PAPER II

ON PARALLEL LOCAL SEARCH FOR PERMUTATIONS

A. Riise and E.K. Burke

Journal of the Operational Research Society, Palgrave Macmillan, 65(5) (2014)

Abstract We investigate some ways in which massively parallel computing devices can be exploited in local search algorithms. We show that the substantial speed-ups that can be gained from parallel neighbourhood evaluation enables an efficient best improvement local search, and that this in turn enables further speed-ups through selection and parallel application of a set of independent, improving moves. Our experiments demonstrate a total speed-up of up to several hundred times compared to a classical, sequential best improvement search. We also demonstrate how an exchange of good partial solutions between the incumbent and best found solutions improves the efficiency of the Iterated Local Search algorithm.

1 Introduction

In this paper, we investigate how the rapidly developing technologies of massively parallel computation hardware may be exploited in the context of local search methods. Moreover, we explore how this can enable new algorithmic approaches. Different technologies are currently available, including Field Programmable Gate Arrays (FPGAs), multiple-core CPUs, and General Purpose Graphical Processing Units (GPGPUs) [3]. For this paper, we have elected to use the GPU for our experiments. Graphics cards are affordable, and exist in every personal computer. The GPU's computational efficiency, in terms of floating point operations per time unit, is today significantly higher than that of top end CPUs, and the current trend is that this gap is increasing [21, 18]. Originally a highly specialized graphics processor, the GPU has over the last few years been developed towards general purpose computation devices. This development is supported by a parallel evolution of higher level programming models, languages, and tools [18]. In our experiments, we use NVIDIA's proprietary C-like programming language, CUDA. There are, however, several other alternatives, including the recent programming language standard for heterogeneous computing, OpenCL www.khronos.org/opencl. An overview of other programming models and languages is given by Owens et al [18]. Although much is still desired in terms of tools for the high level programmer, the fact remains that GPUs are rapidly developing into flexible, highly programmable parallel computation engines. GPUs support a 'Single Program Multiple Data' (SPMD) programming model, in which many individual processors run the same code in parallel, but on different data. Successful use of this model requires that the application model has a high degree of data parallelism, that the kernel is not too complicated (for example in terms of conditional branching), and that the kernel can be made computationally efficient in terms of calculations per memory access. We will not give an introduction to GPU programming in this paper, as this has already been given by several authors. For a simple introduction to GPU programming, see for example [17]. More details on Cuda may be found through www.nvidia.com. We also recommend that interested readers look up the web site dedicated to general purpose GPU programming, www.gpgpu.org.

The usefulness of the GPU as a general purpose computation engine is reflected in the plethora of applications for which the GPU, and similar hardware, has

been used. The GPU has been successfully applied to achieve higher computational efficiency for a diversity of tasks, including sorting, searching, differential equation solving, computer vision, graph algorithms, string matching [19], and numerical linear algebra [18]. Compared to other application areas, very little has been published on the use of GPUs for heuristic search algorithms. One exception is evolutionary algorithms, which is a class of heuristics that lends itself very naturally to SPMD parallelisation. Research into parallel genetic algorithms has been carried out for a number of years. In particular parallel fine-grained (cellular) genetic algorithms have been adapted to the GPU architecture. The implementations in [21] and [22] use the GPU both for fitness evaluation and for performing genetic operations. Some authors also use the GPU in genetic programming [11, 15].

In this paper, our focus is on iterative—or local—search methods. One obvious task to parallelise in the context of local search methods is the evaluation of objective function values and constraint violations for all solutions in a neighbourhood. Such evaluation is typically responsible for a dominating part of the computation time used by local search algorithms, and the evaluation of each neighbour is typically independent of that of other neighbours. Parallel neighbourhood evaluation has already been studied by a few authors. Janiak et al [13] uses the GPU to evaluate neighbourhoods in a Tabu Search method [9] for the travelling salesman problem and the flow shop scheduling problem. Similarly, Luong et al [17] do the same for the travelling salesman problem, the quadratic assignment problem and the permuted perceptron problem. They also use this GPU-based neighbourhood evaluation in conjunction with co-operating solvers working in parallel on several CPUs. In this paper, we first present the effect of parallel neighbourhood evaluation for general permutation problems, confirming earlier results, but with somewhat larger gains in computational efficiency than previously reported. We then go on to demonstrate how the local search itself can be naturally modified to take the maximum advantage of this gain in neighbourhood evaluation efficiency. We show that a parallel local search, applying a set of mutually independent moves at each iteration rather than selecting a single move, significantly improves the search performance.

We describe a simple algorithm for dynamic adaptation of diversification strength. This minimises the problem of our chosen meta-heuristic, Iterated Local Search (ILS), repeatedly returning to the same local optimum, while still balan-

cing diversification against intensification. We also modify the ILS by exploiting good quality partial solutions that are found in locally optimal solutions returned by the underlying local search. We achieve this by combining the currently best found solution with the returned incumbent solution, which may improve the best solution even if the incumbent in itself is not an improvement. The effort made to produce the incumbent is then not completely wasted, and the overall search becomes more efficient. The idea of combining solutions is similar to that of 'tour merging' (see for example Cook and Seymour [6]), although our implementation is different.

We consider problems whose solutions can be modelled as a sequence –or permutation—of distinct elements. Without loss of generality we will represent solutions to problems of size n as permutations of the integers in $[0, n-1]$. We consider the general case where permutations are cyclic, by which we mean that the permutation $\{0,5,6,1,4,2,3\}$ is equivalent to $\{4,2,3,0,5,6,1\}$. When formulating rules for move independence we will assume that the objective function has the form:

$$f(s) = \sum_{i=0}^{n-1} C_(s_i, s_{i+1}) + C_(s_{n-1}, s_0) \tag{1}$$

Without loss of generality, we will assume that this function is to be minimized. In (1), s_i is the i'th element in the permutation s, and $C(a,b)$ is the cost that comes from letting element a be followed by element b. In addition, we assume that $C(a,b) = C(b,a)$. Such problems may be found in a wide range of planning, scheduling, and design applications. If there are no additional constraints, this assumption gives us the well known Travelling Salesman Problem [2]. This is the problem of finding the shortest route between n cities, where each city is visited exactly once. Equivalently, the problem is to find the shortest Hamilton cycle in a complete undirected graph where each node represents a city. This is the problem that we use as our test bench for numerical experiments, although our algorithm and move operators are valid for permutations in general (with the above mentioned assumptions).

In Section 2, we define some fundamental concepts and introduce our local search framework. This includes the introduction of our modified ILS. In order to avoid stagnation of the search, we embed the ILS in a restart framework, which

again exploits the above idea of exploiting good partial solutions. This is presented in Section 3. Section 4 demonstrates how the GPU can be used for parallel evaluation of neighbours at each local search iteration, and how this improves performance compared to using normal sequential evaluation. We then move on to the main contribution of this paper, showing how local search can benefit further from parallel move evaluation by applying a number of independent improving moves simultaneously. This mechanism is, to the best of our knowledge, the first application of compound moves based on an exact definition of inter-move dependence, and implemented in conjunction with a massively parallel move evaluation. It is presented in detail in Section 5. Computational results for all experiments are reported and discussed in Section 6, and we draw conclusions and suggest directions for further research in Section 7.

2 The local search framework

In this section, we present the algorithmic framework in which we have carried out our experiments. The framework has largely been selected based on ease of implementation for general permutation problems, rather than being especially tuned to any given problem type. This is because our aim is to demonstrate the effectiveness of massively parallel local search algorithms in general, rather than creating a new record-breaking algorithm for a particular problem.

2.1 Local search

At the heart of the applied search methodology is a basic best improvement iterative improvement algorithm, commonly known as *Descent*. Assuming that a feasible solution s to the problem P is known, this method performs a complete evaluation of the neighbourhood defined by some move operator k on solution s. The result is a set of all improving moves, M_s^k. If $M_s^k \neq \emptyset$, the best move, $m^* \in M_s^k$ is applied to s to produce a new solution . This process is iterated until no more feasible and improving moves can be found ($M_s^k = \emptyset$). s is then a local optimum with respect to the objective function and the operator k, but is not in general a global optimum. An introduction to local search algorithms can be found in Hoos and Stützle [12].

Our move operators are quite common for permutation problems. The first is called 'Relocate' (sometimes called 'Insert'). Let $s \in S$ be a feasible solution, modelled as a permutation. Then, $s(i)$ is the i'th element of s. The Relocate

operator, *Rel(i,j)* removes item $s(i)$, and inserts it immediately after $s(j)$ in the sequence. The other operator is the well known '2-opt' operator. Let the existence of an edge (a,b) in the permutation signify that $s(a)$ is immediately followed by $s(b)$. The 2-opt operator, *2-Opt(i,j)* replaces the edges $(i,i+1)$ and $(j,j+1)$ with the edges (i,j) and $(i+1,j+1)$. The sub sequence $\{s(i+1),s(i+2),\ldots,s(j)\}$ is reversed, yielding a new consistent permutation.

```
VND
Input:  initial solution s
  1. b = s
  2. k = twoOpt
  3. while (k != NULL)
     (a) s' = Descent(s, k)
     (b) k = SelectMO(s, s')
     (c) s=s'
  4. return s
```

Figure 1: The basic Variable Neighbourhood Descent.

Since we want to use *Descent* with both of these move operators, we wrap the basic *Descent* function in a standard Variable Neighbourhood Descent [10], as illustrated in Figure 1. The basic idea of Variable Neighbourhood Descent is that when a run with *Descent* with a given move operator reaches a locally optimal solution, this solution is not necessarily a local optimum with respect to other move operators. When the function *Descent* returns, therefore, the next operator is chosen by the function *SelectMO*, and a new Descent search is attempted. In our implementation, *SelectMO* chooses the next move operator deterministically. When a complete descent search has been attempted with both move operators without yielding any improvement, we know that s is a local optimum with respect to all move operators, and *SelectMO* returns *NULL*. *VND* then returns this locally optimal solution.

In order for the search for a better solution to continue when *VND* returns, we need to escape the well of attraction of the current local optimum. There are many methods for doing this, often called meta-heuristics. The presentation of an overview of popular meta-heuristics is outside the scope of this paper, but there are several books on the subject [9, 4, 12].

2.2 Iterated Local Search

In this paper, we will use a simple and general meta-heuristic framework known as Iterated Local Search (ILS) [16]. This choice is motivated by the generality and simplicity of this meta-heuristic, and its transparency to the effectiveness of the underlying local search move operators that are the focus of the paper. Our

implementation of ILS may be simply described by Figure 2. On line 2.a, VND returns a locally optimal solution. The function *Combine* is an addition to the otherwise traditional ILS formulation. This mechanism seeks to exploit the results of the last local search even in cases where s will not be accepted as the new best solution. The function may modify both s and b (see line 2.b in Figure 2). It will be described in detail in Section 2.3. The function *Accept* is used to determine if a new incumbent solution s is to be accepted as the current best, b. When s is not accepted, the function also chooses whether to continue the search with the incumbent solution, s, or to continue from the best found solution, b. In most implementations, the incumbent is accepted as a new best solution if it has a better objective value than the existing best solution. We apply the same strategy, and our implementation of *Accept* always returns b.

```
IteratedLocalSearch
Input:  initial solution s
  1. b = s
  2. while (!  stop)
      (a) s = VND(s)
      (b) Combine(s,b)
      (c) s = Accept(s, b)
      (d) s = Diversify(s)
  3. Return b
```

Figure 2: Pseudo-code for ILS

It is, of course, the diversification step (line 2.d) that enables *IteratedLocalSearch* to continue searching after *VND* has returned a locally optimal solution. The *Diversify* function typically changes the solution more drastically than our move operators, and the change is generally not improving the objective value. The desired effect is that the starting point solution for the next local search is outside the well of attraction of the previously returned local optimum. Since the topology of the search space in general is unknown, it is difficult to analytically determine the degree of perturbation of the solution that is necessary to achieve this escape. We therefore introduce a heuristic mechanism for continuous adaptation of diversification strength. The details of the function *Diversify* will be given in Section 2.4.

2.3 Implementation of *Combine*

When the function *VND* returns a solution s as described in Figure 2, s represents a locally optimal solution that may, or may not, be better than the best

found solution so far, b. However, even if s is worse than b, there may be parts of s that are better than corresponding parts of b. The last run of *VND* may have discovered some good partial solution structure that is compatible with b, but that is not present in b. Since s is worse than b, this information would be lost in a standard ILS meta-heuristic. This is a waste of computing effort that we wish to amend by introducing a simple combination operator on the two solutions. Let us illustrate how this can be done by taking the Travelling Salesman Problem as an example. Let $b = \{0,6,4,7,2,3,1,5,8,9,10,11,12\}$, and $s = \{0,6,2,4,7,3,1,5,12,11,10,9,8\}$ be two solutions to a problem instance. Assume that b is better than s because the partial sequence $\{..8,9,10,11,12\}$ is very much better than $\{..12,11,10,9,8\}$. Assume also, that the sub-sequence of $s, \{\ldots 2,4,7\ldots\}$ is better than the sub sequence in b, $\{\ldots 4,7,2\ldots\}$. Apart from these differences, s and b are identical. Since the last two sub sequences contain the same elements, and are preceded and followed by the same elements in both solutions, we can freely replace the sub-sequence of b with the sub sequence from s (remember that we are not considering any explicit constraints in this simple example). This improves b, while keeping it as a valid permutation. The computational effort of finding s is therefore not wasted, even if s is not an improvement over b. b is improved anyway. Symmetrically, the returned local optimum s may be improved by looking for better and compatible sub-structures in b, before the two solutions are compared by the *Accept*-function in Figure 2. Note that for the Travelling Salesman Problem, this combination mechanism has a similar function as the tour merging mechanism used by for example [6]. However, while their implementation is based on solving a reduced Travelling Salesman Problem constructed from a pool of solutions, our implementation focuses on extracting interchangeable partial solutions between two given solutions. In our case, we use this combination mechanism as an integrated part of our restart ILS implementation. Although our implementation is for permutation problems, the same idea may be applied to solutions to many kinds of optimisation problems. However, it is a requirement that different parts of the solution are independent in the sense that the total objective value for the solution is calculated as the sum of contributions from such parts. It must also be checked that exchanging partial solutions between s and b leaves both solutions feasible with respect to the constraints. For some problems, as for example our simple permutation problems, this is ensured

by the modelling, and does not need to be checked explicitly. In Section 6.3 we will show how this combination mechanism improves the efficiency of the ILS.

2.4 Implementation of *Diversify*

As our basic diversification mechanism we chose the so called 'double-bridge' move, which has been much used with ILS for the Travelling Salesman Problem [9]. This move is implemented by randomly removing four edges from the (cyclic) permutation. The resulting sub-sequences (s_1, s_2, s_3, s_4) are re-ordered as (s_1, s_4, s_3, s_2) to form a new cyclic permutation. Note that the selection of cutting points in our double-bridge implementation is completely random, which is very general, but which is likely to be inferior to a tailored version that exploits domain knowledge for a particular application. Our initial experiments showed that when using this simple perturbation mechanism, the subsequent local search very often returned to the previous local optimum, or another recently visited local optimum. The corresponding search effort was therefore wasted. To minimize this problem, our diversification applies a number (x) double-bridge moves at each ILS iteration. If the last found local optimum already exists in a list of recently found local optima, x is increased with an amount that increases with the number of (ILS) iterations since that optimum was last seen. The local optimum is then added to the list. The list is kept at below a maximum length (5 in our experiments), if necessary by deleting the oldest entry. If a local optimum is returned that is not in the list, x is decreased with the current list length (but with a minimum value of 1). This adaptive diversification strength reduced our algorithm's cycling in the search space, and improved our search results considerably.

3 Restart and combination of solutions

The ILS algorithm tends to stagnate after a while, giving less improvement per time unit than what can be expected for a stochastic process [12]. We make a simple amendment to this by performing a soft restart [20] of the ILS at regular intervals, each from a new random solution. The interval length is set to n iterations, where n is the size of the permutation problem. Again we use our combination mechanism. Before each re-start, we try to combine the best solution from the last ILS with each solution that has been found in previous searches. The point of combination of solutions at this level is that run with ILS may discover different partial solution structures. It may, therefore, be that a new returned solution

from the latest run with ILS can be further improved by using some sub-structure from one of the previously found solutions in the pool. Although our experiments show that this does not happen very often, it does occur. The computational effort is insignificant compared to a search with ILS. Note that we do not let the combination of solutions modify the previously found pool solutions. This is because we want to keep the diversity that is represented in the pool; such modifications would make the solution pool more and more homogenous.

4 Parallel move evaluation

The efficiency of a local search algorithm depends heavily on the complexity associated with evaluating moves in a neighbourhood at each search step. Since neighbourhood evaluation is normally computationally expensive, many sequential methods explore neighbourhoods only partially, for example by random sampling, accepting the first found improving move ('first improvement' search), or applying some kind of neighbourhood reduction technique. In this study, however, we consider the case when all neighbours are evaluated, and the best is chosen as the next incumbent solution. This is sometimes called 'steepest descent' or 'best improvement' search. The normal *Descent* function traverses the entire neighbourhood sequentially at each iteration. Each possible move is evaluated based on the change it would incur to the objective function value. The computational complexity of sequential neighbourhood evaluation is proportional to the size of the neighbourhood. For both our neighbourhoods the size is of order $O(n^2)$, where n is the length of the permutation.

In this section, we show how a neighbourhood can be evaluated in parallel. We demonstrate what effect this has on the total search performance by comparing sequential and parallel neighbourhood exploration in an otherwise identical search algorithm. To achieve this we use a modified descent algorithm that we call *DescentWithParallelEvaluation*. This is identical to the normal *Descent* function, except that at each iteration the evaluation of moves is distributed among a large number of GPU kernel threads, each of which evaluates one move. The kernels threads are in reality grouped in blocks, and the moves in each block—or actually each 'warp'—are evaluated in parallel. These complications are hidden by the CUDA programming interface, and the program code can be written as if all calculations are done simultaneously. The speedup compared to sequential

evaluation is significant. As reported in Section 6.1, using *DescentWithParallelEvaluation* instead of the basic *Descent* speeds up the average computation time by a factor of more than 70. Note, however, that since the number of processors on the GPU is finite, the computation time is still of order $O(n^2)$.

Once the parallel evaluation is done, the GPU kernel code extracts the best move in each thread block. The selection of the overall best move is then done by sequentially comparing the best moves from each block. This is done on the CPU, and the total computational complexity of the move selection is also of order $O(n^2)$.

Even though the concept of parallel move evaluation is very general one should note that there are some limitations. The method requires that the code for evaluating all objectives and constraints for each individual move is quite simple, as a single GPU kernel thread is slower than a typical CPU. Also, evaluation should not involve too much data transfer between the host memory and the GPU memory, as this represents a considerable overhead.

5 Parallel search

As mentioned in Section 2.1, traditional best improvement Descent iteratively applies the best improving move, $m^* \in M_s^k$, where k is the move operator and s is the current solution. We define two moves $m_1, m_2 \in M_s^k$ to be *independent under k* if the three solutions $s_1 = m_1(s)$, $s_2 = m_2(s)$, and $s_3 = m_1(s_2) = m_2(s_3)$ are feasible, and the objective values satisfy $(f(s)-f(s_1))+(f(s)-f(s_2)) = f(s)-f(s_3)$. The idea behind the parallel search algorithm that is presented below is that a set of pairwise independent moves can be applied simultaneously to achieve a objective value improvement equal to the sum of the improvements associated with each move. A set of rules can be formulated for each move operator to determine if two moves are independent. The dependency rules for the operators that we use in this paper are defined in Section 5.3.

5.1 Parallel Descent

We now formulate a new parallel version of the core descent search algorithm (see Figure 3). This algorithm is a further refinement of *DescentWithParallelEvaluation,* and like that function, it uses a parallel evaluation of all possible moves (line 2.a in Figure 3).

```
ParallelDescent
Input:  initial solution s, type of
neighbourhood, nh
  1. b = s
  2. do
     (a) moves = nh.EvaluateAllMoves(s)
     (b) imp_moves=ExtractImproving(moves)
     (c) sel_moves=SelIndepMoves(imp_moves)
     (d) s.ApplyAll(sel_moves)
     (e) if( s.value < b.  value)
           b = s
     (f) else
           return b
```

Figure 3: The parallel Descent algorithm.

ParallelDescent, however, selects a set of independent improving moves in each iteration (*SelIndepMoves*, on line 2.c), which are then all applied simultaneously (*ApplyAll* on line 2.d).

The actual selection of independent improving moves (function *SelIndepMoves*) is described in detail in Section 5.2. Initially, we tried implementing both move selection and move application on the GPU to avoid copying information about moves back to the host. However, it turned out that a simple sequential selection heuristic implemented on the CPU was more efficient. This is because the number of improving moves on average is quite low, and the GPU's parallel computation capacity could therefore not be well exploited. Also, the CPU is a more powerful processor than the GPU multiprocessors. For the same reason, once we had already accepted the overhead of copying move information back to the host, it was also beneficial to implement a sequential application of the few selected independent moves in the *ApplyAll* function on the host. The function *ExtractImproving* (line 2.b in Figure 3) uses the GPU-based compacting functionality found in the CUDPP Cuda library (see gpgpu.org/developer/cudpp) to return a list of improving moves based on the parallel evaluation.

Note that selecting and applying a set of independent simple moves is equivalent to selecting and applying one *compound move* from a neighbourhood that consists of all possible combinations of pair wise independent 'simple moves'. If, as for our simple neighbourhoods, the basic neighbourhood has a size that is polynomial in the problem size, such a 'compound move' must be selected from a neighbourhood of non-polynomial size. For our case, the basic neighbourhoods

have size $l = n^2$. If it is theoretically possible to select at most k independent moves out of these, then the size of the compound neighbourhood will be of order

$$O\left(\sum_{i=1}^{k}\binom{l}{k}\right) = O\left(\sum_{i=1}^{k}\frac{l!}{i!(l-i)!}\right) \tag{2}$$

The value of k will of course depend on n, and on the defined move dependency rules. Instead of selecting among all such compound moves, we select a set of independent simple moves only from the set of improving simple moves. This corresponds to a drastic reduction in the corresponding compound neighbourhood.

Several authors have investigated exponentially sized neighbourhoods with special structure or properties that allow them to be explored in polynomial time [7]. A survey of search methods using such neighbourhoods, often called Very Large Neighbourhood Search, is presented by Ahuja et al [1]. In particular, our strategy is similar to the strategy applied by 'Dynasearch' [5], in the sense that both rely on selecting a set of independent simple moves to construct 'compound moves'. In Dynasearch, however, the set of independent moves are selected exactly by a dynamic programming algorithm. To achieve this, Congram et al [5] use approximate, simplified, dependency rules. Similarly, Ergun et al [8] also use a weak definition of swap move dependency for the Travelling Salesman Problem, defining only moves that span overlapping ranges of indices to be inter-dependent. Obviously, this strategy excludes many combinations of moves that are, strictly speaking, independent.

In the present work, we have chosen a different strategy. Instead of using an exact solution method to an approximate move dependency problem, we apply a heuristic solution method to the selection problem resulting from an exact move dependency definition. This means that our dependency rules are complete and exact, allowing for example independence between two nested 2-opt moves (see Section 5.3 for details).

5.2 Selecting a set of independent improving moves

Intuitively, the parallel application of moves should give a speed-up in the search that is proportional to the number of moves applied at each iteration. However, there is a computational cost associated with the selection of these moves. The dependency rules for the move operator defines a dependency graph $G(V,E)$,

where the vertexes V represents the improving moves, and the edges, E, represents dependencies between them. The selection of pair wise independent improving moves can be viewed as a maximum weight stable set problem in G. For a general graph (we have no á priori knowledge of the structure of the graph, since this is completely determined by which moves in the neighbourhood that is found to be improving), this problem is the weighted version of the maximum stable set problem, which is known to be NP-hard [14]. To achieve a reasonable scalability, we therefore chose to implement an approximate—heuristic—selection method. Pseudo-code for the sequential selection function *SelIndepMoves* is given in Figure 4, which should be self-explanatory. The parameter *n_passes* determines the number of passes through the comparison loop. More passes will give better results, but at the cost of increased computation time. In the experiments in Section 6 we have used *n_passes = 1*, because our initial experiments showed that this gave the best overall performance for the algorithm.

5.3 Dependency rules

Dependency rules for a certain type of move depend not only on move operator definitions, but also on problem constraints, how the solution is modelled, and what the objective function is. For very rich, real world problems, evaluating constraints and objectives may also involve propagating some entity (for example time or resource capacity) through parts of the solution. Such propagation will typically increase the connectivity of the move dependency graph.

In this paper, we base the dependency rules on the assumption that the (cyclic and reversal-symmetric) permutation problem has no additional constraints. For such problems, all constraints are handled implicitly by the modelling. Assuming that (1) holds (as for the Travelling Salesman Problem), we can formulate the following dependency rules:

- For the Relocate neighbourhood, two moves, *Rel(i,j)* and *Rel(k,l)*, are considered dependent if and only if $|i-k| \leq 1 \vee |i-l| \leq 1 \vee |k-j| \leq 1 \vee j = l$.
- For the 2-opt neighbourhood, two moves, *2-Opt(i,j)* and *2-Opt(k,l)* are considered dependent if and only if and only if $i < k < j < l \vee k < i < l < j \vee k = j \vee k = i \vee l = i \vee l = j$. Note that this allows for independence between nested 2-opt moves.

```
SelIndepMoves
Input:  moves, n_passes
 for pass = 1 to n_passes
      for i = 1 to moves.Size
          m = moves[i]
          m_lost = false
          for j = 1 to selmoves.Size
              sm = selmoves[j]
              if(m.ConflictsWith(sm))
                  if(m.value<sm.value)
                    loosers.Add(sm)
                  else
                    m_lost = true
          if( !  m_lost)
              selmoves.Add(m)
              for k = 1 to loosers.size
                   selmoves.Remove(loosers[k])
```

Figure 4: Outline of the heuristic for selecting independent moves.

6 Experimental results

As mentioned in the introduction, our test bench is the well known Travelling Salesman Problem, which is one of the most thoroughly investigated problems in the literature. The Travelling Salesman Problem has the advantages that it is easy to explain, and it is a typical example of a permutation problem. The problem is NP-complete. There are a number of well known benchmark instances available, which makes for easy testing and comparison. However, we should note that in the following experiments, no tuning of the algorithm has been done that takes into account any properties of the Travelling Salesman Problem other than those that are present in any general permutation problem. Our aim is to investigate the effect of the various search mechanisms on general permutation problems, rather than finding new best solution to the benchmarks. The selected test cases were downloaded from TSPLIB (comopt.ifi.uni-heidelberg.de/software/TSPLIB95) and the set of 'National Traveling Salesman Problems' presented by William Cook at www.tsp.gatech.edu/world/countries.html. The problem instances are of different

sizes; some have very apparent structure, while in others cities are more randomly distributed. Optimal values are known for all the instances.

The experiments where performed on a PC with an Intel Core i7 2.67 GHz CPU and 6 GB of memory. The GPU was a NVIDIA GeForce GTX 280, with 240 cores and 30 multiprocessors. The "compute capability" of this card, which describes which features a NVIDIA GPU supports, is 1.3.

6.1 Parallel versus sequential move evaluation

In Section 4, we discussed the idea of evaluating moves in parallel. In this section we present experimental results from a comparison between parallel and sequential move evaluation. Both algorithms use the function *IteratedLocalSearch* as illustrated in Figure 2. The only difference is the implementation of the *Descent* function used in the *VND* function in Figure 1. The local search with parallel evaluation uses the function *DescentWithParallelEvaluation*, which has some of the functionality implemented on the GPU as explained in Section 5.1. The search with sequential evaluation uses the normal sequential *Descent* function, which is implemented on the CPU.

Both algorithms use evaluations of the difference in objective values associated with each move, rather than applying the move and performing a full evaluation of the resulting solution. The GPU code has been designed to minimise global memory access on the graphics card, but little additional fine tuning has been done. No special effort has been made to use efficient data structures in the host code, other than using common data structures defined in the .Net framework.

To evaluate the practical benefit of using parallel evaluation, we ran experiments for problems of different sizes. Since the same search logic is used in both cases, the search follows the same path through the search space. We are at this point only interested in demonstrating the speedup of this search using parallel evaluation, rather than comparing different algorithms. It is therefore not necessary to run each case more than once, even if our algorithms are of a stochastic nature (provided we use the same random seed for both algorithms). We ran each algorithm for the cases *d198*, *d493*, and *d657*. We found that parallel evaluation speeds up the search by a factor of more than 70 for all three cases. This factor is larger than those found in previous work, but this may be due to the rapid development of GPU hardware.

6.2 Parallel versus sequential search

In this section we report from our investigation of the parallel local search method, as introduced in Section 5. We compare the performance of this parallel search algorithm with the sequential search. Both searches uses parallel neighbourhood evaluation, and both use the functions *IteratedLocalSearch* and *VND*. The only difference is in the implementation of the descent-function. The parallel search algorithm uses *ParallelDescent* of Figure 3, while the sequential search algorithm uses *DescentWithParallelEvaluation*.

Table 1: Mean time to reach a 1% deviation from the optimum value.

	Sequential Search				**Parallel Search**				
Case	Run%	Mean	Min	Max	Run%	Mean	Sp.up	Min	Max
d198	100	0.17	0.00	0.55	100	0.56	0.3	0.05	1.26
d493	100	34.49	11.40	71.89	100	17.52	2.0	5.85	44.65
d657	100%	99.84	45.66	288.88	100%	50.47	2.0	9.84	150.03
uy734	100%	136.74	72.49	315.26	100%	33.11	4.1	14.60	69.00
d1291	100%	444.87	119.48	1052.79	100%	63.44	7.0	26.33	118.37
d1655	90%	1083.72	363.11	2208.15	100%	144.15	7.5	33.48	478.76
d2103	100%	724.88	309.80	1187.33	100%	162.06	4.5	8.64	624.59
nu3496	0%	-	-	-	88%	1586.16	-	608.06	3537.78

Table 2: Mean time to reach a 3% deviation from the optimum value.

	Sequential Search				**Parallel Search**				
Case	Run%	Mean	Min	Max	Run%	Mean	Sp.up	Min	Max
d198	100%	0.01	0.00	0.06	100%	0.10	0.1	0.03	0.25
d493	100%	0.69	0.16	1.78	100%	0.36	1.9	0.05	1.11
d657	100%	10.78	3.70	22.53	100%	2.61	4.1	0.70	5.07
uy734	100%	16.65	5.27	33.46	100%	5.23	3.2	2.11	8.19
d1291	100%	22.13	0.58	67.56	100%	2.74	8.1	0.14	8.05
d1655	100%	73.92	33.20	135.97	100%	9.42	7.8	3.57	15.58
d2103	100%	32.55	0.98	98.09	100%	2.40	13.6	0.12	8.88
nu3496	100%	1154.20	545.52	2196.62	100%	137.58	8.4	69.12	210.02

We ran a series of experiments with both algorithms, and for each test case. Each run was terminated after one hour. For each case, we collected statistics for a qualified run time distribution (QRTD) [12], which gives the fraction of all runs that reached a certain solution quality as a function of computation time. Fig-

Table 3: Mean time to reach a 5% deviation from the optimum value.

	Sequential Search				Parallel Search				
Case	Run%	Mean	Min	Max	Run%	Mean	Sp.up	Min	Max
d198	100%	0.01	0.00	0.02	100%	0.06	0.2	0.03	0.17
d493	100%	0.16	0.11	0.20	100%	0.06	2.5	0.03	0.17
d657	100%	0.98	0.27	6.46	100%	0.47	2.1	0.06	1.53
uy734	100%	1.35	0.36	5.21	100%	0.97	1.4	0.08	2.00
d1291	100%	1.94	0.50	10.31	100%	0.37	5.3	0.09	1.14
d1655	100%	3.11	1.00	8.86	100%	1.25	2.5	0.25	5.34
d2103	100%	1.17	0.98	1.31	100%	0.16	7.2	0.12	0.20
nu3496	100%	71.05	12.46	136.56	100%	13.90	5.1	3.46	31.89

ure 5 shows the QRTD for the case d657, at a solution quality gap of 0.5% from the optimum value. It demonstrates clearly the advantage of the parallel search. A summary of the results for all cases are given in Tables 1 - 3. In each table, 'Mean', "Min" and "Max" denotes the mean, minimum, and maximum time used, taken over those runs that reached the given deviation from the optimal value. 'Runs' shows the fraction of such runs (out of all runs), while "Sp.up" denotes the mean speedup. The parallel search clearly achieves a significant speedup over the sequential search. The results are summarized in Figure 6, where the mean speedup is shown as a function of the size of the considered test instances. We see that the overall trend is that the speedup increases with n. This seems reasonable because, typically, so does the number of independent parallel moves that may be applied at each iteration. Each additional simultaneous move saves the computation time of one full neighbourhood evaluation, which for our neighbourhoods is $O(n^2)$. However, the speedup is not a uniformly increasing function of n. We believe this is because the individual instances have different structural properties, which, in addition to size, also influences the number of independent improving moves at each iteration.

Note that for the smallest case in our collection, *d198*, sequential search is faster. This is probably because this case gives few independent improving moves at each iteration, and the overhead selecting such moves dominates the savings in neighbourhood evaluation time.

We see from Figure 6 that for all but the smallest case, the parallel search is between approximately 2 and 14 faster than the sequential search, even if both use parallel evaluation. Multiplying with the speed up factor found in Section 6.1,

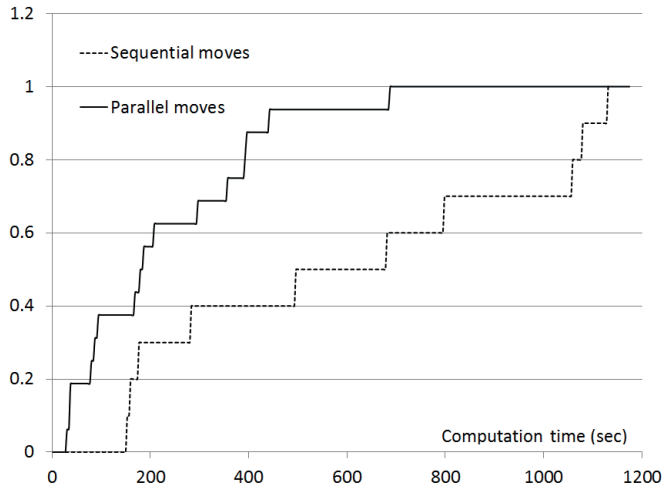

Figure 5: QRTD's for parallel and sequential search for d657, at 0.5% deviation from the optimum.

this gives us a considerable total speedup compared to the traditional sequential best improvement local search with sequential neighbourhood evaluation. For example, to reach a solution quality at 1% from the optimum, a total speedup factor of between approximately 140 and 525 is achievable for all but the smallest test instance.

6.3 The effect of combining local optima

In all the above experiments we have used the *Combine* function, as introduced in Section 2.3. In this section we present empirical results that illustrate the advantage of using the combination of found local optima as an extension to the traditional ILS meta-heuristic. We compare two versions of our algorithm. Both versions use *IteratedLocalSearch*, *VND*, and *ParallelDescent.* The only difference between them is that one version uses the Combine function, both inside the ILS (Figure 2) and in the restart framework, while the other version does not. We ran both versions of the algorithm several times each on each of our test cases, with a set time limit. Early on in the search, the effect of re-combination of local optima

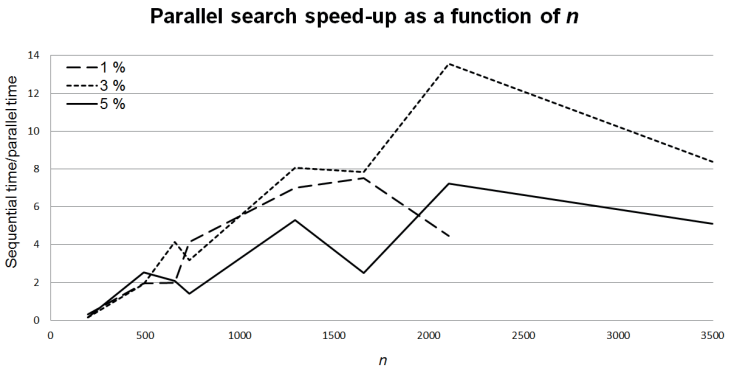

Figure 6: The speedup ratio of parallel search relative to sequential search, based on the mean computation times used to reach different deviations from the optimal value.

does not seem to be important. As can be seen from Table 4, however, as the search approaches the optimum the combination has a positive effect on mean run times for all but one case. This effect is mostly due to the use of the Combine function in the ILS. Every now and then, however, combining the results of each restart of ILS also results in a new best solution.

Table 4: The effect of combination of local optima, at 1% deviation from optimal values.

	Without Combine				**With Combine**				
Case	Run%	Mean	Min	Max	Run%	Mean	Sp.up	Min	Max
d198	100%	0.94	0.03	2.07	100%	0.56	1.7	0.05	1.26
d493	100%	49.59	3.06	140.60	100%	17.52	2.8	5.85	44.65
d657	100%	75.07	24.27	160.27	100%	50.47	1.5	9.84	150.03
uy734	100%	69.93	18.21	155.11	100%	33.11	2.1	14.60	69.00
d1291	100%	69.37	21.51	195.56	100%	63.44	1.1	26.33	118.37
d1655	100%	366.81	61.60	828.66	100%	144.15	2.5	33.48	478.76
d2103	100%	100.41	9.38	467.78	100%	162.06	0.6	8.64	624.59

7 Conclusions

Selecting and applying a set of independent moves at each iteration is applicable to best improvement local search in general, even if neighbourhood evaluation is done in sequence (see for example [5]). However, sequential local search usually employs a first improvement strategy, or some kind of neighbourhood filtering technique, to avoid an expensive evaluation of the entire neighbourhood. In this paper, however, we have shown that an efficient, parallel evaluation of the entire neighbourhood also enables an efficient best improvement search, which in turn enables a parallel local search with a total speedup of up to several hundred times over the corresponding sequential algorithm. We have also shown the beneficial effect of combining incumbent and best found solutions as a part of the ILS, although the gain in search efficiency is less than that resulting from the parallel search. Our use of adaptive diversification strength also contributed to the efficiency of all versions of the algorithm. SPMD-enabled technologies such as the GPU are becoming increasingly mature and available. The above results show that, where it is possible, this kind of technology should be exploited to achieve a significant performance gain for local search methods. Research into new algorithms that can exploit this technology will therefore become increasingly important. While the present, and earlier, work exploits the computational power of the GPU within the framework of an essentially sequential, CPU-based algorithm, we believe that new meta-heuristic approaches should be investigated that can more fully exploit this emerging technology.

Acknowledgements

We wish to thank colleagues at SINTEF ICT, Department of Applied Mathematics, especially Christopher Dyken, Johan S. Seland, Geir Hasle, and Oddvar Kloster for valuable input and interesting discussions. This work is supported by the Research Council of Norway and the UK Engineering and Physical Sciences Research Council (EPSRC).

Bibliography

[1] R. K. Ahuja, z. Ergun, J. B. Orlin, and A. P. Punnen. A survey of very large-scale neighborhood search techniques. *Discrete Applied Mathematics*, 123(1-3):75–102, 2002. 0166-218X.

[2] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study (Princeton Series in Applied Mathematics)*. Princeton University Press, 2007. 1374811.

[3] A. R. Brodtkorb, C. Dyken, T. R. Hagen, J. M. Hjelmervik, and O. O. Storaasli. State-of-the-art in heterogeneous computing. *Sci. Program.*, 18(1):1–33, 2010.

[4] E. Burke and G. Kendall. *Search methodologies: introductory tutorials in optimization and decision support techniques*. Springer, 2005.

[5] R. K. Congram, C. N. Potts, and S. L. v. d. Velde. An iterated dynasearch algorithm for the single-machine total weighted tardiness scheduling problem. *INFORMS J. on Computing*, 14(1):52–67, 2002. 769543.

[6] W. Cook and P. Seymour. Tour merging via branch-decomposition. *INFORMS JOURNAL ON COMPUTING*, 15(3):233–248, 2003.

[7] V. G. Deineko and G. J. Woeginger. A study of exponential neighborhoods for the travelling salesman problem and for the quadratic assignment problem. *Mathematical Programming*, 87(3):519–542, 2000. 10.1007/s101070050010.

[8] Z. Ergun, J. B. Orlin, and A. Steele-Feldman. Creating very large scale neighborhoods out of smaller ones by compounding moves. *Journal of Heuristics*, 12(1-2):115–140, 2006. 1116088.

[9] F. Glover and G. Kochenberger. *Handbook of Metaheuristics (International Series in Operations Research & Management Science)*. Springer, 2003.

[10] P. Hansen and N. Mladenović. Variable neighborhood search: Principles and applications. *European Journal of Operational Research*, 130(3):449–467, 2001. 0377-2217.

[11] S. Harding and W. Banzhaf. Fast genetic programming on gpus. In *Genetic Programming*, pages 90–101. Springer Berlin Heidelberg, 2007.

[12] H. H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, 2005.

[13] A. Janiak, W. Janiak, and M. Lichtenstein. Tabu search on gpu. *Journal of Universal Computer Science*, 14(14):2416–2427, 2008.

[14] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Plenum Press, 1972.

[15] W. Langdon and W. Banzhaf. A simd interpreter for genetic programming on gpu graphics cards. In *Genetic Programming*, pages 73–85. Springer Berlin Heidelberg, 2008.

[16] H. R. D. Lourenco, O. C. Martin, and T. Stutzle. Iterated local search. In *Handbook of Metaheuristics*, pages 321–353. Kluwer Academic Publishers, Norwell, MA, USA, 2002.

[17] T. V. Luong, N. Melab, and E.-G. Talbi. Parallel local search on gpu. Report RR-6915, DOLPHIN (INRIA Lille - Nord Europe), 04-05-2009 2009.

[18] J. D. Owens, M. Houston, D. Luebke, S. Green, J. E. Stone, and J. C. Phillips. Gpu computing. *Proceedings of the IEEE*, 96(5):879–899, 2008.

[19] M. Schatz and C. Trapnell. Fast exact string matching on the gpu. Report, Center for Bioinformatics and Computational Biology, 2007.

[20] T. Stutzle and H. H. Hoos. Analysing the run-time behaviour of iterated local search for the travelling salesman problem. In *Essays and Surveys on Metaheuristics*, Operations Research/Computer Science Interfaces Series, pages 589–611. Springer US, 2001.

[21] Q. Yu, C. Chen, and Z. Pan. Parallel genetic algorithms on programmable graphics hardware. In *Advances in Natural Computation*, pages 1051–1059. Springer Berlin Heidelberg, 2005.

[22] L. Zhongwen and L. Hongzhi. Cellular genetic algorithms and local search for 3-sat problem on graphic hardware. In *IEEE Congress on Evolutionary Computation, 2006. CEC 2006*, pages 2988–2992, 2006.