A Scalable Heterogeneous Parallelization Framework for Iterative Local Searches

Martin Burtscher
Department of Computer Science
Texas State University-San Marcos
San Marcos, TX 78666, USA

Hassan Rabeti
Department of Mathematics
Texas State University-San Marcos
San Marcos, TX 78666, USA

***Abstract*—This paper describes and evaluates a highly-scalable framework for running iterative local searches on heterogeneous HPC platforms. The user only needs to provide serial CPU or single-GPU code that implements a simple interface. The framework then executes this code in parallel using MPI between compute nodes and OpenMP and multi-GPU support within nodes. It handles all parallelization aspects, seed distribution and program termination, and it regularly records the currently best solution. We evaluate our framework on three supercomputers using a heuristic iterative hill-climbing TSP solver as well as a search for good finite-state machines. The framework scales to 2048 nodes (32,768 cores) on Ranger with less than a 5% drop in efficiency, searches over 12.2 trillion TSP tours per second on Stampede using 1024 nodes, and evaluates over 21.5 trillion FSM transitions per second using 256 CPUs and 384 GPUs on Keeneland.**

Keywords: parallelization framework, heterogeneous CPU/GPU computing, iterative local champion search

I. Introduction

Most HPC systems are built of interconnected compute nodes whose complexity is steadily increasing. Whereas older systems may have used one or two single-core processors per compute node, many recent systems employ several multi-core NUMA CPUs that are paired with GPU accelerators [1]. As a consequence, multiple levels of program parallelization are needed to fully exploit today's high-performance computers. However, parallel programming is more complex and error prone than serial coding, and supporting heterogeneity makes it even more difficult. Moreover, HPC application writers are typically domain experts, *i.e.*, not computer scientists, who tend to have little formal training in parallel programming.

One application domain that can greatly benefit from parallelization is iterative local searches (ILS). Many such search techniques exist [2], including *n*-opt iterative hill climbing, ant colony optimization, and other random-restart greedy algorithms. ILS algorithms are frequently used in engineering and real-time domains because they produce a (potentially better) solution in every iteration and can therefore be terminated at any time, for example, when a certain result quality or a runtime limit has been reached. This is in contrast to exact solvers, which generally only provide the final result but no directly useful intermediate information. For problems where the running time grows exponentially or worse with the input size, determining the optimal result is often intractable, rendering exact solvers unusable for large inputs.

To simplify the implementation of ILS on parallel systems, we have developed the iterative local champion search (ILCS) framework. It handles all complexities related to parallelization, including threading, communication, locking, resource allocation, heterogeneity, load balance, termination decision, and result recording. The user only has to write three serial C functions and/or three single-GPU CUDA functions with simple interfaces (see below). The framework then executes these functions in parallel to maximally exploit the underlying hardware. It automatically detects how many CPU cores and GPUs each compute node has. It utilizes multi-GPU and OpenMP parallelization within a node and MPI across nodes. The user has the option to only provide CPU or GPU code and can omit the MPI component for single-node processing. The framework terminates the search if the solution has not improved over a user-defined period of time.

The primary design goals of our framework are ease of use and scalability. From personal experience we know that many sophisticated tools and frameworks are underutilized in practice because they are overly complicated. For instance, they might require cryptic command line arguments and flags or configuration files, which makes it challenging to start using the tool and easy to forget how to use it. Thus, we decided to opt for maximal simplicity, even at the expense of some flexibility, to ensure that scientists and engineers can readily use our framework and to encourage them to continue using it. The ILCS framework takes no command-line arguments. Instead, it passes the command line unaltered to the first user-provided function, which returns the size of a user-defined data structure for storing a search result. The second user-provided function returns nothing and takes three parameters: a search seed, a pointer to the current champion (the best solution found so far), and a pointer to a location for saving the search result (see Section III for more details). The third user-provided function simply records or outputs the search result passed to it. We believe this interface to be simple yet powerful enough that HPC users from a wide range of domains can quickly and successfully utilize the ILCS framework.

We test and evaluate our framework on two examples. The first example is a CPU/GPU-based heuristic solver for the trav-

eling salesman problem (TSP), which is one of the most widely explored combinatorial optimization problems. Its objective is to find the shortest tour that visits all cities (*i.e.*, predetermined locations) in a given set of cities. It is used in producing and optimizing vehicle routes, service schedules, radiation hybrid maps in genome sequencing, robot arm movement, drilling in semiconductor manufacturing, overhauling gas turbine engines, and other codes where the travel distance is important [3][4][5].

Since finding an optimal TSP solution is NP-hard [6], ILS algorithms such as iterative hill climbing (IHC) are often employed to find near-optimal tours. These algorithms produce an initial solution and then improve it using heuristic techniques until a local optimum is reached that cannot be further improved. In each IHC step, a set of tour modifications, called moves, is evaluated to determine the best move [7][8]. For instance, a tour can be improved using the 2-opt heuristic, which removes edges (v_A, v_B) and (v_C, v_D) and adds edges (v_A, v_C) and (v_B, v_D) [9]. The IHC algorithm repeatedly chooses the best move as the next step, thus reducing the length of the tour until it finds a locally optimal solution. Then it restarts with a new initial solution. This process of local improvements and restarts continues until a sufficiently high-quality solution has been found or a limit on computing resources is reached [10]. Generally, the larger the number of cities, the more restarts are needed to find a good solution with high probability, making this approach computationally expensive for large inputs.

The second example is taken from our research in computer architecture. It is a CPU/GPU-based configuration-space evaluation of finite-state machines (FSMs) for predicting a long sequence of binary digits as accurately as possible. Such FSMs are widely used in dynamic branch predictors, memory disambiguation hardware, *etc*. We use them for confidence estimation [11] and real-time compression of program traces [12].

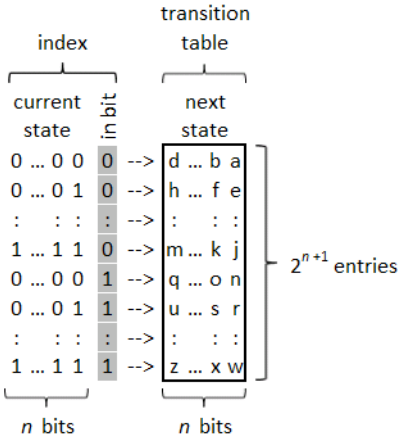

Figure 1: State transition table of *n*-bit FSM with 1-bit input

We use the (arbitrarily chosen) least significant bit of the *n*-bit FSM to predict the next bit in the input sequence. Then the FSM transitions to the next state based on the current state and the true value of the input bit. In other words, the FSM implements a transition table like the one shown in Figure 1. The *n* bits of current state are concatenated with the input bit to form an address (or index) to determine which *n*-bit state to transition to. As the boxed-in letters in the figure illustrate, the transition table holds $n \times 2^{n+1}$ independent bits, yielding $2^{\wedge}(n \times 2^{n+1})$ possible *n*-bit FSMs. Whereas not all bit combinations result in meaningful FSMs (*e.g.*, there are redundancies and not all FSMs can reach all states), the number of possibilities grows super-exponentially with *n*. The ILCS framework is ideal for searching such a large configuration space to determine well-performing FSMs.

The rest of this paper is organized as follows. Section II summarizes related work. Section III presents the ILCS framework, explains how to use it, and discusses its internal operation. Section IV introduces the supercomputers we used for evaluation. Section V presents and analyzes the results. Section VI concludes with a summary. The ILCS framework is available at http://cs.txstate.edu/~burtscher/research/ILCS/.

II. Related Work

Several frameworks targeting different domains exist that execute serial user code in parallel. The two most closely related frameworks are MapReduce [13] and PADO [14].

MapReduce is a distributed computation framework developed at Google to process huge amounts of data while shielding the user from the many intricacies of parallel and distributed computing. The Map function takes key/value input pairs and produces a set of intermediate key/value pairs, which are sorted by their keys, and the Reduce function 'merges' all values that are associated with the same key.

MapReduce, and its open-source counterpart Hadoop [15], can be used for running the 2-opt random restart TSP heuristic or for finding good FSMs. For example, the Map function can map a random seed to a tour length by generating an initial tour based on the seed, performing the IHC steps, and returning the length of the resulting tour. The Reduce function then determines the shortest tour. Or the Map function could map a random seed to an FSM by generating a configuration based on the seed, evaluating the FSM on a provided input bit sequence, and returning the number of incorrectly predicted bits. The Reduce function determines the best-performing FSM.

Solving ILS problems in this or a similar manner takes advantage of several features of MapReduce, including the scalability, design simplicity, load balancing, and distributed automation. However, other features are superfluous for ILS algorithms and give rise to substantial overhead. Since MapReduce is designed for huge datasets requiring large numbers of reductions, the result pairs from the Map stage are transferred to the Reduce stage via secondary storage, which is unnecessary for iterative local searches. Moreover, ILS algorithms only need a single reduction over all the map results rather than many reductions for different keys, making the Reduce functionality overly general and slow if it is not internally parallelized. For non-random restart heuristics such as tabu search [16] and genetic algorithms, the MapReduce framework would have to be invoked repeatedly, resulting in startup overhead. The only termination criterion in MapReduce is the completion of all work, making it difficult to use in real-time environments. Al-

so, Hadoop currently does not support GPUs. However, there are projects such as MARS [17] that provide MapReduce functionality for GPU clusters.

PADO is a population-based (multiple islands) metaheuristic parallelization framework with partially ordered knowledge sharing consisting of two components. The frontend is based on the Java Opt4J framework [18], and the backend is the Cyber-application framework [19], which supports both shared-memory and distributed-memory parallelism. In PADO, the user specifies the problem using the Opt4J interface by expressing an algorithm in terms of genotypes, phenotypes and objectives and then implementing the solver using a creator, decoder and evaluator. In this interface, a genetic TSP heuristic can, for example, be expressed as follows. The phenotype would be a permutation of the cities, the genotype is a particular encoding of a tour, and the fitness of an individual is the tour length. PADO can tackle a large number of problems with this model. It is scalable and robust due to its ordered knowledge sharing and loosely coupled island model. Also, PADO supports flexible termination criteria, including runtime, generation, and convergence ratio. However, because PADO targets population-based optimization, the fixed interface can be limiting for single-state local searches, such as hill climbing or tabu search, thus reducing its applicability to a subset of the iterative local search methods. Additionally, PADO's cyber-application framework also does not currently support GPUs.

In summary, both MapReduce and PADO can be used to implement iterative local searches. However, due to their much broader target domains, they include many features that are not needed for ILS algorithms. These extra features incur overhead and may complicate the implementation. Neither PADO nor Hadoop support GPUs. PADO uses Java, which is not typically available on HPC systems. However, there exist HPC versions of MapReduce such as MapReduce-MPI [20].

HTCondor [21], a job submission batch system, also shares some commonalities with our ILCS framework. HTCondor focuses on workload management and distribution with the goal of using resources on compute nodes that would otherwise be idle. It employs a Classified Advertisements (ClassAd) mechanism for flexible and dynamic resource matching. This mechanism gives the compute nodes the ability to specify the type of work they can accept, allowing the system to dynamically distribute work to a large range of architectures and environments. Similar to our framework, HTCondor can take advantage of accelerators and of compute nodes with different types and numbers of CPUs and GPUs. While HTCondor offers many other features that are beyond the scope of ILCS, such as job scheduling and prioritization as well as multiuser support, the primary distinction between it and our framework is that HTCondor delivers a High Throughput Computing (HTC) environment whereas ILCS offers a High Performance Computing (HPC) environment. In particular, HTCondor does *not* parallelize any code. Instead, it executes multiple serial and/or already parallelized user jobs concurrently.

Though not designed as frameworks and therefore not directly related to our work, we also want to briefly mention some parallel GPU implementations of TSP heuristics, many if not all of which could be used in the ILCS framework. (To the best of our knowledge, there are no public GPU implementations for FSM configuration-space evaluation.) One such TSP implementation by Fujimoto and Tsutsui makes use of a genetic algorithm with an order crossover operator and a 2-opt local search [22]. The authors report a 24.2-fold speedup relative to the corresponding CPU algorithm for problem instances with up to 512 cities. Another GPU implementation resulted in speedups of up to 6.02 using a decomposition of the 3-opt procedure and the associated data structure on problems ranging from 100 to 3038 cities [23]. A paper by Van Luong *et al.* proposes a guideline to design and implement general GPU-based multi-start local search algorithms and reports up to a 12-fold speedup. The authors characterize local search heuristics as solution-level, iteration-level or algorithmic-level parallel models. They illustrate these models by re-designing hill climbing, tabu search, and simulated annealing for GPUs [24].

III. The ILCS Framework

The ILCS framework requires the user to either supply serial CPU C code or single-GPU CUDA code. Ideally, both are provided on heterogeneous systems for best performance.

A. CPU Interface

The CPU code implements the following interface.

```
size_t CPU_Init(int argc, char *argv[]);

void CPU_Exec(long seed, void const
  *champion, void *result);

void CPU_Output(void const *champion);
```

The first function's purpose is to perform initialization. It has the same signature as the main function in C programs. Its arguments are passed verbatim from the command line used to invoke the framework. It returns the size in bytes of a user-defined data structure for recording a search result. The framework's only restriction on this data structure is that it starts with a field of type `long` that records the quality of the search result. The CPU_Init function is called once on each compute node before any calls to CPU_Exec are made.

The CPU_Exec function is repeatedly invoked with different seeds. Based on the seed (and the current champion, depending on the heuristic used), it generates a solution and then improves it until a local optimum is reached. The function returns the local optimum through the location pointed to by the third argument. We use this approach rather than a return value so that the system can handle the memory allocation (using malloc's default alignment) and, more importantly, the reuse of the return data structure. The framework keeps track of the champion by inspecting the quality field of the returned solution and updating the champion if necessary. It automatically spawns an OpenMP thread for each detected CPU core (including SMT or hyperthreading cores). Each thread continually calls CPU_Exec to evaluate seeds with the goal of keeping all available CPU cores busy.

The CPU_Output function is periodically called by the main thread to output (*e.g.*, print or save) the current champion.

B. GPU Interface

The GPU interface is very similar to the CPU interface except for one additional parameter and return value. The GPU code implements the following host functions.

```
size_t GPU_Init(int argc, char *argv[]);
long GPU_Exec(long seed, long stride,
  void const *champion, void *result);
void GPU_Output(void const *champion);
```

The first function again performs initialization. It has the same prototype as its CPU counterpart. GPU_Init is called once for each detected GPU. A different GPU is selected as the default device before each call.

The GPU_Exec function is then repeatedly called with different seeds (and information on the current champion). However, rather than evaluating a single seed, which would be inefficient on a massively parallel device like a GPU, it evaluates multiple seeds. The seeds are computed as follows.

$seed_k = seed + k*stride,$ where $k = 0, 1, 2, \ldots, n-1$

The implementer is free to choose the value n but has to inform the framework about how many seeds were evaluated by returning n from the function call. The remaining parameters are identical to their CPU counterparts. The framework spawns additional OpenMP threads, one for each detected GPU, that repeatedly invoke GPU_Exec for the associated device. The goal is to keep all available GPUs busy.

The GPU_Output function is intermittently called by the master thread to record the current champion. Hybrid CPU/GPU code exclusively uses the CPU_Output function.

C. Sample User Code

We illustrate how to utilize this interface on a very simple CPU code fragment. Its primary purpose is to demonstrate how the user-defined data structure `DS` can be set up and used.

```
struct DS {
  long quality;  // lower is better
  // other fields
};

long map(long seed) {
  long result = func(seed);  // perform ILS
  return result;
}

size_t CPU_Init(int argc, char *argv[]) {
  return sizeof(struct DS);
}

void CPU_Exec(long seed, void const
    *champion, void *result) {
  ((struct DS*)result)->quality = map(seed);
  // update other fields of result
}

void CPU_Output(void const *c) {
  if (c != NULL) {
    printf("%ld", ((struct DS*)c)->quality);
    // print or save other fields of c
  }
}
```

D. Code Restrictions

So as not to interfere with the framework's operation, certain restrictions are imposed on the user code. For instance, the CPU code must be serial and cannot include OpenMP pragmas or MPI calls. Global variables are allowed as long as they are only read in the CPU_Exec function. Similarly, the GPU code must be genuine single-GPU code that does not include calls to cudaSetDevice. Global device variables are allowed but global host variables are not. Instead, the GPU_Init function should transfer any needed information to the GPU. Both the CPU and the GPU code should be deterministic so that the same answer is always computed for a given set of arguments.

If these restrictions are violated, the program may not execute properly, may produce incorrect output (not necessarily in every run), or may run at a reduced performance level. For example, using OpenMP pragmas might yield an unbalanced work distribution, writes to global variables might cause data races, and switching to a non-default GPU might result in an unavailable device and the termination of a handler thread.

E. Sample Applications

Our framework can be used to implement many common iterative local search heuristics. Here we discuss a few such heuristics and their interaction within the framework.

- N-opt random restart: The user code generates a starting permutation based on the seed, computes a local optimal solution, and returns the solution.
- Genetic with local search: The user code starts with a random permutation based on the seed, climbs to a local optimum, performs a crossover with the champion to perturb the state, and again climbs to a new local optimal solution. The code returns this solution [22].
- Chained Lin-Kernighan: Based on the seed, the user code applies a random or random-walk kick to the current champion, computes a local optimal solution from the result of the kick, and returns the solution.

These and similar heuristics are often used in combinatorial optimization problems. Subset-selection in regression or feature selection [25], an NP-hard problem, is another example domain that can benefit from our framework. Here, the user code would generate a binary sequence from the seed representing the presence/absence of a regressor/feature. Then the code iteratively adds and removes a regressor/feature, based on their respective qualities (*e.g.*, R^2), until it reaches a local optimal solution, which it returns. This popular stepwise approach is similar to n-opt. Notably, genetic algorithms have proven useful for subset selection in linear regression, which could also be applied to dimensionality reduction for discriminant analysis, semi-parametric mixture model density estimation, and reduced kernel estimators [26].

F. Internal Operation

The ILCS framework starts executing one MPI process per compute node (the master thread). It queries the number of CPU cores and GPUs present in each node. Then it calls CPU_Init once and GPU_Init for each GPU. Next it forks a worker thread for each detected CPU core as well as a handler

thread for each GPU. These threads repeatedly call the respective Exec function and record the result. We oversubscribe the threads because the GPU handler threads are expected to sleep most of the time while they wait for the GPU code to finish.

The master thread handles all MPI communication and also sleeps most of the time. Once the worker threads are running, its primary job is to scan the results of the workers to find the best solution computed so far (*i.e.*, the local champion). This information is then globally reduced to determine the current system-wide champion. Node 0 outputs this information. Then the master threads sleep for a while before repeating their task.

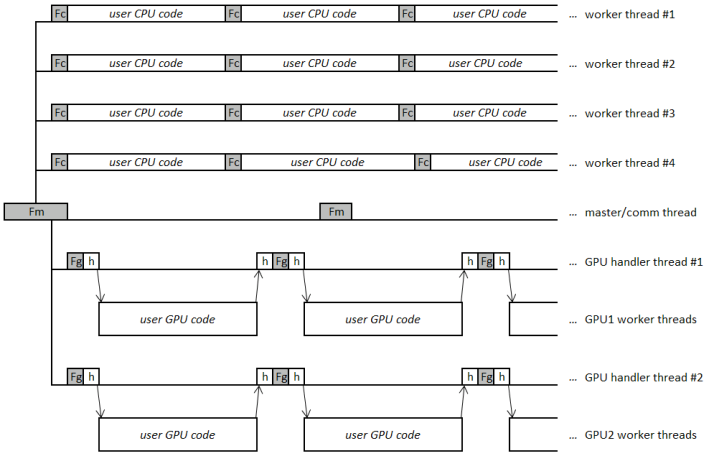

Figure 2: Threads and thread activity in the ILCS framework: Fc = framework CPU code, Fg = framework GPU code, Fm = Framework master code, h = user host code for accessing the GPUs

Figure 2 illustrates the operation of the ILCS framework on a node of a hypothetical system with four CPU cores (without hyperthreading) and two GPUs. The following happens on each node of the system. First, the master thread starts four worker threads (one per CPU core) that repeatedly call the user's CPU code with different seeds and record the results. In addition, the master thread starts two GPU handler threads (one per GPU) that repeatedly call the user-provided GPU host code with different seeds and gather the results. The host code in turn invokes the user's GPU code and sleeps while waiting for the GPU kernel to finish. Then the master thread goes to sleep. It awakens periodically to communicate with the master threads of the other nodes to determine the current global champion.

Based on the number of compute nodes, the framework assigns non-overlapping ranges of unique seeds to each node. The CPU threads work their way up from the bottom of the range while the GPUs work their way down from the top of the range. This approach is similar to how the stack and heap grow towards each other and was chosen to achieve a balanced workload independent of the ratio of the CPU-to-GPU performance. It also works if either the GPUs or CPUs are not used.

Figure 3 illustrates how the seeds are distributed on the example of a four-node system with four CPU cores and two GPUs per node. First, the seed range (0 through 2^{64}-1) is evenly distributed over the four nodes. Within each node, the four CPU worker threads (labeled a, b, c, and d) get values from the bottom of their node's seed range, assigned in round-robin fashion. The two GPUs (labeled 1 and 2) are assigned values from the top of their node's seed range. In this case, GPU1 gets chunks of odd numbers and GPU2 chunks of even numbers.

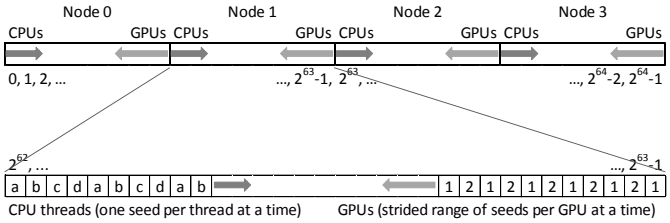

Figure 3: Seed distribution

In ILS algorithms, it is often unknown which seeds are good, so any distribution of seeds that avoids duplicates is a priori equally good. Users can employ the seeds to generate other values and distributions. As long as this mapping is injective, the independent searches will not explore overlapping regions. Our TSP and FSM codes use the seeds provided by the framework to initialize a random-number generator, so the actual values that the codes utilize are not piecewise sequential. We found the most common elements among ILS algorithms to be the use of random seeds and a champion solution, which is why we provide both in our framework.

As it is unlikely that even the largest supercomputer will be able to scan the entire 64-bit seed range in a reasonable amount of time, the framework has to decide when to terminate the search. Because ILS algorithms typically improve the result quality rapidly in the beginning but then gradually plateau out as the quality approaches the optimal solution, the framework terminates the search when the quality has not improved over a certain period of time. The default value for this timeout is 20 seconds. Note that this termination decision, which is based on an MPI_Allreduce, and all other components of the framework require no centralized entity that might impact scalability.

Users can easily update the timeout value in the framework's header file. This is also where the user selects how frequently the Output function is called, whether the framework should run in single-node mode or use MPI, and whether only CPU, only GPU, or both types of code should be used.

IV. Experimental Methodology

A. HPC Systems

We evaluated the ILCS framework on Keeneland at NICS as well as on Ranger and Stampede at TACC. Table I provides pertinent information about the three supercomputers.

Keeneland is an HP cluster with dual 8-core Intel Xeon E5-2670 processors and three NVIDIA M2090 GPUs per node. The Fermi-based GPUs each have 512 CUDA cores in 16 streaming multiprocessors. Ranger is a Sun cluster with four quad-core AMD Opteron (Barcelona) processors per node.

Stampede is a Dell cluster with two 8-core Intel Xeon E5-2680 processors per node. A few of the nodes contain GPUs, but at the time of this writing, not all GPUs were operational in this brand new system. We also could not exploit the MIC accelerators as symmetric processing was not yet enabled.

Table I: System Information

system	*compute nodes*	*CPUs*	*CPU cores*	*CPU clock frequency*	*GPUs*	*GPU cores*	*GPU clock frequency*
Keeneland	264	528	4,224	2.6 GHz	792	405,504	1.3 GHz
Ranger	3,936	15,744	62,976	2.3 GHz	-	-	-
Stampede	6,400	12,800	102,400	2.7 GHz	128*	n/a	n/a

B. *Software and Compilers*

We compiled and linked the framework, the TSP code, and the FSM code on the three systems with the following compilers and flags. On the Keeneland system, we use nvcc 4.2 with '-O3 -arch=sm_20 -use_fast_math' and icc 12.1.5 with '-O3 -xhost -openmp'. On the Ranger system, we use icc 10.1 with '-O3 -xW -openmp'. On the Stampede system, we use icc 13.0.1 with '-O3 -xhost -openmp'.

To obtain the results presented in this paper, we instrumented the framework and user code to time itself and to count the number of moves or transitions evaluated, respectively. The timer is started by the master thread after an MPI barrier at the point where the OpenMP threads are forked. It is stopped just before the master thread prints the final statistics and terminates. Note that we only evaluated the instrumented code to avoid having to rerun every experiment without instrumentation. We expect the uninstrumented code to be slightly faster.

We use O'Neil *et al.*'s CUDA TSP solver [27], from which we extracted a serial C version for the CPUs. We use their TSP implementation for the GPU but with Rocki and Suda's optimization to support problem sizes above 110 cities [28] as well as some modifications to fit the code into our framework. We run these codes on four successively larger datasets from TSPLIB [29] that are relatively difficult for their size [30]. They are kroE100, ts225, rat575, and d1291. The values in the names represent the number of cities.

We wrote the FSM code from scratch and use it to evaluate the configuration space of 3-, 4-, 5-, and 6-bit FSMs as illustrated in Figure 1. We use a 720,320-bit long confidence-estimation trace from a load-value predictor as input [11].

V. Results

A. *Performance*

Table II lists the largest configuration we tested on the four systems along with the resulting framework performance on the TSP code in trillion (10^{12}) moves evaluated per second. Figure 4 shows the same results in graphical format.

On Stampede, the framework exceeds 12.2 trillion tour evaluations per second on the ts225 input, highlighting the tremendous potential of using parallelism for local search problems. On the three larger inputs, Stampede outperforms the old Ranger system even though the latter uses twice as many CPU cores. On the GPU-accelerated Keeneland cluster, the performance tends to increase with larger problem sizes whereas on the CPU-only systems the performance tends to drop off for larger inputs. This is why Ranger, which has the most CPUs, yields the best performance on the smallest input and Keeneland, using nearly 200,000 GPU cores, provides the highest performance on the largest input.

Table II: Best performing system configuration we tested and number of TSP moves evaluated per second (in trillions)

system	compute nodes	total CPUs	total GPUs	total CPU cores	total GPU cores	kroE100 Tmoves/s	ts225 Tmoves/s	rat575 Tmoves/s	d1291 Tmoves/s
Keeneland	128	256	384	2048	196,608	3.392	4.577	5.176	4.610
Ranger	2048	8192	0	32768	0	10.754	10.363	7.427	1.683
Stampede	1024	2048	0	16384	0	10.630	12.239	10.819	2.502

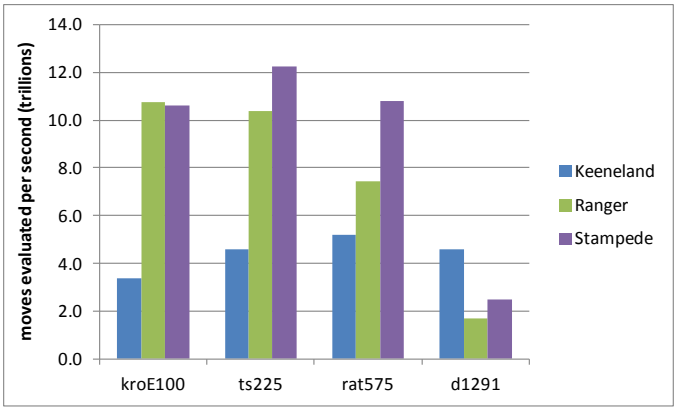

Figure 4: Number of TSP moves evaluated per second with the largest evaluated system configuration

This performance increase and decrease is a consequence of a key implementation difference between the CPU and the GPU code. The CPU code is based on a matrix that stores the distance between every city pair. Since the matrix size grows with the square of the number of cities, distance lookups tend to miss in the CPU caches for large inputs, thus lowering performance. In contrast, the GPU code is based on an array of city coordinates, which requires the (repeated) calculation of the distance between city pairs but only grows linearly with the problem size. In fact, the coordinates fit into the GPU's shared memory (a software-controlled data cache) for all four problem sizes. Due to the high frequency of short-running GPU kernels for small inputs, calling, initialization, and handler-thread overheads significantly lower the performance of the framework for the smallest input. To improve performance, a matrix-based GPU implementation [27] combined with larger seed-range chunks should be used for small inputs, and an array-based CPU implementation should be used for large inputs.

Table III lists the largest configuration we tested along with the resulting framework performance on the FSM code in trillion transitions evaluated per second. Figure 5 shows the same results in graphical format.

Table III: Best performing system configuration we tested and number of FSM transitions evaluated per second (in trillions)

system	compute nodes	total CPUs	total GPUs	total CPU cores	total GPU cores	3-bit FSM Ttrans/s	4-bit FSM Ttrans/s	5-bit FSM Ttrans/s	6-bit FSM Ttrans/s
Keeneland	128	256	384	2048	196,608	21.532	21.050	20.670	12.435
Ranger	2048	8192	0	32768	0	9.837	9.839	9.824	9.688
Stampede	1024	2048	0	16384	0	6.551	6.543	6.530	6.654

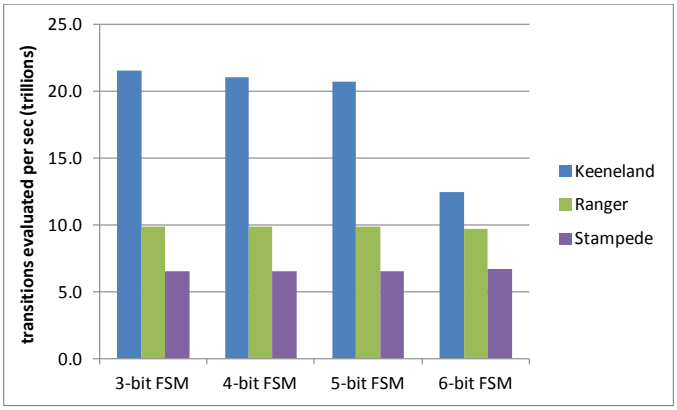

Figure 5: Number of FSM transitions evaluated per second with the largest evaluated system configuration

On Keeneland, the framework reaches over 20 trillion FSM transitions per second on the three smaller FSM sizes. Both Ranger and Stampede result in quite stable performance on all four inputs. But on Keeneland, the throughput for the largest FSM is substantially lower. The reason for this performance drop is that the transition tables are stored in the GPUs' 48 kB on-chip shared memory, which reduces the number of thread blocks that can simultaneously run in each streaming multiprocessor to one for the largest input. Nevertheless, the GPUs contribute a tremendous amount of performance, as the single-node results in Table IV reveal. On Keeneland, the three GPUs provide over 96% of the node performance on the three smaller inputs and over 93% on the largest input. Note, however, that it takes each superscalar CPU core only 2.5 ns (6.7 cycles) on average to evaluate one FSM transition and each GPU core about 10 ns (12.5 cycles) on the three smaller inputs.

Table IV: Number of FSM transitions evaluated per second on one node

system	compute nodes	total CPUs	total GPUs	total CPU cores	total GPU cores	3-bit FSM Gtrans/s	4-bit FSM Gtrans/s	5-bit FSM Gtrans/s	6-bit FSM Gtrans/s
Keeneland	1	2	3	16	1,536	169.241	165.620	163.366	97.474
Ranger	1	4	0	16	0	4.807	4.800	4.799	4.756
Stampede	1	2	0	16	0	6.420	6.422	6.420	6.531

The single-node TSP results are presented in Figure 6. Stampede uses a later generation of CPUs and a higher clock speed than Ranger, which is why Stampede's nodes are much faster than Ranger's. Keeneland's compute nodes are the fastest overall because of the GPUs. Nevertheless, it should again be noted that the CPUs are very efficient. On the ts225 input, it takes each Stampede core only 3.6 machine cycles on average to evaluate a tour alternative. This high speed is possible because the code only evaluates and compares the change in tour length due to an 2-opt move, which makes the amount of computation per move small and independent of the input size. The GPU code, in contrast, has to first compute four distances between cities before it can evaluate an 2-opt move, which is why it takes 46 cycles on average even on the most efficient input.

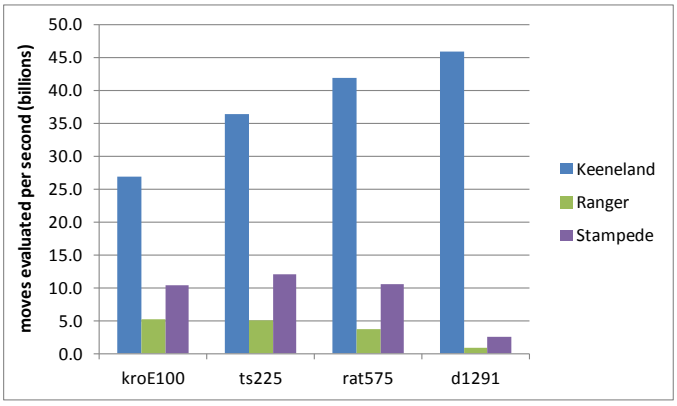

Figure 6: Number of TSP moves evaluated per second on one compute node

B. *Scaling*

Figure 7 displays the TSP node scaling on Ranger on a log-log plot. The results for the kroE100 input are mostly hidden 'behind' the results for the ts225 input. The results for Stampede (not shown) are very similar except for higher absolute values. Figure 8 shows the node scaling on Keeneland.

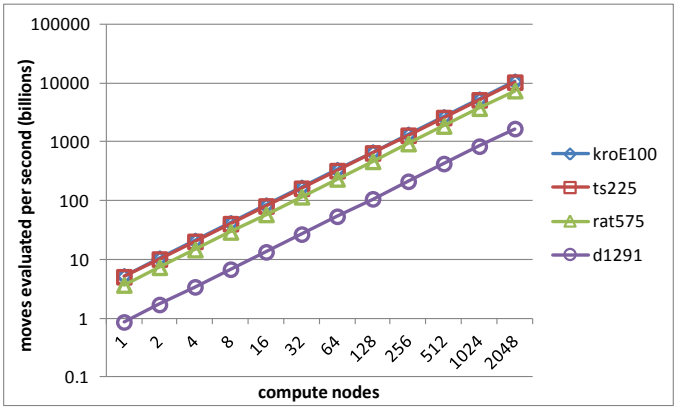

Figure 7: Number of TSP moves evaluated per second on Ranger using different numbers of compute nodes

The ILCS framework scales very well as indicated by the parallel efficiency, *i.e.*, the deviation from linear speedup relative to the single-node performance. On any of the four TSP inputs, the efficiency does not drop by more than 1% on Stampede, 5% on Ranger, and 7% on the three smaller inputs on Keeneland when scaling to the node counts listed in Table II. On the d1291 input, Keeneland incurs up to a 21% loss in efficiency. The reason for the relatively poor scaling on this input

is that the termination threshold is too short for the assigned seed-range size. In particular, the time the system waits for all GPU threads to finish once the termination decision has been made amounts to a third of the overall runtime. During this time, the parallelism decreases as the CPU worker threads and the first two GPUs stop processing, lowering the efficiency. On the other inputs, this overhead is much smaller because the waiting time represents only a small fraction of the overall execution time. Hence, the efficiency on the largest input can likely be improved with a longer termination threshold.

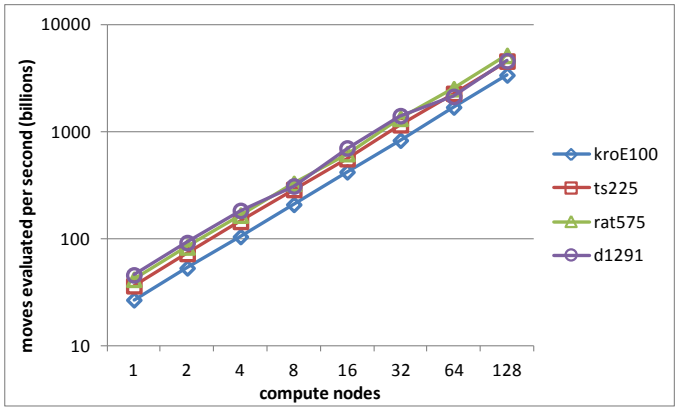

Figure 8: Number of TSP moves evaluated per second on Keeneland using different numbers of compute nodes

Nevertheless, these results demonstrate that the ILCS framework generally scales very well over several orders of magnitude. Clearly, the infrequent MPI_Allreduce, which is performed once per four seconds and is the only inter-node communication, does not significantly affect the scalability.

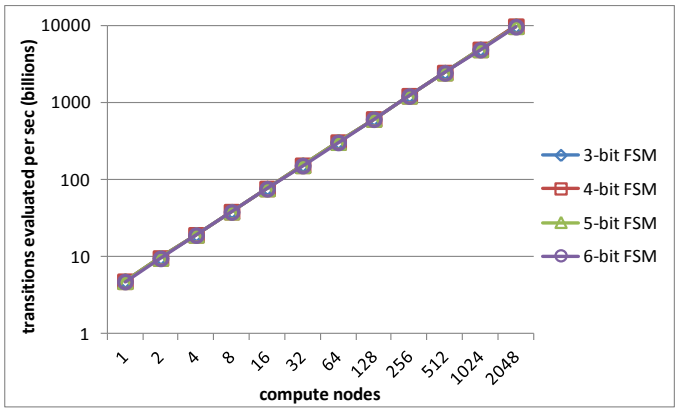

Figure 9: Number of FSM transitions evaluated per second on Ranger using different numbers of compute nodes

Figure 9 shows the FSM node scaling on Ranger on a log-log plot. The results for all four inputs overlap completely. On this code, the framework scales perfectly, *i.e.*, the parallel efficiency drops by only 0.6% when going from 1 to 2048 nodes. Aside from the absolute performance, the results for Stampede and Keeneland look almost identical (not shown), except on Keeneland the performance of the 6-bit FSM is noticeably lower than that of the three smaller FSMs, as discussed before. On Stampede, the efficiency drops by no more than 0.7% and on Keeneland by no more than 1.2% when scaling to 1024 and 128 nodes, respectively.

We illustrate the TSP intra-node scaling on the example of Stampede in Figure 10. Note that this figure uses linear axes. The parallel efficiency relative to the performance with one worker thread (and a master thread) is 98.9% or better on all inputs. These results show that the master thread is rarely awake and that the oversubscription of threads is warranted.

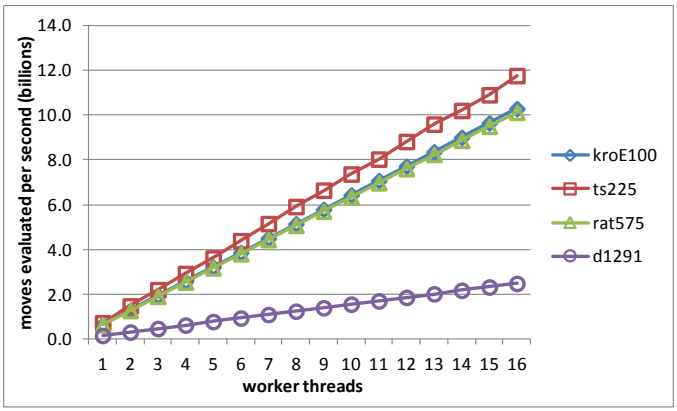

Figure 10: Number of TSP moves evaluated per second on Stampede using different numbers of worker threads in one compute node

C. *Quality*

Whereas it is beyond the scope of this work to improve the TSP code per se, for completeness we also provide results on the quality of the solutions. It should be noted, however, that the runtime and champion quality of ILS algorithms generally depend on luck, *i.e.*, how quickly a 'good' seed is encountered.

Table V: Runtime in seconds until termination, total number of evaluated seeds, and final champion tour length over optimal tour length on the largest system configuration

inp	system	runtime	seeds evaluated	error
kroE100	Keeneland	24.5	166,162,849	0.00%
	Ranger	24.3	522,364,458	0.00%
	Stampede	24.0	509,523,246	0.00%
ts225	Keeneland	24.8	17,396,687	0.00%
	Ranger	24.4	38,704,838	0.06%
	Stampede	24.0	45,054,888	0.06%
rat575	Keeneland	63.0	2,970,364	6.48%
	Ranger	37.1	2,511,527	6.44%
	Stampede	28.5	2,811,434	6.44%
d1291	Keeneland	71.7	228,182	5.02%
	Ranger	28.2	32,768	6.45%
	Stampede	38.0	65,536	6.41%

Table V presents the TSP runtime in seconds on the largest system configuration we tested (see Table II for the actual configuration), the total number of distinct seeds evaluated, and the quality of the final champion tour in terms of how much longer it is than the truly optimal tour, as computed by the exact solver Concorde [31]. Note that the step size is four seconds and the termination threshold is five steps, which are the default values.

Concorde is an example of an exact solver that, as discussed earlier, provides no useful solution while it computes. Whereas it works well on the four inputs we use, it has not terminated on some inputs and takes very long on others. For instance, it takes days to solve the d2103 input. Also, there is no description of what problems will result in substantial runtime.

For the smallest input, our three systems find the optimal solution almost right away and then keep running for five more steps (20 seconds) until the framework terminates the search. Clearly, the chosen termination threshold is too large for small TSP problems. For the other inputs, some of the searches do not find the optimal solution, but each system comes within about 6.5% in approximately half a minute to a minute of runtime. Note that additional experiments with larger termination thresholds resulted in better solutions (not shown).

Keeneland is able to find a better (in fact optimal) solution for ts225 with fewer evaluated seeds than the other two systems. The reason for this behavior is twofold. First, due to the presence of the GPUs, it evaluates seeds early on that the other systems are not evaluating, which may, by pure chance, lead to a better solution. Second, the CUDA code uses a different random number generator to create the initial tours than the C code, which might also lead to finding a good solution faster.

The number of evaluated seeds decreases with larger problem sizes as there are more 2-opt moves to consider and more IHC iterations to perform per seed. In fact, each Ranger core evaluates only one seed for the d1291 input, indicating that a substantially larger termination threshold should be chosen for this input.

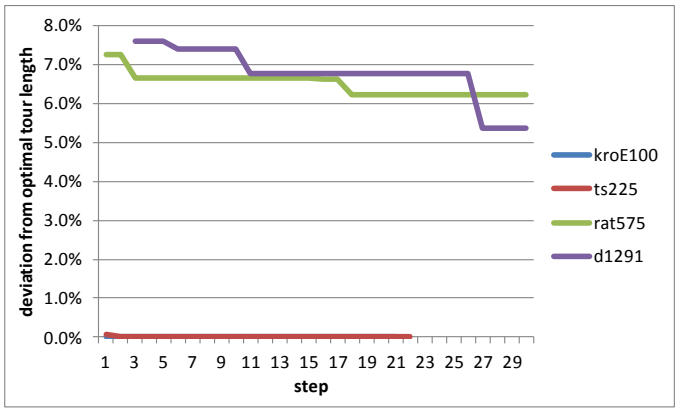

Figure 11: Champion tour length over optimal tour length after each one-second step with 128 nodes on Keeneland

Figure 11 demonstrates, on the example of the largest Keeneland configuration we tested but with one-second steps, how the champion quality improves over time. The results from the other systems follow the same pattern and are not shown. The curve for the d1291 input only starts at step 3 because it takes almost three seconds for the first result to be returned to the framework. The figure is cut off at step 30 for clarity even though the d1291 run extends to step 47 without achieving any further improvement.

The results in Figure 11 exhibit the step-wise improvement that is typical for ILS algorithms. Initially, the champion tends to improve often, but then the improvements become less frequent as more and more seeds need to be evaluated to beat the current champion.

Figure 12 shows the quality of the champion for different node counts after 6 steps (24 seconds), *i.e.*, after almost identical runtimes. We only provide results for Stampede as the results for the other systems are qualitatively very similar.

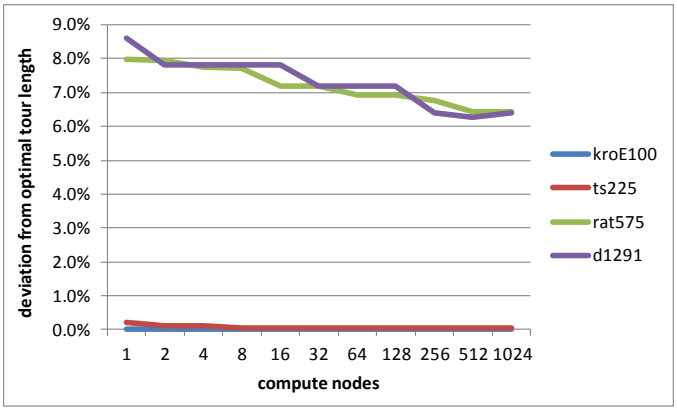

Figure 12: Champion tour length over optimal tour length after six steps for different node counts on Stampede

For the smallest input, even a single compute node with 16 CPU cores finds the optimal solution. Whereas the optima for the three larger inputs are not found, Figure 12 illustrates that larger node counts generally result in better solutions. This is not always the case, though, as can be seen on the d1291 input, where 512 nodes compute a better champion in the first 6 steps than 1024 nodes do. The reason for this anomaly is that, due to subtle timing variations, slightly different numbers of seeds have been evaluated per node after six steps, *i.e.*, the seeds evaluated on the larger system configuration are not a superset of the seeds evaluated on the smaller configuration. Note that the timeout threshold hides these variations in full runs. In general, more parallelism clearly helps to find better solutions faster.

VI. Summary and Future Work

This paper presents and evaluates the ILCS parallelization framework for (heterogeneous) HPC systems. It is designed for iterative local searches with the goal of providing a quick turnaround for implementing an ILS algorithm, running it in parallel, and obtaining answers on whatever size machine the user has access to. The framework records the currently best solution, called the champion, every few seconds so that the search

can be stopped at any time. It handles the MPI communication between the compute nodes, provides OpenMP and multi-GPU support within nodes, and is completely decentralized for maximal performance and scaling. The ILCS framework is available at http://cs.txstate.edu/~burtscher/research/ILCS/.

Using an iterative hill-climbing heuristic TSP solver and an FSM configuration-space exploration as examples, we demonstrate that the ILCS framework runs on systems with different numbers and types of CPUs and GPUs, scales to 2048 compute nodes with just a few percent loss in efficiency, searches over 12.2 trillion TSP tour alternatives per second on a machine with 2048 CPUs and evaluates over 21.5 trillion FSM transitions per second on a system with 256 CPUs and 384 GPUs.

In future work, we want to adapt the master thread's sleep time and the termination threshold based on how long it takes for the first results to be computed. This should make ILCS' default parameters useful for a larger range of input sizes.

ACKNOWLEDGMENTS

This work was supported by NSF grants 1141022 and 1217231 as well as donations from NVIDIA Corporation and Intel Corporation.

This research was supported by an allocation of advanced computing resources provided by the National Science Foundation. The computations were performed on Keeneland at the National Institute for Computational Sciences (NICS).

The authors acknowledge the Texas Advanced Computing Center (TACC) at the University of Texas at Austin for providing HPC resources that have contributed to the research results reported within this paper.

REFERENCES

[1] http://www.top500.org/ (February 2013)

[2] H.R. Lourenco, O.C. Martin, and T. Stutzle. "Iterated Local Search." *Handbook of Metaheuristics*, by G.A. Kochenberger (Ed.), pp. 321-354. Springer, 2003.

[3] R. Agarwala, D.L. Applegate, D. Maglott, G.D. Schuler, and A.A. Schaffer. "A Fast and Scalable Radiation Hybrid Map Construction and Integration Strategy." *Genome Research*, 10350-364. 2000.

[4] S. Mobaieen, A. Rabii, and B. Mohamady. "Optimal Robot Arm Movement using Tabu Search Algorithm." *Research Journal of Applied Sciences, Engineering and Technology*, vol. 4, no. 4, pp. 383-386. 2012.

[5] R. Matai, S.P. Singh, and M.L. Mittal. "Traveling Salesman Problem: An Overview of Applications, Formulations, and Solution Approaches." *Traveling Salesman Problem, Theory and Applications*, by D. Davendra (Ed.). InTech, 2010.

[6] M.R. Garey and D.S. Johnson. "Computers and Intractability: A Guide to the Theory of NP-Completeness." San Francisco: W.H. Freeman. 1979.

[7] J. Ambite and C. Knoblock. "Planning by Rewriting." *Journal of Artificial Intelligence Research*, pp. 207-261. 2001.

[8] L.S. Pitsoulis and M.G.C. Resende. "Greedy Randomized Adaptive Search Procedures." *Handbook of Applied Optimization*, pp. 168-183. Oxford University Press, 2001.

[9] D. Johnson and L. McGeoch. "The Traveling Salesman Problem: A Case Study in Local Optimization." *Local Search in Combinatorial Optimization*, by E. Aarts and J. Lenstra (Eds.), pp. 215-310. John Wiley and Sons, 1997.

[10] C. Rego and F. Glover. "Local Search and Metaheuristics." *The Traveling Salesman Problem and its Variations*, by G. Gutin and A.P. Punnen (Eds.), pp. 309-368. Kluwer Academic Publishers, 2002.

[11] S. J. Jackson and M. Burtscher. "Self Optimizing Finite State Machines for Confidence Estimators." *2006 Workshop on Introspective Architecture*. 2006.

[12] V. Uzelac, A. Milenkovic, M. Burtscher, and M. Milenkovic. "Real-time Unobtrusive Program Execution Trace Compression Using Branch Predictor Events." *International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 97-106. 2010.

[13] J. Dean and S. Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters." *6th Symposium on Opearting Systems Design and Implementation*. 2004.

[14] J. Kim, M. Kim, M.O. Stehr, H. Oh, S. Ha. "A Parallel and Distributed Meta-heuristic Framework based on Partially Ordered Knowledge Sharing." *Journal of Parallel and Distributed Computing*, vol. 72, no. 4, pp. 564-578. 2012.

[15] http://hadoop.apache.org/ (February 2013)

[16] S. Jain and M. Mallozzi. "Parallel Heuristics for TSP on MapReduce." *Brown University Technical Report*. 2010.

[17] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. "Mars: a MapReduce Framework on Graphics Processors." *17th International Conference on Parallel Architectures and Compilation Techniques*, pp. 260-269. 2008.

[18] http://opt4j.sourceforge.net/ (February 2013)

[19] M. Kim, M.O. Stehr, J. Kim, and S. Ha. "An Application Framework for Loosely Coupled Networked Cyber-physical Systems." *2010 IEEE/IFIP Conference on Embedded and Ubiquitous Computing*. 2010.

[20] http://mapreduce.sandia.gov/ (February 2013)

[21] D. Thain, T. Tannenbaum, and M. Livny. "Distributed Computing in Practice: The Condor Experience." *Concurrency and Computation: Practice and Experience*, vol. 17, no. 2-4, pp. 323-356. 2005.

[22] N. Fujimoto and S. Tsutsui. "A Highly-Parallel TSP Solver for a GPU Computing Platform." *Numerical Methods and Applications, Lecture Notes in Computer Science*, vol. 6046/2011, pp. 264-271. 2011.

[23] A. Delévacq, P. Delisle, and M. Krajecki. "Parallel GPU Implementation of Iterated Local Search for the Travelling Salesman Problem." *Learning and Intelligent Optimization*. 2012.

[24] T. Van Luong, N. Melab, and E.G. Talbi. "GPU-based Multi-start Local Search Algorithms." *Learning and Intelligent Optimization, Lecture Notes in Computer Science*, vol. 6683/2011, pp. 321-335. 2011

[25] H. Vafaie and I.F. Imam. "Feature Selection Methods: Genetic Algorithms vs. Greedy-like Search." *International Conference on Fuzzy and Intelligent Control Systems*. 1994.

[26] B.C. Wallet, D.J. Marchette, J.L. Solka, and E.J. Wegman. "A Genetic Algorithm for Best Subset Selection in Linear Regression." *28th Symposium on the Interface.* 1996.

[27] M. A. O'Neil, D. Tamir, and M. Burtscher. "A Parallel GPU Version of the Traveling Salesman Problem." *2011 International Conference on Parallel and Distributed Processing Techniques and Applications*, pp. 348-353. 2011.

[28] K. Rocki and R. Suda. "An Efficient GPU Implementation of the Iterative Hill Climbing based TSP Solver." *24th ACM Symposium on Parallelism in Algorithms and Architectures*. 2012.

[29] http://www.iwr.uni-heidelberg.de/groups/comopt/software/TSPLIB95/tsp/ (February 2013)

[30] http://www.tsp.gatech.edu/concorde/benchmarks/bench99.html (February 2013)

[31] http://www.tsp.gatech.edu/concorde/ (February 2013)