1, cube2, cube3;
      cube0 = .A[(X^1)-X]. ;
      cube1 = .A[(X^2)-X]. ;
      cube2 = .A[(X^4)-X]. ;
      cube3 = .A[(X^8)-X]. ;

      sum = value + @cube0.value;
      sum = sum  +  @cube1.sum;
      sum = sum  +  @cube2.sum;
      sum = sum  +  @cube3.sum;
   }
   return 0;
}
```

Figure 2. aCe example of a hypercube topology

Rectangle	Width	Height	Rectangle	Width	Height
0	24	16	7	22	26
1	28	16	8	42	44
2	28	16	9	18	70
3	60	14	10	62	26
4	60	14	11	18	48
5	20	28	12	18	48
6	22	26			

Table 1. The BK data set: $n = 13$, $W = 80$, Total area = 11112

segment shows how it sums its neighboring cells' values. Similarly in aCe, the four neighbors of a thread in a 16-node "hypercube" bundle can be determined by using an XOR operation on the thread's ID with the integers 1, 2, 4, and 8. Each thread can then participate in the calculation of the global sum of values in the code segment of Figure 2.

3. The Two–dimensional strip packing problem

We will utilize the data-parallel nature of aCe's threads to implement a parallel genetic simulated annealing solution for a resource allocation problem where a set of rectangles of given height and width are to be placed orthogonally in a fixed width open-ended strip so that no rectangles overlap and the total height of the packing is minimized. The rectangle placements are limited to those which form guillotine layouts, i.e. arrangements where the rectangles can be obtained by using a series of vertical and horizontal edge–to–edge cuts. Figure 3 illustrates an optimal guillotine layout for the benchmark data set shown in Table 1. Here, the number of rectangles is $n = 13$, the strip width is $W = 80$, and the best known minimal-height packing is 140 as shown.

Many researchers have proposed packing heuristics which produce guillotine patterns. For example, classical algorithms such as First-Fit Decreasing Height or Split packing are discussed in [2, 4, 10, 19]. Other recent studies on strip packing [11, 12, 14] have applied genetic algorithms to problems with fewer than 100 rectangles and have reported superior performances for meta-heuristic algorithms over the classical heuristics. In [17], however, it was demonstrated that for very large size data sets, the classical heuristics can frequently produce better solutions than meta-heuristics.

4. The aCe GSA algorithm

We have formulated a parallel genetic simulated annealing (GSA) algorithm in hopes of obtaining better solutions than the classical heuristics for large sized problems. The

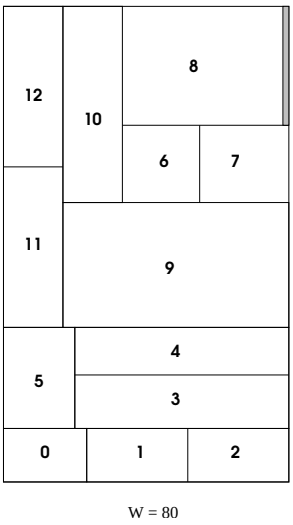


Figure 3. The optimal guillotine packing for the BK data set

GSA approach has been successfully applied to find good, often optimal solutions to a number of difficult optimization problems [3, 13, 15]. In particular, Chen *et al.* formulated a massively parallel GSA for solving the travelling salesperson and error correcting code problems in [3]. By using their format with a genetic encoding scheme that we developed in [17, 21, 22], we have coded an aCe GSA algorithm. As we will see, the preliminary experiments indicate that this solution can obtain good layouts for "small" data sets. However, much work remains to both improve the approach and formally evaluate its effectiveness.

The high-level pseudo-code presented in Figure 4 outlines the data-parallel approach of our GSA algorithm. Using a virtual grid topology, a population of SIZE chromosomes is first generated; each chromosome is associated with an aCe thread. Then, as commonly employed in simulated annealing algorithms, a schedule (i.e. initial temperature and cooling factor α) is determined for *each* thread from the user-specified N (number of iterations) and initial (`P_START`) and final (`P_END`) acceptance probabilities. The parameter (`INIT_TIMES`) sets the maximum number of random trials that are made during the schedule determination procedure. Note that all threads will perform N iterations of the annealing schedule and use the same value of α. Each thread, however, will have *different* starting and ending temperatures as suggested in [3].

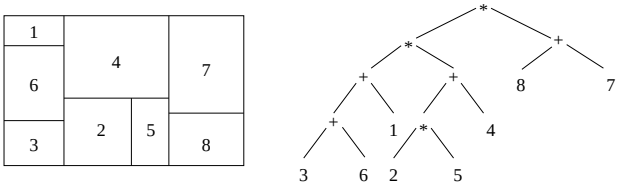
In preparation for the crossover and mutation operations, one thread (GRID[0]) randomly selects one of eight compass directions (N, NE, E, SE, S, SW, W, NW) and a distance between 1 and (`MAX_DIST`). Each thread subsequently receives the chromosome from its thread neighbor in the

```
#include<stdio.aHr>
#include<stdlib.aHr>

/* Total size of the chromosome population */
#define SIZE 1024
#define SQRT_SIZE 32

/* Max distance to neighbor = SQRT_SIZE-1 */
#define MAX_DIST 31

#define INIT_TIMES 5 /* No. times for delta averaging */
#define P_START 0.985         /* Starting probability */
#define P_END  0.0000000001     /* Ending probability */
#define N 3000      /* Number of annealing iterations */

/* Define a grid topology of threads */
threads GRID[SQRT_SIZE][SQRT_SIZE};

int main {

   GRID. {
      int j, direction, distance;
      double temperature, alpha;

      InitializeData();
      CreateSequences();
      InitializeTemperatures(&temperature,
                             INIT_TIMES, &alpha);
      for (j=0; j<N; j++) {

         if ($$i == 0) {
            direction = Random(1,8);
            distance  = Random(1,MAX_DIST);
         }
         direction = GRID[0].direction;
         distance  = GRID[0].distance;

         GetNeighbor(direction,distance);
         Crossover_Mutation();
         Selection(temperature);
         temperature = temperature*alpha;
      }
   }
}
```

Figure 4. Outline of the aCe GSA code

chosen direction and distance. The resident and incoming chromosomes become parents which generate two offspring. A mutation operation may then be performed on the offspring. Next, the `Selection` procedure is applied. An offspring may replace the resident parent if its cost is better; if its cost is worse, it may still replace the parent if the increase in cost is within an acceptance amount determined by using the current temperature. Finally, the annealing temperature is decreased by the cooling factor and the process is repeated until the total number of iterations, N, has been reached.

Guillotine packings for the GSA algorithm are encoded as normalized postfix expressions [17, 21, 22] which utilize the binary operators $+$ and $*$ to represent the placement of one rectangle on top of another ($+$), or one rectangle next to another ($*$). Alternatively, $+$ and $*$ represent horizontal and vertical cuts when viewed from a top-down cutting perspective. *Normalized* postfix expressions are characterized by strings of alternating $*$ and $+$ operators separating the rectangle IDs. Figure 5 illustrates a slicing floorplan, its slicing tree representation, and the corresponding normalized postfix expression.



3 6 + 1 + 2 5 * 4 + * 8 7 + *

Figure 5. Slicing Trees and Normalized Postfix Strings

The chromosome representation stored with each thread is order-based and consists of an array of n records. Each record contains three fields:

- *a rectangle ID field*: this identifies one of the basic rectangles from the set $\{1, 2, 3,..., n\}$
- *an op-type flag*: this boolean flag distinguishes two types of normalized postfix chains, $+ = + * + * + * + \ldots$ and $* = * + * + * + * \ldots$
- *a chain length field*: this field specifies the maximum length of the operator chain consecutive with the rectangle identified in the first field.

From each array of records, a decoder constructs a legal normalized postfix expression. The rectangle IDs in the first field of each record are transcribed in the sequence specified; at the same time chains of alternating operators are

inserted following each ID as specified in the second field of each record (i.e., either $+ * + * + \ldots$ or $* + * + * \ldots$). The length of each individual chain of alternating operators is the value in the third field of each record.

Whenever possible, the chain lengths in the records are adopted. However, if such an adoption produces an illegal postfix expression or partial expression, the decoder corrects it either by adding operators to, or subtracting operators from, an offending chain. Below is an example showing an encoded string and its normalized postfix interpretation:

rectangle 5	rectangle 2	rectangle 4	rectangle 1	rectangle 3
op-type $*$	op-type $+$	op-type $*$	op-type $*$	op-type $+$
length 2	length 1	length 0	length 2	length 0

Postfix expression generated: $5\ 2\ +\ 4\ 1\ *\ +\ 3\ +$

The initial population of postfix expressions is created by randomly generating the records described above for each GRID thread. Rectangle IDs are chosen with uniform probability, as are the op-type flags, while the chain length fields are generated from a Poisson distribution. As the population is generated, postfix strings are rejected if they correspond to packings that are too wide for the strip as dictated by the value of W for the data set. For each thread, strings will continue to be generated until one is found whose width constraint is satisfied.

At each annealing temperature, the aCe GSA code currently performs a *cycle crossover* (CX) [18] on each thread's resident and incoming chromosomes. A single mutation is then applied to each offspring: the mutation is selected from three alternatives: M1, M2 or M3. M1 swaps the position of two rectangles, M2 switches the op-type flag from $+$ to $*$ or vice versa, and M3 increments or decrements (with equal probability) the length field.

In the `Selection` procedure, an improved solution is always accepted and replaces a thread's resident string. To escape any local minimum which may be encountered by accepting only improved solutions, simulated annealing strategies permit the acceptance of poorer solutions. For the aCe GSA algorithm, each solution was measured by its cost, a linear combination of its packing height and packing waste:

$$Cost = \nu * Packing\_height + \rho * Packing\_waste.$$

By adjusting the ν and ρ parameters, it is possible to put more or less emphasis on the relative importance of the packing height versus the packing waste.

5. Preliminary experiments

Our current goal is to establish the viability of the aCe GSA approach. We report on the results we have obtained for five benchmark strip packing data sets that have been used in the literature for comparing meta-heuristic approaches. Some of these data sets have optimal layouts that are not guillotine. Thus, the best reported solutions for those data sets may not be achievable when only guillotine arrangements are permitted. Nevertheless, we found that our GSA algorithm would often obtain reasonable solutions for these data sets.

Three of the data sets used for our initial experiments were obtained from the SICUP website [9] and are referred to as the BK, Dagli, and J1 data sets. Two additional sets, Test1 and Test2, were taken from [15]. The J1, Test1, and Test2 data sets have been used in the literature to test serial GAs and a GSA approach for solving non-guillotine strip packing problems. The characteristics of these data sets are listed in Tables 1 – 5.

The packing heights that our aCe GSA algorithm ob-

Rectangle	Width	Height	Rectangle	Width	Height
0	12	12	11	12	9
1	12	12	12	12	9
2	12	12	13	12	9
3	12	12	14	12	9
4	12	10	15	12	8
5	12	10	16	12	8
6	12	10	17	12	8
7	12	10	18	12	8
8	12	10	19	12	8
9	12	9	20	12	8
10	12	9			

Table 2. The Dagli data set: $n = 21$, $W = 60$, Total area = 2400

Rectangle	Width	Height	Rectangle	Width	Height
0	12	6	13	4	5
1	4	7	14	2	4
2	6	7	15	8	4
3	10	2	16	8	6
4	2	5	17	8	3
5	6	4	18	6	3
6	4	2	19	2	6
7	4	6	20	8	2
8	7	9	21	3	5
9	4	5	22	2	5
10	6	4	23	3	4
11	4	6	24	2	4
12	6	3			

Table 3. The J1 data set: $n = 25$, $W = 40$, Total area = 600

Rectangle	Width	Height	Rectangle	Width	Height
0	200	100	5	100	60
1	100	50	6	60	60
2	100	50	7	40	100
3	100	100	8	160	40
4	100	120	9	200	40

Table 4. The Test1 data set: $n = 10$, $W = 400$, Total area = 80000

Rectangle	Width	Height	Rectangle	Width	Height
0	100	50	8	200	40
1	100	50	9	100	70
2	100	30	10	100	50
3	100	70	11	50	60
4	100	100	12	50	60
5	100	50	13	50	60
6	100	50	14	50	60
7	200	40			

Table 5. The Test2 data set: $n = 15$, $W = 400.0$, Total area = 80000

tained for these tests are displayed in Figures 6–10. As a basis of comparison, we used the same parameter settings for all executions. The only varying factors were the size of the population (i.e. the number of threads) and the number of annealing iterations N. For each test, the postfix expressions in the initial population were generated from a Poisson distribution with mean operator length of 1.5. The schedule determination parameter `INIT_TIMES` was set to 5. The mutations M1 and M3 were applied to each offspring with probability 0.4 and the mutation operator M2 was applied with probability 0.2. For the cost function, the values of $\nu = 2.0$ and $\rho = 0.0375$ were used.

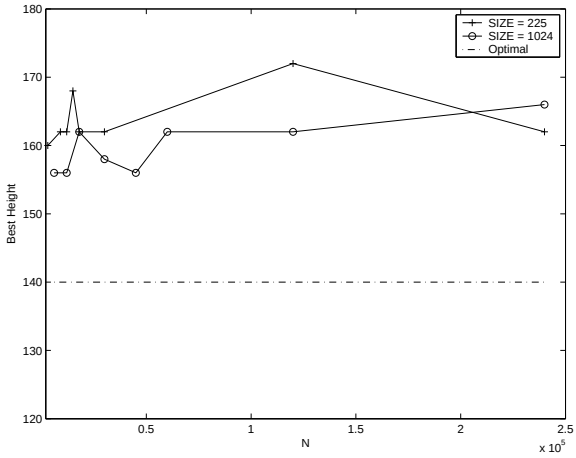


Figure 6. aCe GSA results for the BK data set

For the BK data set shown in Figure 6, the best solution was found using a population size of 1024. This packing height was within 12% of the optimal for the problem, which was reasonable, but not as good as expected. However, for the Dagli data set, the aCe GSA code found the optimal solution using the same population size (see Figure 7).

For the J1 data set, the reported optimal height of 15 is achieved with a non-guillotine layout. When restricted to

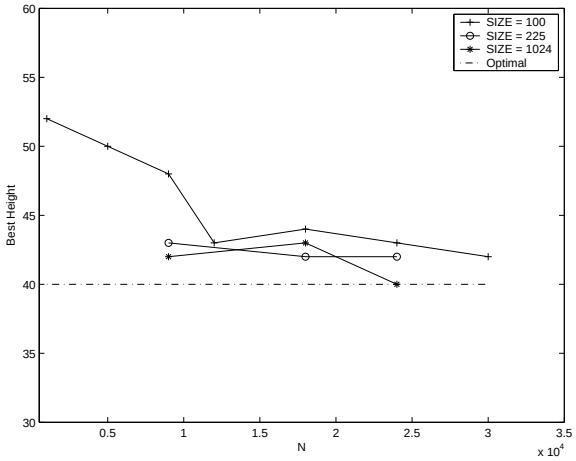


Figure 7. aCe GSA results for the Dagli data set

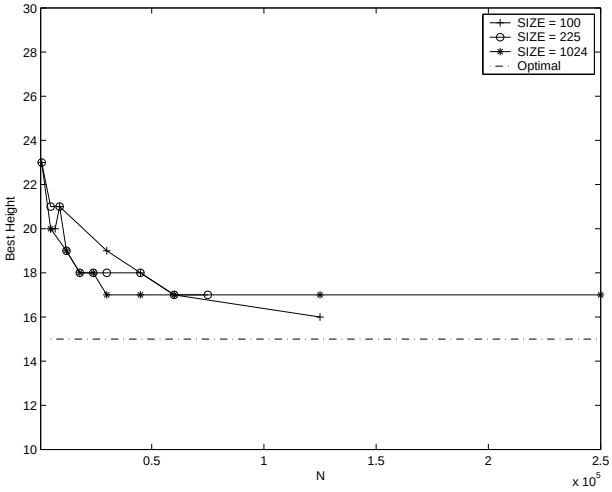


Figure 8. aCe GSA results for the J1 data set

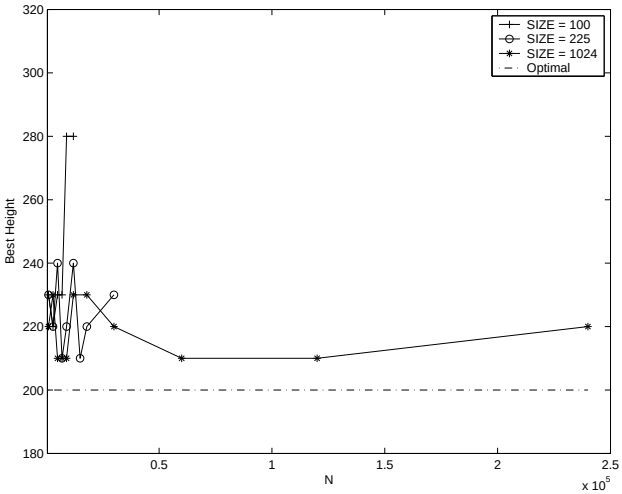


Figure 9. Results for the Test1 data set

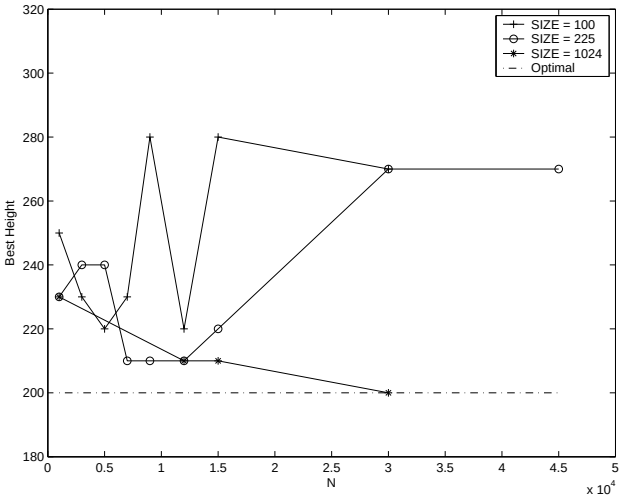


Figure 10. Results for the Test2 data set

guillotine layouts, the least height packing that the GSA obtained was 16 (see Figure 8), a value that was also reported earlier in [16, 17]. It is not known if this is the best possible height under the guillotine restriction. A packing obtained by our GSA with this height is shown in Figure 11.

Finally, for the Test1 and Test2 data sets which contain just $n = 10$ and $n = 15$ rectangles, the aCe GSA algorithm found a solution within 5% of the best possible height for Test1 and was able to discover an optimal solution for the Test2 set as shown in Figures 9 and 10.

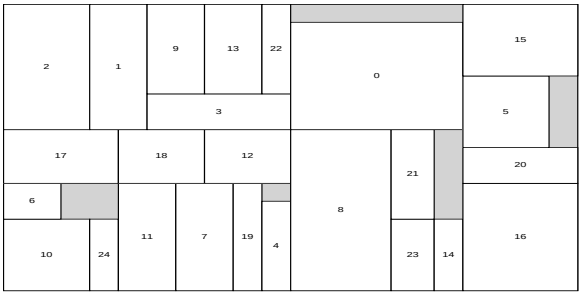


Figure 11. A GSA packing with height 16 and width 40 for the J1 data set

6. Work in progress

While these initial results have been encouraging, they are by no means rigorous enough to argue that the approach is effective and scalable for larger packing problems and population sizes. To accomplish this, we are conducting further studies using a larger range of data sets from the literature, including the very large *nice* and *path* problem sets used in [17]. We continue to examine different settings for the aCe GSA code parameters and their effects on solution quality. Alternative crossover operators may also prove beneficial, as might population reseeding strategies when solutions converge too early.

The aCe GSA algorithm experiments were carried out in virtual mode– that is, the aCe C compiler generated code for a single AMD 1.7 GHz Athlon based workstation. The execution time needed for the reported tests ranged from several seconds to several hours for the larger populations requiring N = 240,000 schedule iterations. We expect the total time to reduce significantly when the parallel aCe compiler is used; we have observed that this is the case for some preliminary tests that we have conducted on a 12-node Beowulf cluster. In the long term, the run-time performance of the method necessarily depends on the aCe C compiler and its underlying data partitioning and communication strategies. These issues are also currently

under study.

Acknowledgements

The authors would like to acknowledge the aCe programming tests performed for this research by Anh-Tuan Tran, a former graduate student at George Mason University.

References

[1] C. Amza, A. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, 1996.

[2] B. S. Baker and J. S. Schwarz. Shelf algorithms for two-dimensional packing problems. *SIAM Journal of Computing*, 12(3):508–525, August 1983.

[3] H. Chen, N. S. Flann, and D. W. Watson. Parallel Genetic Simulated Annealing: A Massively Parallel SIMD Algorithm. *IEEE Transactions on Parallel and Distributed Systems*, 2(9):126–136, 1998.

[4] E. G. Coffman Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing – an updated survey. In G. Ausiello, N. Lucertini, and P. Serafini, editors, *Algorithm Design for Computer Systems Design*, pages 49–106. Springer-Verlag, Vienna, 1984.

[5] J. E. Dorband. *aCe C Language Reference*. NASA Goddard Space Flight Center, Greenbelt MD.

[6] J. E. Dorband. An Architecture-Adaptive Computing Environment. URL: newton.gsfc.nasa.gov/aCe. NASA Goddard Space Flight Center.

[7] J. E. Dorband and M. F. Aburdene. Architecture-adaptive computing environment: A tool for teaching parallel programming. In *Proceedings* 22^{nd} *ASEE/IEEE of Frontiers in Education C onference*. IEEE, 2002.

[8] J. E. Dorband, M. F. Aburdene, and M. McCotter. aCe-ing Parallel Programming. August 2002.

[9] EURO Special Interest Group on Cutting and Packing. URL: www.apdio.pt/sicup/.

[10] I. Golan. Performance bounds for orthogonal oriented two-dimensional packing algorithms. *SIAM Journal of Computing*, 10(3):571–581, August 1981.

[11] E. Hopper and B. C. H. Turton. An empirical investigation of meta-heuristic and heuristic algorithms for a 2d packing problem. *European Journal of Operational Research*, 128:34–57, 2001.

[12] S. M. Hwang, C. Y. Kao, and J. T. Horng. On solving rectangle bin packing problems using genetic algorithms. In *Proceedings of the 1994 IEEE International Conference on Systems, Man and Cybernetics*, pages 1583–1590, 1994.

[13] S. Koakutsu, M. Kang, and W. W.-M. Dai. Genetic Simulated Annealing and Application to Non-slicing Floorplan Design. Technical report, University of California, Santa Cruz, Computer Engineering & Information Sciences, November 1995.

[14] B. Kröger. Guillotineable bin packing: A genetic approach. *European Journal of Operational Research*, 84:645–661, 1995.

[15] T. Leung, C. K. Chan, and M. D. Troutt. Application of a mixed simulated annealing-genetic algorithm heuristic for two-dimensional orthogonal packing problem. *European Journal of Operational Research*, 145:530–542, 2003.

[16] D. Liu and H. Teng. An improved BL-algorithm for genetic algorithm of the orthogonal packing of rectangles. *European Journal of Operational Research*, 112:413–420, 1999.

[17] C. L. Mumford-Valenzuela, J. Vick, and P. Y. Wang. Heuristics for Large Strip Packing Problems with Guillotine Patterns: an Empirical Study. In M. G. Resende and J. P. de Sousa, editors, *Metaheuristics: Computer Decision-Making*, volume 86, pages 501–522. Kluwer Academic Publishers B.V., December 2003.

[18] I. M. Oliver, D. J. Smith, and J. R. C. Holland. A study of permutation crossover operators on the travelling salesman problem. In *Genetic Algorithms and their Applications: Proceedings of the Second International Conference on Genetic Algorithms*, pages 224–230, 1987.

[19] D. D.K.D.B.. Sleator. A 2.5 times optimal algorithm for packing in two dimensions. *Information Procesing Letters*, 10(1):37–40, February 1980.

[20] Unified Parallel C. URL: upc.gwu.edu.

[21] C. L. Valenzuela and P. Y. Wang. A Genetic Algorithm for VLSI Floorplanning. In *Parallel Problem Solving from Nature – PPSN VI*, Lecture Notes in Computer Science 1917, pages 671–680, 2000.

[22] C. L. Valenzuela and P. Y. Wang. VLSI Placement and Area Optimization Using a Genetic Algorithm to Breed Normalized Postfix Expressions. *IEEE Transactions on Evolutionary Computation*, 6(4):390–401, 2002.