Journal of Parallel and Distributed Computing 169 (2022) 242–251

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

www.elsevier.com/locate/jpdc

Asynchronous simulated annealing on the placement problem: A beneficial race condition

Mark Vousden *, Graeme M. Bragg, Andrew D. Brown

University of Southampton, University Road, Southampton, SO17 1BJ, United Kingdom

ARTICLE INFO

Article history:
Received 30 April 2021
Received in revised form 21 January 2022
Accepted 7 July 2022
Available online 18 July 2022

Keywords:
Optimization
High performance computing
Parallel computing
Simulated annealing
Place and route

ABSTRACT

Race conditions, which occur when compute workers do not synchronise correctly, are considered undesirable in parallel computing, as they introduce often-unintended stochastic behaviour. This study presents an asynchronous parallel algorithm with a race condition, and demonstrates that it reaches a superior solution faster than the equivalent synchronous algorithm without the race condition. Specifically, a parallel simulated annealing algorithm that solves a graph mapping problem (placement) is used to explore this. This paper illustrates how problem size and degree of parallelism affects both the collision rate caused by the race condition, and convergence time. The asynchronous approach reaches a superior solution in half the time of the equivalent synchronous approach. The solver presented here can be applied to application deployment in distributed systems, and the concept can be applied to problems solvable by global optimisation methods, where fitness errors can be tolerated in exchange for faster execution.

1. Introduction

Race conditions are thought of as undesirable in parallel computing, as they introduce often-unintended stochastic behaviour. To avoid race conditions, parallel algorithms typically introduce a resource-management mechanism (usually mutexes or locks) to control access to data, but these mechanisms increase execution time. We present a parallel simulated annealing algorithm where compute workers do not synchronise more than is necessary - this approach introduces a race condition. The algorithm exploits the stochastic behaviour the race condition produces, in order to find a superior solution faster. This paper demonstrates that consciously introducing a race condition, and taking advantage of the resulting stochastic behaviour, can yield significant performance benefits for certain parallel compute problems. Specifically, a Simulated Annealing (SA) implementation of a placement problem is considered here.

1.1. Simulated annealing

Simulated annealing [12,4] is an optimisation method, where the state of a system is repeatedly perturbed over time, gradually improving it with respect to some fitness measure. SA uses stochastic behaviour to facilitate global exploration of the state-space. Consequently, perturbations occasionally worsen the SA solution to overcome undesirable local optima in its search for the best solution. While SA performs on a broad class of problems, it is frequently modified for improved performance when some properties of the problem are known [14], though such modification for best performance is not trivial, as the most effective fitness heuristics [21], problem representation and perturbation operators may require unusual tradeoffs [22]. The global-searching nature of SA, its ease of implementation and tuning, along with its near-forty-year provenance, have made SA the subject of considerable research effort across a broad set of academic disciplines [5,16]. Many research domains employ a modified form of SA as part of a multistage hybrid approach, such as in power-grid monitoring (where measurement units must be placed optimally across the grid network) [11], placement in reconfigurable systems [17,20], and software interaction testing [21].

Top-end desktop processors of today host 64 compute cores: a dramatic advancement on the eight-core processors that were novel four years ago. Advancements in parallel computing architectures are of considerable value for SA approaches, as a superior solution can be more quickly identified. Parallel SA either parallelises fitness evaluations (single-trial parallel), or runs each iteration in a separate compute thread with some synchronisation (multiple-trial parallel) [19,8,3,1,18]. The former of these is only appropriate when the fitness function is compute-intensive, and the latter imposes a severe speed limit on the processing of the problem.

* Corresponding author.
E-mail addresses: m.vousden@soton.ac.uk (M. Vousden), gmb@ecs.soton.ac.uk (G.M. Bragg), adb@ecs.soton.ac.uk (A.D. Brown).

https://doi.org/10.1016/j.jpdc.2022.07.001

Motivated by hardware advancements, much of present-day SA research is focused on parallel SA, though certain valuable properties of serial SA difficult to parallelise. As an example, adaptive annealers in parallel can be achieved by restarting each annealer with an initial annealing length and schedule [6]. Determinism is a valuable property of annealers (at the cost of performance) in certain applications, though dependency-checking and collisions can be "repaired" to maintain determinism [13].

Distributed-memory parallelism has been applied to SA [7], though communication costs between the distributed compute workers imposes a significant runtime penalty for problems where the fitness function is cheap to compute, and for problems with a high degree, such as in highly-connected placement problems. More interconnected (large) problems motivate a shared-memory approach, where the state of the "current iteration" is available to all compute workers simultaneously. Shared-memory SA approaches have been explored, but only for small numbers of parallel workers on commodity desktop machines, whereas distributed-memory SA placement approaches are commonly employed. However, recent innovations in desktop compute technology, such as in the AMD Ryzen Threadripper 3990X, have resulted in processors with as many as 64 physical 4.3 GHz (boost) compute cores, with access to over 512 GB of volatile memory. Such massively-parallel computing platforms provide an alternative to distributed-compute architectures, like Graphics Processing Units (GPUs) and coprocessors.

Due to recent innovations, and due to the rise in more highly-connected problems to anneal, there is value in exploring shared-memory *asynchronous* simulated annealing approaches, as they may converge more quickly than their synchronous equivalents. Such an asynchronous approach would consciously introduce a race condition, and consequently data collisions, between compute workers. This approach will only be of value when the consequence of these collisions is small (such as in placement problems), as the effect of these collisions may become indistinguishable from the already-induced stochastic behaviour in SA.

1.2. Placement

The placement problem, fundamentally, is concerned with mapping the elements of one large graph onto another large graph. Examples of industry-relevant placement problems include the core affinity problem, which maps processes onto compute cores in a processor to maximise compute efficiency [2], and the optimisation of placement and routing of logic blocks in a system-on-chip [17,20], though many more exist. Placement is chosen here as a suitable SA target problem, due to its epistatic, nonlinear nature, due to the large interconnected nature of the problem, and due to the frequency with which placement problems are solved with SA in the literature [5,16,11,20]. Placement problems of industrial and academic relevance are all large, motivating the use of parallel computing to find superior placement configurations more quickly.

1.3. Objectives

Motivated by the need to perform shared-memory simulated annealing to solve placement problems, and improvements in architectures supporting shared-memory desktop processing, this paper introduces an *asynchronously-parallel* SA implementation, and explores how it performs on a particular class of placement problems. This implementation is run on a high-end desktop machine to generate results. These results are used to illustrate the trade-offs between asynchronous and synchronous shared-memory SA approaches, including execution time, collision rate, and the overall impact of collisions, as a function of the size of the placement problem, and the number of compute workers.

2. Theory: placement and parallel simulated annealing

2.1. The placement problem

The placement problem considered in this paper attempts to map a multiprocess application onto a distributed computing substrate. This problem is of particular importance given the rise of off-the-shelf high-power distributed compute systems (GPUs, for example), and burgeoning research into more bespoke event-based reconfigurable technologies [10,15]. These enable orders-of-magnitude increases in the size of problems being solved by computers, resulting in orders-of-magnitude savings spanning a variety of problem domains.

The multiprocess application can be represented as a simple graph A (*application graph*, red graph with circle nodes in Fig. 1), where nodes represent discretised behaviours, and where edges represent the interaction between the nodes. For context, the arrangement of nodes could represent, a program to solve a partial-differential equation (e.g. thermal diffusion, aerodynamics), or an model of agents communicating a/synchronously (e.g. computational chemistry, cellular automata).

The distributed computing substrate can also be represented as a simple weighted graph H (*hardware graph*, black graph with square nodes in Fig. 1), where nodes represent units of compute hardware in a distributed compute system (e.g. compute cores), and where edges represent the communication pathways between those nodes. Edges in this hardware graph are assigned a weight, which represents the "cost" of communicating between two compute units (and function as an approximate aggregation of the effects of latency, bandwidth, and contention).

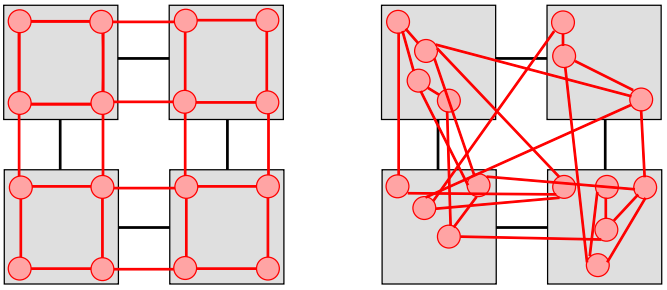
Given a hardware graph and an application graph, the task is to create a many-to-one map of all application nodes to some hardware nodes, such that no hardware node has no more than $N_{\mathrm{A,MAX}}$, application nodes assigned to it. Such a mapping is a *solution* to the placement problem.

Given there are enough nodes in the hardware graph to hold the application graph, a solution can be obtained by mapping application nodes chosen at random into another hardware node chosen at random that is not full (i.e. when the hardware node has less than $N_{\mathrm{A,MAX}}$ application nodes), repeating until there are no more application nodes. However, such a solution is unlikely to perform well in practice, making it undesirable. Fig. 1 shows an optimal (a) and a sub-optimal (b) solution to an example placement problem. Sub-optimal solutions may result in order-of-magnitude slower applications, which are clearly undesirable.

2.1.1. Fitness metric

Fitness allows solutions to be compared: more fit solutions exhibit properties that make them superior to less fit solutions. Fit solutions exhibit:

- **Locality Fitness:** Neighbouring nodes in the application graph are placed as close as possible to each other. This minimises the latency between two communicating nodes. If an application requires two such nodes to synchronise with each other, being close together makes the synchronisation as fast as possible. If the nodes are far away, synchronisation is slow, compromising execution time. Given that the "locality cost" of an application edge is equal to the sum of the weights of the hardware edges it traverses, the locality fitness is minus the sum of the locality costs associated with all application edges (better solutions have a higher fitness).



(a) Optimal placement (b) Random placement

Fig. 1. Examples of placement of a gridded application graph (red circles) onto a gridded hardware graph (black/grey squares), where hardware nodes can contain at most five application nodes ($N_{A,MAX} = 5$). The four hardware edges (black) each have weight w. (a): The application graph is placed optimally onto the hardware - the hardware nodes are evenly loaded, and only eight application edges span multiple hardware nodes. (b): The same application graph placed randomly, resulting in an inefficient placement solution, because the application nodes are unevenly distributed, and cross hardware edges more than is necessary.

- **Clustering Fitness:** The compute hardware is loaded as evenly as possible. Each hardware node has to divide its time between the application nodes mapped to it. An uneven hardware node loading results in some application nodes being executed "faster" than others. For most applications, the slowest-executing node defines the runtime of the application, so an even loading is preferred. Given the "clustering cost" of a hardware node is equal to the square of the number of application nodes mapped to it, the clustering fitness is minus the sum of the clustering costs of all hardware nodes. The square term imposes a greater clustering cost on overloaded hardware nodes.

Note that locality fitness favours a tightly-clustered placement solution, whereas clustering fitness favours a more spread-out solution. To balance these competing fitness contributions, we define the total solution fitness as being the sum between the locality and clustering fitness contributions.

2.1.2. Example fitness computation

To illustrate the conflicting nature of the different fitness contributions, consider the example presented in Fig. 1, which has four hardware nodes connected in a two-by-two arrangement, sixteen application nodes connected in a four-by-four arrangement, and where each hardware edge has the same weight w. A grid placement, such as in Fig. 1 (left), results in locality fitness $-8w$ (as eight edges in the application graph each traverse one edge in the hardware graph), and clustering fitness $-4(4^2) = -64$ (as each of the four hardware nodes are loaded with four application nodes), resulting in a total fitness of $-(8w + 64)$. The random placement in Fig. 1 (right) results in a lower locality fitness $(-16w)$, a lower clustering fitness (-66), and consequently a lower fitness overall $(-(16w + 66))$. Because the random placement has a lower fitness than the grid placement, the grid placement solution is superior to the random placement solution.

An optimal solution to the problem shown in Fig. 1 is evident by inspection, due to the regularity of the problem. However, the optimum solution for similar "grid-like" hardware and application graphs may be irregular. For example, a regular deployment of a grid of application nodes onto a grid of hardware nodes is not optimal if the grids have different aspect ratios, if the application nodes do not divide evenly into the hardware nodes, or if the value of w in the fitness computation strongly favours a tight clustering of application nodes. The value of irregular solutions, coupled with the commonality of irregular application graphs in nature, motivates an iterative numerical approach for finding suitable solutions to the placement problem - inspection cannot be relied upon in the general case, particularly for non-gridded graphs.

2.2. Simulated annealing

SA is a stochastic global optimisation metaheuristic. It is a search method that allows, exploration of a solution space and selection of a guaranteed local optimum, with some concession for global search. The canonical SA algorithm (CSA) is [12,4]:

Initialisation: Define an initial state $s = s_0$ (for iteration $n = 0$).
Selection and Transformation: Select a trial state s_{trial} that is "adjacent" to the current state using an atomic transformation $\delta(s)$. The selected state is chosen at random.
Evaluation: Evaluate the fitness of the selected state $F[s_{\text{trial}}]$, given the fitness of the previous state $F[s]$.
Determination: If the new state is fitter ($F[s_{\text{trial}}] > F[s]$), accept it without further question ($s \leftarrow s_{\text{trial}}$). If not, accept it subject to some random disorder. The effect of this disorder decreases slightly with each iteration.
Termination: If a termination condition has been met (e.g. maximum number of iterations, or some function of the solution), return the most recently accepted solution. Otherwise, loop back to **Selection and Transformation**.

The algorithm loops from **Selection and Transformation** to **Termination** repeatedly - each execution of this loop is an *iteration*. The disorder used in **Determination** is analogous to temperature in traditional annealing, and must decrease monotonically with each iteration, "cooling" the system. A high disorder (or temperature) corresponds to a higher probability of acceptance (exploratory behaviour), whereas a low disorder corresponds to a lower probability (exploitary behaviour). CSA can be used to solve the placement problem introduced at the start of this Section. One approach is outlined in Fig. 2.

2.2.1. Parallel simulated annealing

The literature outlines many traditional approaches to shared-memory-parallel SA [8,1]. Of particular note is the "error algorithm", in which multiple neighbourhood moves are evaluated simultaneously [8]. A variant of this algorithm can be described as follows, where each compute worker performs each non-**Initialisation** step asynchronously:

Initialisation: Define an initial state $s = s_0$. All compute workers can view and modify this state.
Evaluation (1): Evaluate the fitness of the state $F[s]$ before modification.
Selection and Transformation: Modify the state using an atomic transformation $\delta(s)$. This modification acts on the global copy of s, meaning each transformation affects the state visible by all other compute workers.
Evaluation (2): Evaluate the fitness of the transformed state (the new $F[s]$, and compare it to the old fitness computed in **Evaluation (1)** as before. A race condition is introduced by the asynchronously-parallel selection operation.
Determination: As in canonical SA.
Termination: If a worker identifies that the termination condition has been met, that worker communicates with all other workers to stop computation. Otherwise, loop back to **Evaluation (1)**.

The "error" exists when multiple compute workers act on the same region of state at the same time, which results in an erroneous solution fitness, which in turn may affect the outcome of **Determination**. Due to this race condition, implementations of this algorithm

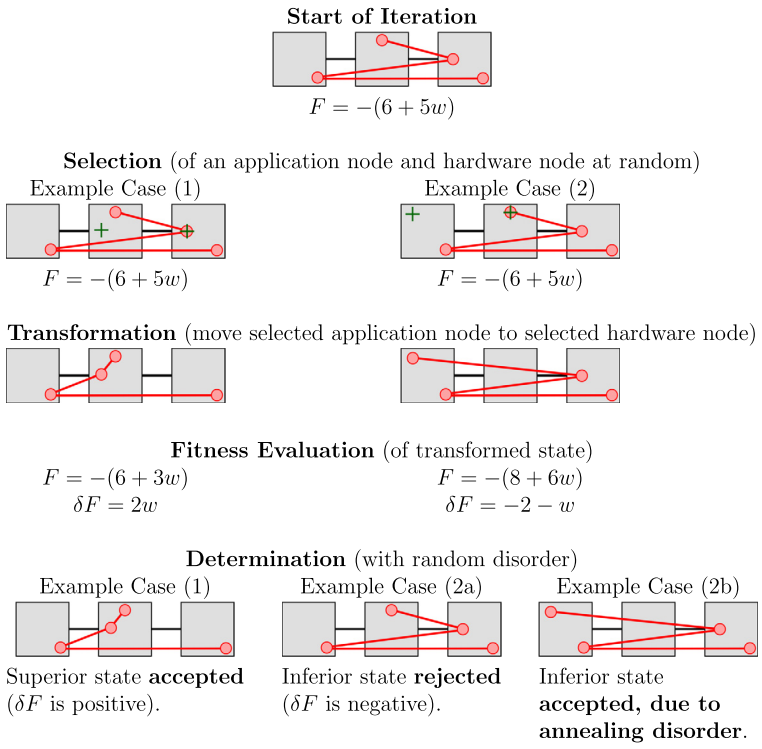


Fig. 2. Three cases demonstrating how the iterative loop of CSA can operate on a placement problem, similar to the one shown in Fig. 1, but with three hardware nodes (black/grey squares) and four application nodes (red circles). Each case demonstrates how a single iteration of CSA can influence the state. New states are found by moving one application node to one hardware mode (**Selection and Transformation**). The fitness value of the new state is computed, which determines whether or not the state is accepted or rejected, subject to a disorder term (explained in the Section 3.2). Case (1) shows an iteration to a more fit state (as F is larger). Case (2) shows an iteration where an inferior state has been selected, where (2a) the state is rejected because it is worse, and another iteration where (2b) the state is accepted, because the disorder of the annealing process has influenced **Determination**.

have a non-deterministic (micro-scale) outcome. However, for certain classes of problems (and we count the aforementioned placement problem among these), and in situations where the problem size is significantly greater than the number of compute workers (reducing the probability of a collision), *we posit that the macro-scale outcome does not significantly change, as fitness improvements will still propagate*.

It is possible to eliminate the aforementioned race condition by introducing synchronisation during the **Selection** operation, to prevent workers from operating on the same region of the problem. By way of example, the placement problem demonstrates spatial locality in its fitness computation, as the change in locality fitness and clustering fitness is only a function of the selected nodes in hardware graph and application graph, and their neighbours. This means it is sufficient to "lock" the selected nodes and their neighbours, preventing them from being selected by other workers, avoiding the race condition. However, this locking mechanism imposes a runtime penalty.

3. Method: design and implementation of annealers

Three SA implementations are compared in this paper, that each operate on the same placement problem: a serial implementation (a performance baseline), a fully-synchronous shared-memory-parallel implementation (that avoids race conditions, at performance cost), and an asynchronous shared-memory-parallel implementation (that introduces a race condition to go as fast as possible). The execution time and final fitness are compared across these three implementations for two sizes of placement problem. The effect of the race condition in the asynchronous implementation is also investigated as a function of placement problem size and the number of compute workers. This Section formally defines these implementations, and the parameters used in this investigation.

3.1. Data structures

Each SA implementation uses a similar data structure, which is designed to minimise the execution time of the iterative loop in SA. This structure contains a set of hardware nodes and application nodes, which *does not change during annealing*. Each application node holds a reference to each of its neighbours in the application graph - these references also *do not change during annealing*. Also, to encapsulate the behaviour of hardware nodes containing application nodes, all hardware nodes hold references to the application nodes that are mapped to them (one to many), and application nodes hold a reference to the hardware node that contains them (one to one). *These references will change during annealing*. These relationships perform well, as lookups are needed in both directions to compute clustering and locality fitness, but introduce an integrity trap: these two elements of the data structure must not contradict at any time. As such, a per-node lock is introduced in the parallel implementations, which allows transformations that affect these nodes to occur in sequence. These locks are used in both the synchronous and the asynchronous implementations to ensure transformations do not compromise the integrity of the datastructure.

The locality fitness calculation requires the sum of the weights of edges connecting two given hardware nodes. Storing weights on a per-edge basis and computing these costs during an **Evaluation** step is computationally expensive. For efficiency, the data structure holds a matrix **W** of weight-sums for each pair of hardware nodes in the problem, which is pre-computed using the Floyd-Warshall algorithm prior to any annealing [9]. This approach significantly

reduces execution time for SA problems that require many iterations to converge satisfactorily.

3.2. Determination: cooling schedule

The **Determination** stage of Canonical Simulated Annealing (CSA) contains a stochastic component to facilitate both explorative and exploitative search behaviours. In this study, the traditional exponentially-decaying disorder parameter

$$T(n) = \frac{n_{\text{end}}}{n} \frac{1}{2\ln(2)} \tag{1}$$

is used, which decays monotonically with increasing n, and where the n_{end}/n co-efficient is arbitrarily chosen to best relax the system from observation. We also use the traditional probability function introduced in [12]:

$$P(\delta F, T(n)) = \exp\left(\frac{\delta F}{T(n)}\right), \quad \text{for } \delta F < 0, \tag{2}$$

where n_{end} is the total number of annealing iterations, P determines the probability with which the trial solution is accepted, and $\delta F = F[s_{\text{trial}}] - F[s]$. Note that T is large at the start of the annealing process ($n \ll n_{\text{end}}$), encouraging exploration of the solution space, and is small at the end ($n \approx n_{\text{end}}$) to encourage exploitation of the basin of attraction. The probability function P is only evaluated when s_{trial} is inferior to s (i.e. negative δF), ensuring that its range is bound between zero and one.

3.3. Atomic transformation and fitness

In each iteration, a trial state s_{trial} is chosen in the **Selection** stage of the CSA algorithm by applying the atomic transformation $\delta(s)$ to the current state. The transformation $\delta(s)$ moves a selected application node to a selected hardware node. This transformation enacts a small change upon the state, and is sufficient to allow the algorithm to explore the state-space of the problem in its entirety. Once selected, the trial state is then compared, using fitness calculation, with the current state s in the **Evaluation** stage. In CSA, evaluation occurs after the state is transformed to the trial state. However, a fitness evaluation of a large placement problem is expensive. Also for large problems, where the state requires a considerable memory footprint, it is unreasonable to store two similar states in memory at the same time, particularly when the transformation only affects a small portion of the state. Furthermore, the **Determination** stage (as shown in the previous Section) only requires the difference in fitness between the two states, and not their absolute value. In this case, the **Selection** and **Evaluation** stages can be efficiently combined by performing some of the comparison prior to selection:

Selection: Select an application node to move, and a hardware node destination (both at random).

Pre-Evaluation: Compute the locality fitness of the application node, the clustering fitness of the selected hardware node, and the clustering fitness of the hardware node that currently contains the selected application node.

Transformation: Move the selected application node to the selected hardware node.

Post-Evaluation: Repeat **Pre-Evaluation**, and use the locality and fitness differences to compute the total fitness change from the transformation.

To proceed, formal definitions for node-specific locality and clustering fitness contributions are required, based on their definitions in the Section 2.1. These definitions consider the simple application graph $A(N_{\text{A}}, E_{\text{A}})$ with nodes N_{A} and edges E_{A}, and the simple weighted hardware graph $H(N_{\text{H}}, E_{\text{H}})$ with nodes N_{H} and edges E_{H} weighted on a per-edge basis. The locality fitness of a state is

$$F_{\text{L}}[s] = -\sum_{j \in E_{\text{A}}} \sum_{i \in M(j)} w_i, \tag{3}$$

where the first sum iterates over each edge in the set of application edges, where $M(j)$ defines the set of hardware edges that application edge j overlays in the state, where w_i is the weight of hardware edge i, and where the second sum iterates over each hardware edge overlaid by j, and adds its weight contribution. The clustering fitness of a state is

$$F_{\text{C}}[s] = -\sum_{k \in N_{\text{H}}} C(k)^2, \tag{4}$$

where the sum considers each hardware node k, and where $C(k)$ denotes the number of application nodes mapped to hardware node k.

Equations (3) and (4) define the locality and clustering fitness contributions of a given state respectively. The "local" clustering fitness (the contribution to the clustering fitness from a single hardware node n_{H}) is:

$$F_{\text{C}}[s, n_{\text{H}}] = -C(n_{\text{H}})^2. \tag{5}$$

Likewise, the "local" locality fitness of an application node n_{A} is:

$$F_{\text{L}}[s, n_{\text{A}}] = -\sum_{j \in E_{n_{\text{A}}}} \sum_{i \in M(j)} w_i, \tag{6}$$

where $E_{n_{\text{A}}}$ denotes the set of edges connected to application node n_{A}.

From the data structure, these local fitness contributions can be computed using only the selected nodes and their properties. This fitness-computation approach allows the fitness of a state to be computed without evaluation of the entire solution - a considerable time saving. Notably, this approach allows compute workers in parallel annealing algorithms to segregate regions of data - essential for synchronisation and for maintaining the integrity of the data structure.

Using Equations (5) and (6), the fitness change δF between two states s and s_{trial}, where the trial state is obtained by moving application node n_{a} from hardware node $n_{\text{H,old}}$ to $n_{\text{H,new}}$ is:

$$\delta F = F[s_{\text{trial}}] - F[s] \tag{7}$$

$$= +\left(F_{\text{C}}[s_{\text{trial}}, n_{\text{H,new}}] + F_{\text{C}}[s_{\text{trial}}, n_{\text{H,old}}] + F_{\text{L}}[s_{\text{trial}}, n_{\text{A}}]\right) - \left(F_{\text{C}}[s, n_{\text{H,new}}] + F_{\text{C}}[s, n_{\text{H,old}}] + F_{\text{L}}[s, n_{\text{A}}]\right). \tag{8}$$

Note that only the selected nodes, and the adjacent application nodes, are used in this fitness computation. This provides a means to compute the change in fitness brought about by the transformation of an existing solution, which is sufficient for the **Determination** stage.

3.4. Serial algorithm (comparator)

Given the data structure, transformation, local fitness computation, and cooling scheduled outlined in this Section, Algorithm 1 defines the serial SA algorithm used as a comparator in this paper.

Algorithm 1 Serial SA algorithm for the placement problem (see text for definitions of variables).

1: **Initialisation:** $n = 0$ (iteration counter).
2: Pre-compute Hardware Weight Matrix $\mathbf{W} : N_\mathrm{H}, N_\mathrm{H} \to \mathbb{R}^+$ from H.
3: Randomly define an initial containment relationship between elements of N_H and N_A such that $|n_\mathrm{H}| < P_\mathrm{MAX}$ for all hardware nodes. This is the initial state.
4: Compute fitness F from the initial state.
5: **Iterate:** $n \leftarrow n + 1$
6: If $n \geq n_\mathrm{end}$, then **end algorithm**.
7: **Selection:** Randomly select application node n_A. Let $n_\mathrm{H,old}$ be the hardware node that contains n_A.
8: Randomly select hardware node $n_\mathrm{H,new}$ such that $n_\mathrm{H,new} \neq n_\mathrm{H,old}$ and $|n_\mathrm{H,new}| < N_\mathrm{A,MAX} - 1$.
9: **Pre-Evaluation:** Compute the F_L contribution of n_A (using $\mathbf{W}$ and N_H), and the F_C contributions of $n_\mathrm{H,old}$ and $n_\mathrm{H,new}$.
10: **Transformation:** Define the container of n_A as $n_\mathrm{H,new}$ via $\delta(s)$.
11: **Post-Evaluation:** Perform **Pre-Evaluation** again to determine the fitness contributions of the selected nodes post-transformation.
12: Compute fitness change from the transformation δF from the fitness contributions obtained in the **Evaluation** steps.
13: **Determination:** If a random number $U(0, 1)$ is greater than $P(\delta F, T(n))$, then define the container of n_A as $n_\mathrm{H,old}$ via $\delta(s)$ (**Revert**). Otherwise, $F \leftarrow F + \delta F$ (**Accept**).
14: Go to **Iterate**.

3.5. Synchronous-parallel algorithm (comparator)

Given the aforementioned serial algorithm, Algorithm 2 is second comparator that anneals in shared-memory parallel. Each compute worker locks a set of nodes each iteration. These locks are claimed as part of **Selection**, preventing race conditions with other compute workers. Also note that, since **Determination** only requires a fitness difference δF, the absolute fitness is free to differ between compute workers - the "true" fitness can be computed by pausing the algorithm and computing the total fitness of the system using Equations (3) and (4).

Algorithm 2 Synchronous parallel SA algorithm for the placement problem (see text for definitions of variables). All data is shared across workers unless otherwise stated. Changes from Algorithm 1 in red.

1: **Initialisation:** $n = 0$ (atomic iteration counter).
2: Pre-compute Hardware Weight Matrix $\mathbf{W} : N_\mathrm{H}, N_\mathrm{H} \to \mathbb{R}^+$ from H.
3: Randomly define an initial containment relationship between elements of N_H and N_A such that $|n_\mathrm{H}| < P_\mathrm{MAX}$ for all hardware nodes.
4: Compute fitness F from the initial state. Each compute worker holds a local copy of F.
5: Spawn all parallel compute workers.
6: **Iterate:** $n \leftarrow n + 1$
7: If $n \geq n_\mathrm{end}$, then **end algorithm**.
8: **Selection:** Randomly select application node n_A. Let $n_\mathrm{H,old}$ be the hardware node that contains n_A.
9: Randomly select hardware node $n_\mathrm{H,new}$ such that $n_\mathrm{H,new} \neq n_\mathrm{H,old}$ and $|n_\mathrm{H,new}| < N_\mathrm{A,MAX} - 1$.
10: **Lock:** Lock n_A, its neighbours, $n_\mathrm{H,new}$, and $n_\mathrm{H,old}$. If any are already locked, unlock all locks claimed in this way, and go to **Selection**.
11: **Pre-Evaluation:** Compute the F_L contribution of n_A (using $\mathbf{W}$ and N_H), and the F_C contributions of $n_\mathrm{H,old}$ and $n_\mathrm{H,new}$.
12: **Transformation:** Define the container of n_A as $n_\mathrm{H,new}$ via $\delta(s)$.
13: **Post-Evaluation:** Perform **Pre-Evaluation** again to determine the fitness contributions of the selected nodes post-transformation.
14: Compute fitness change from the transformation δF from the fitness contributions obtained in the **Evaluation** steps.
15: **Determination:** If a random number $U(0, 1)$ is greater than $P(\delta F, T(n))$, then define the container of n_A as $n_\mathrm{H,old}$ via $\delta(s)$ (**Revert**). Otherwise, $F \leftarrow F + \delta F$ (**Accept**).
16: Unlock all nodes locked in **Lock**.
17: Go to **Iterate**.

3.6. Asynchronous-parallel algorithm

Algorithm 3 describes the asynchronous-parallel SA algorithm for placement, and is used to explore the trade-off between execution time and final solution fitness when the compute workers are permitted to operate on the problem data structure without explicit choreography. The asynchronous-parallel algorithm is similar to the synchronous-parallel variant (Algorithm 2), but does not lock the nodes at **Selection**-time. However, nodes are still locked during **Transformation**, to navigate the integrity trap described in the Section 3.1. Since nodes are locked for fewer instructions, compute workers are blocked at locks for less time each iteration on average, resulting in a faster annealer. This is particularly valuable for placement, as the other stages of the iterative loop are fast (in particular, the fitness delta computation).

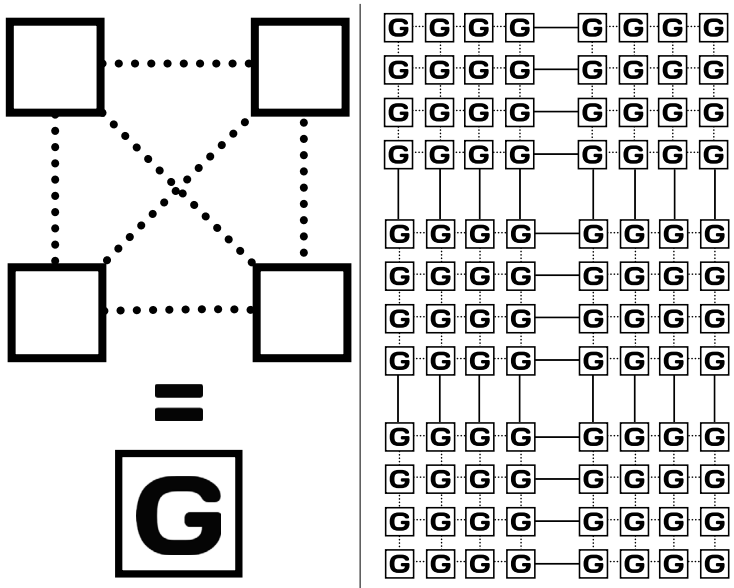
This speed benefit comes at a cost: since a compute worker can now move an application node while another compute worker is performing its fitness computation, Algorithm 3 introduces a race condition. This race condition will cause collisions, resulting in fitness being computed incorrectly during **Post-Evaluation**, which will cause some states being accepted during **Determination** when they would otherwise be rejected (and vice versa).

Algorithm 3 Asynchronous parallel SA algorithm for the placement problem (see text for definitions of variables). All data is shared across workers unless otherwise stated. Changes from Algorithm 1 in red, and from Algorithm 2 in blue.

1: **Initialisation:** $n = 0$ (atomic iteration counter).
2: Pre-compute Hardware Weight Matrix $\mathbf{W} : N_\mathrm{H}, N_\mathrm{H} \to \mathbb{R}^+$ from H.
3: Randomly define an initial containment relationship between elements of N_H and N_A such that $|n_\mathrm{H}| < P_\mathrm{MAX}$ for all hardware nodes.
4: Compute fitness F from the initial state. Each compute worker holds a local copy of F.
5: Spawn all parallel compute workers.
6: **Iterate:** $n \leftarrow n + 1$
7: If $n \geq n_\mathrm{end}$, then **end algorithm**.
8: **Selection:** Randomly select application node n_A. Let $n_\mathrm{H,old}$ be the hardware node that contains n_A.
9: Randomly select hardware node $n_\mathrm{H,new}$ such that $n_\mathrm{H,new} \neq n_\mathrm{H,old}$ and $|n_\mathrm{H,new}| < N_\mathrm{A,MAX} - 1$.
10: **Pre-Evaluation:** Compute the F_L contribution of n_A (using $\mathbf{W}$ and N_H), and the F_C contributions of $n_\mathrm{H,old}$ and $n_\mathrm{H,new}$.
11: **Transformation:** If $|n_\mathrm{H,new}| \geq N_\mathrm{A,MAX} - 1$, go to **Selection**. Define the container of n_A as $n_\mathrm{H,new}$ via $\delta(s)$. While transforming, lock n_A, $n_\mathrm{H,old}$ and $n_\mathrm{H,new}$ to maintain data integrity. If any are already locked, wait until they are unlocked. Nodes are **not** locked outside this step.
12: **Post-Evaluation:** Perform **Pre-Evaluation** again to determine the fitness contributions of the selected nodes post-transformation.
13: Compute fitness change from the transformation δF from the fitness contributions obtained in the **Evaluation** steps.
14: **Determination:** If a random number $U(0, 1)$ is greater than $P(\delta F, T(n))$, then define the container of n_A as $n_\mathrm{H,old}$ via $\delta(s)$ (**Revert**). Otherwise, $F \leftarrow F + \delta F$ (**Accept**).
15: Go to **Iterate**.

3.7. Validation of collision rate (Monte Carlo)

The expected collision rate for a given problem and number of compute workers can be determined by a Monte Carlo experiment, which serves to validate the collision rate of the asynchronous annealer. Under the assumption that all workers perform selection simultaneously, a worker's selection operation will have a collision if either (a) the selected hardware node, (b) the selected application node, (c) the hardware node containing the selected application node, or (d) one of the selected application node's neighbours, is selected by another worker. This process is repeated, until n_end selection operations have been completed - one for each annealer iteration. The aforementioned assumption imposes an upper-bound on the expected collision rate, as the workers in the annealer may not perform selection and fitness computation simultaneously, which avoids some of the collisions identified by this validation method. Validator values are shown with collision rate data presented in the Section 4.



(a) Cores form "core groups" (b) Groups form hardware graph

Fig. 3. (a): How hardware nodes are arranged into a "core group", where dashed edges have weight $W = 0.1$. (b): How "core groups" are arranged into a 8×12 grid to form the hardware graph for the **Small** problem, where dashed edges have weight $W = 100$, and solid edges have weight $W = 800$. The hardware graph for the **Large** problem is a 16×12 arrangement of core groups.

3.8. Parameterisation and implementation details

Implementations of the serial, synchronous-parallel, and asynchronous-parallel algorithms (Algorithms 1, 2, and 3) are run for $n_{\text{end}} = 5 \times 10^9$ iterations, which is sufficient for reaching a local optimum in each of the cases that follow. Each implementation solves two problems, where the application graph is a square Manhattan grid (as in Fig. 1), where the hardware graph is a rectangular Manhattan grid, and where $N_{\text{A,MAX}} = 4096$:

Small: $|N_{\text{A}}| = 1000 \times 1000$, and $|N_{\text{H}}| = 24 \times 16$.
Large: $|N_{\text{A}}| = 1414 \times 1414$, and $|N_{\text{H}}| = 24 \times 32$.

Fig. 3 shows how the hardware nodes are arranged, and the weights assigned to the edges connecting them. The topology of the hardware graph, and the value for $N_{\text{A,MAX}}$, are chosen in line with a contemporary reconfigurable hardware platform [15]. The parallel algorithms are run with varying numbers of compute workers, up to 64. Each run is executed ten times with a same-seed random number generator. Timing begins after initial fitness computation, and ends after the final iteration in the serial implementation, or after all parallel compute workers have despawned, in the case of the parallel implementations. An integrity check is performed before and after each run, verifying that the data structure is not compromised (as described in the Section 3.1). All annealing runs are performed on a single machine with a (multithreaded 32-core) AMD Ryzen Threadripper 3970X processor, and sufficient RAM (250GiB) to contain each problem description. Implementations are created using compiler-optimised C++20 using the POSIX threads parallel execution model, and are optimise-compiled by the GNU Compiler Collection 10.2.1-6. The source for these implementations are available at reference [23]. The Monte Carlo experiments used the validate collision rate data are performed in serial to maintain causality.

4. Results and discussion

4.1. The annealing process in general

Fig. 4 (left) shows how the combination of clustering and locality fitness change as the problem is annealed for the **Large** problem using the serial annealer (solid line). A similar result is obtained using the synchronous parallel annealer (ϒ points). The locality fitness improves as the annealing process "untangles" the initial random state, bringing neighbouring application nodes closer together in the hardware graph. The average hardware weight used by an application edge starts around 2400, and decreases to 78 over the annealing process, resulting in an improved locality fitness. The clustering fitness does not change considerably, as the random initialisation results in a close-to-uniform initial distribution of application nodes across the hardware graph, and the clustering fitness component prevents the annealer from "overloading" hardware nodes with application nodes. The locality fitness does not decrease further due to SA's inability to untangle knots in the mapping that require a (temporary) cost to clustering fitness to resolve, when disorder is low.

4.2. Asynchronous annealer performance

Recall that, with canonical simulated annealing, it is possible for an iteration to move an application node to a hardware node, and create a worse (lower fitness) solution. Also recall that the probability of this happening reduces with more iterations, due to the monotonic decrease in the disorder term. This work posits that collisions from the asynchronous annealer will cause the same behavioural effect (a fitness degradation), but this effect will not decrease with iteration count.

Fig. 4 (left) also shows how the fitness changes as the **Large** problem is relaxed using the asynchronous parallel annealer (• points). The 64-worker asynchronous annealer reaches its optimum ($F = -6.45 \times 10^9$) after around 1.25×10^9 iterations (640 seconds), which is less than that of the 64-worker synchronous

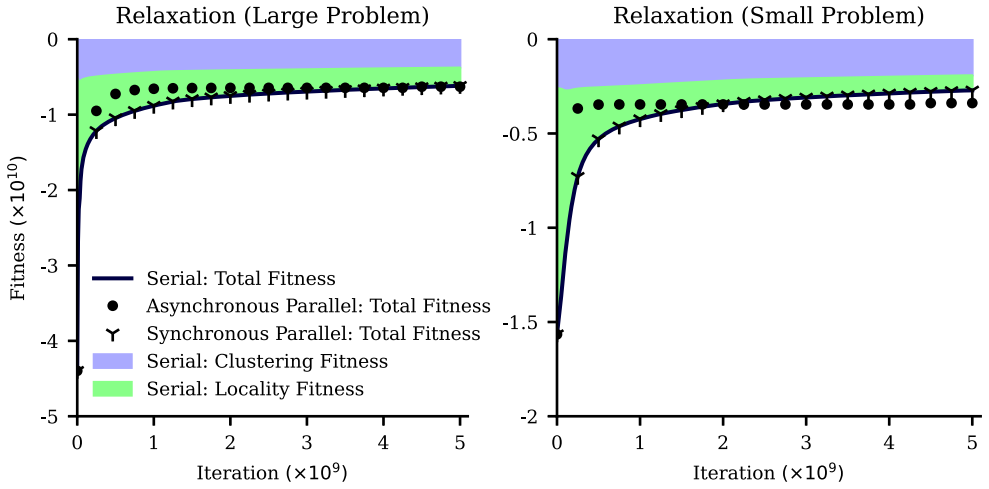


Fig. 4. Total fitness as a function of iteration, for each annealing algorithm. The coloured regions show how each fitness contribution contributes to the total for the serial annealer. The fitness values from the parallel annealers are computed using 64 compute workers (with pauses in the annealing process to compute global fitness), as opposed to the serial annealer where iteration-local fitness changes (δF) are aggregated to determine the total.

annealer, which approaches its optimum ($F = -7.34 \times 10^9$) around 2.25×10^9 iterations (1390 seconds).

Fig. 4 (right) shows the fitness change for the **Small** problem in the same way. As expected, the **Small** problem "relaxes" in fewer iterations. As with the **Large** problem, the asynchronous annealer reaches its optimum faster than the synchronous annealer, though the synchronous annealer finds a better solution given more time.

Each point in Fig. 4 is averaged over ten runs, with a peak fitness deviation of 9.16×10^8 at iteration 5×10^8 for the **Large** problem, and 1.85×10^8 at the same iteration for the **Small** problem. This small deviation demonstrates the repeatability of the asynchronous annealer.

Given that the race condition introduced by the asynchronous algorithm causes solutions to be mistakenly accepted or rejected regardless of their superiority, and given that superior solutions are easier to identify early on in the annealing process, the overshoot of the asynchronous algorithm could be attributed to the higher proportion of superior solutions being accepted compared to the other annealers. This effect is observably similar to a disorder that decays quickly, but also one that doesn't commit the annealer to exploiting a sub-optimal basin of attraction, due to the anisotropic nature of the race condition (it does not implicitly favour superior or inferior solutions, nor acceptance over rejection). This causes the asynchronous annealer to converge more quickly.

As iteration count increases, the fitness of the parallel-annealed solution begins to worsen. Since the effect of the race condition exists independently of the iteration count, and since trial solutions towards the end of the annealing process will typically be inferior, these more-frequent inferior solutions gradually get accepted over superior solutions, worsening the total fitness. Note that the compute workers have no way to detect this without synchronisation; they would see only gradual fitness improvements. This effect is relatively minor, but demonstrates that the induced race condition is not an absolute benefit.

4.3. Collision rate

Fig. 5 shows how frequent collisions are using the asynchronous annealer, alongside the model presented in the Section 3.7. Recall that a worker's selection operation will have a collision if either (a) the selected hardware node, (b) the selected application node, (c) the hardware node containing the selected application node, or (d) one of the selected application node's neighbours, is selected by another worker. As the ratio of problem size to number of compute workers decreases, collision rate increases because there is a higher likelihood of two workers selecting the same node in either of the two graphs. For the problems explored in this study, the chance of a hardware-node collision is significantly greater than that of an application-node collision because, for both problems, there are significantly more hardware nodes than application nodes, and because application nodes have only four neighbours at most.

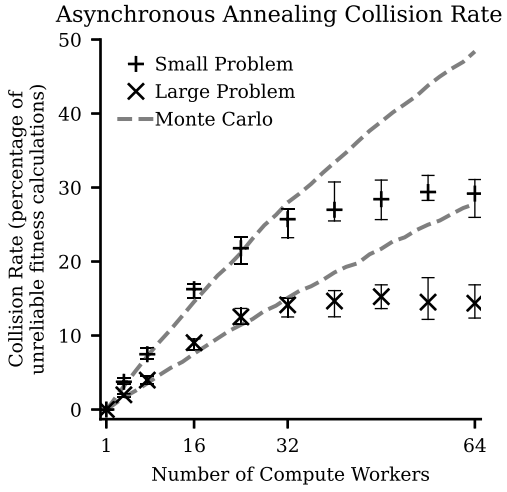


Fig. 5. Cumulative number of collisions for the asynchronous SA implementation as a function of the number of compute workers for both problem sizes. Whiskers show deviation for each point over ten runs.

A greater collision rate (caused by having more compute workers, or a smaller problem) results in convergence in fewer iterations, but also a more rapid solution decay beyond that point. The overshoot effect shown in Fig. 4 is heavily dependent on the collision rate, with a fitness trajectory matching the serial and synchronous cases when only one compute worker is present. The asynchronous annealer has a higher collision rate when applied to the **Small** problem - Fig. 4 (right) shows the exacerbated solution decline relative to the synchronous case, but also the more rapid convergence. For both problems, the collision rate levels out after 32 compute workers as the processor becomes oversubscribed.

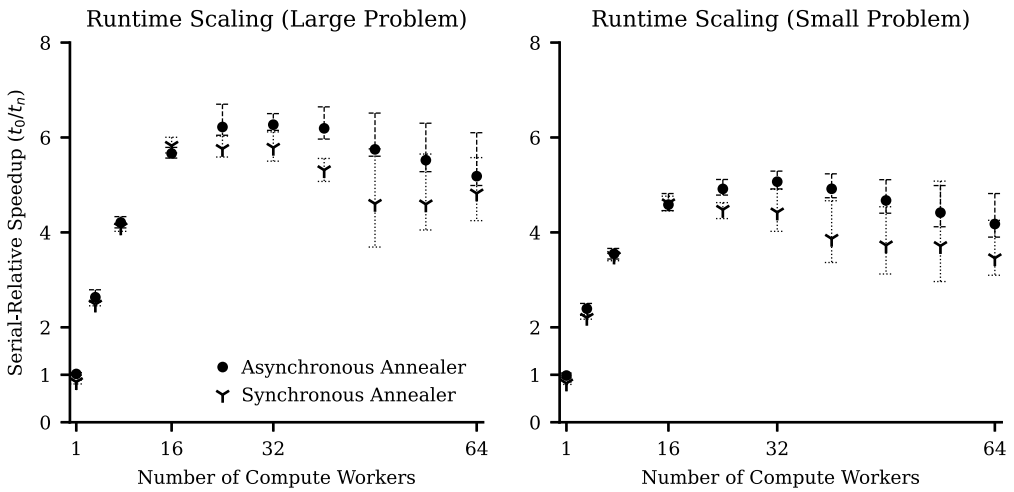


Fig. 6. Speedup relative to the serial SA implementation for the synchronous and asynchronous parallel SA implementations, as a function of the number of compute workers. Whiskers show deviation for each point over ten runs. Measurements taken with no pauses in the annealing process to record intermediate fitness values, nor with any collision-detection logic.

4.4. Speedup and convergence comparisons

Fig. 6 shows the serial-relative speedup $S(n) = t_0/t_n$ of both parallel annealers as a function of the number of compute workers for both problems, where t_0 and t_n are the execution times of the serial annealer and the n-worker-parallel annealer respectively to n_{end} iterations. Performance decays at the 32-compute worker mark and becomes less repeatable, as the processor only contains 32 physical (64 logical), multithreaded cores. The asynchronous annealer iterates slightly faster at the key 32-compute worker point; this demonstrates the potential gain from this method, especially considering that it converges at an earlier iteration than the synchronous method.

5. Closing comments

Simulated annealing approaches are frequently used to solve high-dimensional, large problems across multiple priority research areas. Placement is one such problem where even minor performance improvements yield a drastic impact on the time taken to, for example, deploy an application onto a distributed compute platform, or to arrange logic blocks in a chip. As such, it has become a popular target for parallel simulated annealing approaches.

This paper investigates a technique that delivers a performance improvement over traditional shared-memory parallel annealing: allowing the workers to operate almost asynchronously on the problem, regardless of what the other workers are doing. This approach consciously introduces a race condition, and reduces execution time as workers no longer contend for shared resources. The study presented here explains how such an approach may be implemented. It also demonstrates that, for the largest problem considered here, the race condition introduced by the asynchronous annealer produces a superior solution in half the time of the synchronous annealer, but also that the asynchronous annealer will actually worsen the solution as more time is spent annealing the problem. This effect is exacerbated by decreasing the problem size and by increasing number of compute workers, as the collision rate increases as a result. The result presented here has profound implications for stiff problems, where orders-of-magnitude performance improvements can be realised. Further studies will illustrate how this method can be applied to problems as a function of problem stiffness (here, graph degree), as it would inform the design of future global search algorithms. Future work may also explore how this asynchronous annealing technique applies to other problems studied with simulated annealing in the literature, such as labelling problems [22]. Beyond this, the idea of introducing a beneficial race-condition in this way can be applied to other commonly-used optimisation solving methods, like evolutionary algorithms and particle swarm optimisation, with significant performance improvements.

CRediT authorship contribution statement

Mark Vousden: Conceptualization, Data Curation, Investigation, Methodology, Software, Verification, Visualization, Writing - Original Draft.

Graeme M. Bragg: Investigation, Resources, Writing - Review & Editing.

Andrew D. Brown: Funding Acquisition, Investigation, Methodology, Writing - Review & Editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We acknowledge financial support from EPSRC (EP/N031768/1).

References

[1] E. Aarts, J. Korst, Simulated Annealing and Boltzmann Machines, John Wiley and Sons Inc., New York, NY, 1988.

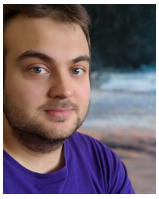
[2] S.R. Alam, R.F. Barrett, J.A. Kuehn, P.C. Roth, J.S. Vetter, Characterization of scientific workloads on systems with multi-core processors, in: 2006 IEEE International Symposium on Workload Characterization, IEEE, 2006, pp. 225–236, https://ieeexplore.ieee.org/abstract/document/4086151.

[3] A. Casotto, F. Romeo, A. Sangiovanni-Vincentelli, A parallel simulated annealing algorithm for the placement of macro-cells, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 6 (5) (1987) 838–847, https://doi.org/10.1109/TCAD.1987.1270327, https://ieeexplore.ieee.org/abstract/document/1270327.

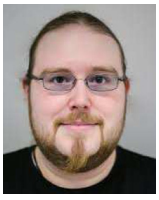
[4] V. Černỳ, Thermodynamical approach to the traveling salesman problem: an efficient simulation algorithm, J. Optim. Theory Appl. 45 (1) (1985) 41–51, https://doi.org/10.1007/BF00940812, https://link.springer.com/article/10.1007/BF00940812.

[5] C. Chen, L.K. Tiong, Using queuing theory and simulated annealing to design the facility layout in an agv-based modular manufacturing system, Int.

J. Prod. Res. 57 (17) (2019) 5538–5555, https://doi.org/10.1080/00207543.2018.1533654, https://www.tandfonline.com/doi/abs/10.1080/00207543.2018.1533654.
[6] V.A. Cicirello, Variable annealing length and parallelism in simulated annealing, in: Tenth Annual Symposium on Combinatorial Search, 2017.
[7] R. Diekmann, R. Lüling, J. Simon, Problem independent distributed simulated annealing and its applications, in: Applied Simulated Annealing, Springer, 1993, pp. 17–44.
[8] R.W. Eglese, Simulated annealing: a tool for operational research, Eur. J. Oper. Res. 46 (3) (1990) 271–281, https://doi.org/10.1016/0377-2217(90)90001-R, https://www.sciencedirect.com/science/article/pii/037722179090001R.
[9] R.W. Floyd, Algorithm 97: shortest path, Commun. ACM 5 (6) (1962) 345, https://doi.org/10.1145/367766.368168.
[10] S.B. Furber, F. Galluppi, S. Temple, L.A. Plana, The SpiNNaker project, Proc. IEEE 102 (5) (2014) 652–665, https://doi.org/10.1109/JPROC.2014.2304638, https://ieeexplore.ieee.org/abstract/document/6750072.
[11] P. Gopakumar, M.J.B. Reddy, D.K. Mohanta, Novel multi-stage simulated annealing for optimal placement of pmus in conjunction with conventional measurements, in: 2013 12th International Conference on Environment and Electrical Engineering, IEEE, 2013, pp. 248–252, https://ieeexplore.ieee.org/abstract/document/6549625.
[12] S. Kirkpatrick, D.C. Gelatt, M.P. Vecchi, Optimization by simulated annealing, Science 220 (4598) (1983) 671–680, https://doi.org/10.1126/science.220.4598.671, https://science.sciencemag.org/content/220/4598/671.
[13] A. Ludwin, V. Betz, Efficient and deterministic parallel placement for FPGAs, ACM Trans. Des. Autom. Electron. Syst. 16 (3) (2011) 1–23.
[14] J. Mingjun, T. Huanwen, Application of chaos in simulated annealing, Chaos Solitons Fractals 21 (4) (2004) 933–941, https://doi.org/10.1016/j.chaos.2003.12.032, https://www.sciencedirect.com/science/article/pii/S0960077903006866.
[15] M. Naylor, S.W. Moore, D. Thomas, Tinsel: a manythread overlay for FPGA clusters, in: 2019 29th International Conference on Field Programmable Logic and Applications (FPL), IEEE, 2019, pp. 375–383.
[16] S.W. Paek, S. Kim, O. de Weck, Optimization of reconfigurable satellite constellations using simulated annealing and genetic algorithm, Sensors 19 (4) (2019) 765, https://doi.org/10.3390/s19040765, https://www.mdpi.com/1424-8220/19/4/765.
[17] N.R. Quinn Jr, The placement problem as viewed from the physics of classical mechanics, in: Papers on Twenty-Five Years of Electronic Design Automation, 1988, pp. 67–72.
[18] D.J. Ram, T. Sreenivas, K.G. Subramaniam, Parallel simulated annealing algorithms, J. Parallel Distrib. Comput. 37 (2) (1996) 207–212.
[19] G. Rudolph, Massively parallel simulated annealing and its relation to evolutionary algorithms, Evol. Comput. 1 (4) (1993) 361–383.
[20] G. Sergey, Z. Daniil, C. Rustam, Simulated annealing based placement optimization for reconfigurable systems-on-chip, in: 2019 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (ElConRus), IEEE, 2019, pp. 1597–1600, https://ieeexplore.ieee.org/abstract/document/8657251.
[21] J. Torres-Jimenez, E. Rodriguez-Tello, New bounds for binary covering arrays using simulated annealing, Inf. Sci. 185 (1) (2012) 137–152.
[22] J. Torres-Jimenez, I. Izquierdo-Marquez, A. Garcia-Robledo, A. Gonzalez-Gomez, J. Bernal, R.N. Kacker, A dual representation simulated annealing algorithm for the bandwidth minimization problem on graphs, Inf. Sci. 303 (2015) 33–49.
[23] M. Vousden, PSAP Software: Parallel Simulated Annealer for Placement (2.0.0), https://doi.org/10.5281/zenodo.5879815, 2022.



Mark Vousden has been employed at BluPoint Ltd. as a software engineer, and is currently a Research Fellow at the University of Southampton, UK. He has six journal publications in micromagnetic simulation. Current research interests are event-driven high-performance computing architectures and paradigms.



Graeme M. Bragg has previously worked for Esri UK and is currently employed by the University of Southampton as a Research Fellow on the POETS project. He has three journal and eight conference publications. His research interests include environmental sensor networks, low-power networks and many-core systems.

Andrew D. Brown has held posts at IBM Hursley Park (UK), Siemens NeuPerlach (Germany), Multiple Access Communications (UK), LME Design Automation (UK), Trondheim University (Norway), Cambridge University (UK), and EPFL (CH). He is a professor of electronics at Southampton University, UK. He is FIET, FBCS, CEng, CITP, Eur Ing and SMIEEE.