Computer Engineering Handbook - 2nd Edition

June 9, 2006

Contents

i

Chapter 1

Parallelization of Iterative Heuristics

Sadiq M. Sait, Habib Youssef and Mohammad Faheemuddin

1.1 Introduction

Iterative heuristics such as Simulated Annealing, Genetic Algorithms, Tabu Search are stochastic optimization algorithms that have found applications in a myriad of complex problems in science and engineering. The previous chapter discussed these algorithms in detail, and elaborated on their various characteristics. However, with increasing domain complexity and sizes, inspite of their robust nature and capability, these algorithms can have very high run-times. Although there are acceleration strategies for these heuristics, these are often just parameter tweaks and are at best problem specific and often non-scalable.

Parallelization of these algorithms to achieve reduced runtime as well as possibly find better solutions has increasingly attracted attention over the years. With the ever decreasing cost-performance ratio of generic computer systems, cluster environments have become a norm in both academia and industry. Also as network technology keeps chipping away at the latency constraint, these distributed computing environments offer very promising and competing alternatives to expensive multi-processor machines.

In this chapter, we document parallelization strategies for different popular iterative heuristics, namely Simulated Annealing, Genetic Algorithms, Tabu Search and Simulated Evolution. A detailed description of these heuristics is available in the book by Sait and Youssef [24]. A broad classification is presented for the different parallel models followed by an enumeration and description of relevant strategies. The focus is on parallel approaches for cluster environments. The chapter concludes with further directions in contemporary research.

1.2 Parallelization Issues

Distributed computing today offers extensive opportunities in collectively utilizing computing power for performance gains. The concept behind parallelization is often to allocate fairly independent sections of the algorithm to individual processors, collect the results, do the needful and continue onwards with the next iteration. However, this approach, though simple and straightforward doesn't scale well - Amdahl's law is soon to set in - the speedup achievable is limited to the non-parallelizable, serial fraction of the algorithm. This is especially aggravated by the inherently sequential nature of various heuristics such as Simulated Annealing, Tabu Search and Simulated Evolution.

Parallelization strategies can be broadly classified into the following three approaches:

1. Low-Level Parallelization (Type 1): Also known as *Move Acceleration*, in this approach the computation-intensive operations within a single iteration are distributed among nodes. Such approaches seek to divide the workload for each iteration across multiple processors, and as a consequence, leave the algorithm characteristics unaffected.

2. Domain Decomposition (Type 2): In this approach, the problem state space is divided and assigned to different processors. Also known as *multiple-trials* parallelism, this strategy can involve either partitioning the single solution across available processors or distributing the search space by assigning processors sets of moves or perturbations. In both these cases, all nodes work with the same single copy of the solution. This usually

implies a conspicuous departure from the functionality and characteristics of the serial algorithm.

3. Multithreaded or Parallel Search (Type 3): Parallelism here is implemented as multiple concurrent exploration of the solution space using search threads with various degrees of synchronization or information exchange. Often modeled as Multiple-Markov Chains (MMC), these methods allow for increasing the variety of the search threads particularly by having different types of searches - same method with different parameter settings or even different meta-heuristics - proceeding concurrently. The distinguishing feature of this approach is that the independent processors work on their own individual solutions and can periodically communicate to cooperatively navigate the search space.

Of these various approaches, *move acceleration* is suitable only to tightly coupled, multi-processor environments, rather than cluster computing. Even on the former, Amdahl's law restricts achievable scalability. As such, the discussion here is restricted to Domain Decomposition strategies and Parallel Search models. Such cooperative parallel models are generally applicable with minor variations to almost all heuristics in general, providing impressive performance gains.

1.3 Simulated Annealing

Simulated Annealing is arguably the pioneering success story of iterative heuristics and their application in combinatorial optimization problems. Proposed in the early eighties, SA simulates the effect of a heat bath on the structure of metals [14, 6]. In the metallurgical annealing process, heated metals are cooled at a controlled rate, thereby transitioning from a higher energy state to a lower one over a period of time. A proper, controlled cooling schedule allows the metal atoms to achieve perfect crystal lattices. This process was first mathematically studied by Metropolis, et al in 1953 who established criteria to simulate how thermodynamic systems change from one energy level to another. Lower energy states are assumed to be always accepted, while acceptance of higher energy states is probabilistic. In simulation parlance, this is referred to as the Metropolis Acceptance Criterion which defines this probability given by the following expression:

$$\text{Prob(accept)} = e^{-(\frac{\Delta E}{K_B T})} \tag{1.1}$$

where K_B is the Boltzman constant and T indicates the temperature.

In Simulated Annealing, every iteration at a certain *temperature*, within the Metropolis loop comprises the following steps:

1. Perturb the current solution to create a new solution,
2. Compute the difference in the cost between the current and new solutions,
3. Decide whether to accept or reject the new solution.

Domain Decomposition Strategies: This multiple-trials parallelization approach of simulated annealing is very general and can be tailored to any particular problem instance. In this strategy, several trials (moves) are generated and evaluated in parallel, where each trial is executed by a single processor. The processors are forced to concurrently search for an acceptable solution in the neighborhood of the same current solution. In order to ensure that all processors are always working with the same current solution configuration, one has to force them to communicate and synchronize their actions whenever at least one of the trials is successful (accepted move).

Figure 1.1 is a possible parallel simulated annealing algorithm following this multiple-trials parallelization approach. Here, it is assumed that one master processor is ordering the concurrent execution and evaluation of p trials, where p is the number of processors. These new solutions are returned to the master which arbitrates between them. In case of no success, the master then orders the parallel evaluation of p new trials; otherwise, it selects the best new current solution, and updates the state of all processors. This process repeats until the defined termination criterion is reached. Also at the end of each p new trials, the master processor checks to see whether equilibrium has been reached at current temperature. If so, the algorithm parameters are updated.

An examination of the algorithm in Figure 1.1 reveals an evident overhead in commu-

```
Algorithm Parallel_SA;
          (*$S_0$ is the initial solution *)
Begin
          Initialize parameters;
          $BestS = S_0$;
          $CurS = S_0$;
                Repeat
                      Repeat
                      Communicate $CurS$ to all processors;
                      ParFor each processor $i$
                            Perturb($CurS, NewS_i$);
                            $A_i$ = Accept($CurS, NewS_i$) (* $A_i$ is true if $NewS_i$ is accepted *)
                      EndParFor
                      If Success Then
                            (* Success = ($\bigvee_{i=1}^{p} A_i = True$) *)
                            Select($NewS$);
                            If $Cost(NewS) < Cost(BestS)$ Then $BestS = NewS$;
                      EndIf
                      Until Time to update parameters;
                Until Time to stop;
                Output Best solution found
End. (*Parallel_SA*)
```

Figure 1.1: General parallel simulated annealing algorithm where synchronization is forced after each trial.

nication, where the synchronous nature forces communication after each trial. However, since the current solution will get updated only when a processor makes a successful trial, the various nodes should be allowed to proceed asynchronously till one of them achieves an acceptable move. Therefore, one can markedly improve the parallel algorithm of Figure 1.1 by making the following change. Synchronization is forced only when one of the processors performs a successful trial. In this new variation communication is minimal. Furthermore, it is a more efficient parallelization since no processor is forced to remain idle waiting for other processors with more elaborate trials to finish.

Both variations of this parallel algorithm can be implemented to run on a multicomputer or a multiprocessor machine. The parallel model assumed is a MISD or a MIMD machine. For both algorithms, it is assumed that each processor must be able to set a common variable to *True* whenever it accepts a move; then the solution accepted by the processor is communicated to a master processor which will force all other processors to halt and to properly update the current solution. Here, there are two possibilities. If the processors do not halt immediately, but rather are allowed to complete the trials that were in progress when the request to stop was received, then there could be more than one solution accepted, and therefore the master processor has to arbitrate between them, select the best, and pass a copy to each processor. The other possibility is when a processor is supposed to abort whatever activity in progress as soon as it receives a request to stop. In that case the first solution accepted by any of the processors would be the new solution of all the processors.

A valid concern here is the behavior of this parallel model with respect to the Simulated Annealing 'Temperature' parameter. In the early regime (high temperature), SA behaves close to a random search algorithm, where almost every move is accepted. This means that for the multiple-trials approach, the speed-up will be low (almost 1) at high temperatures since the processors will be forced to communicate after each trial. On the other hand, as the temperature is lowered, less and less moves are accepted, reducing by the same token the need for communication, thus allowing the p processors to concurrently be working most of the time. Therefore, in the cold regime, the speed-up will be approaching the number of processors.

Multithreaded Parallel Search Strategies: In this class of strategies, each processor runs its own self-contained, independent annealing algorithm on its own solutions, with periodic exchange of information to collectively guide the search process. Each of these search threads can either be synchronous or asynchronous. In the former, all processors periodically stop work at a defined time, and communicate information at once. In the asynchronous approach, there are no such process barriers, and all nodes are free to communicate at their own discretion. We document a recently reported adaptive variation of such an Asynchronous Multiple Markov Chains (AMMC) method here.

The basic AMMC approach is shown in the Figure 1.2, where each slave process runs its own annealing algorithm. The assigned master processor is excluded from the main computational workload and instead manages information exchange between the slaves. After each Metropolis loop, the slave returns its best achieved cost to the master, which then compares this value against the global best reached so far. If the former is better, the slave is instructed to send the entire solution, and the associated global values are updated. Otherwise, the master sends its copy of the global best solution and associated cost to the slave, which replaces its current relevant values and continues with the next iteration.

The adaptivity mechanism deals with a dynamic value of M - the number of perturbations within a single Metropolis routine, i.e., the number of moves allowed before a temperature update. Initially, during the high temperature region, where annealing approaches a random walk, M is kept very low, and is incrementally increased allowing more thorough exploration of the neighboring search space at lower, stable temperatures. This strategy was empirically found to give significantly improved results over static AMMC approaches.

1.4 Genetic Algorithms

Genetic Algorithms are a powerful domain-independent, robust search technique inspired by the Darwinian theory of evolution. Invented in the early 1970s by John Holland and his colleagues [13], GAs emulate the process of natural evolution whereby high fitness individuals survive and mate thus passing on their characteristics to offsprings. This adaptive algorithm works with a population of solutions called chromosomes, which are encoded as strings. Each of these solutions represent a point in the solution space, and in each iteration, referred to as a generation, a new set of strings that represent solutions (called offsprings) is created by crossing some of the strings of the current generation [12]. Occasionally new characteristics are injected to add diversity. In this manner, GAs combine information exchange along with *survival of the fittest* among individuals to conduct their search for the optimum solution.

In GAs, a number of initial solutions are generated as string-based chromosomes. An equal number of offsprings are generated by selecting parent chromosomes, two at a time, and implementing a crossover mechanism that copies substrings of solutions from both parents into the offspring. The generated solutions are evaluated and a selection operator decides which solutions are passed on to the next generation as parents.

Domain Decomposition Strategies: As GAs work with a population of solutions, they lend themselves to straightforward workload division strategies. Such move-acceleration strategies distribute the population among processors in every generation for fitness calculation and

```
**Algorithm** Parallel_Simulated_Annealing($S0, T0, \alpha, \beta, M,$
$Maxtime, my\_rank, p$)
**Notation**
 (* $S0$ is the initial solution. *)
 (* $BestS$ is the best solution. *)
 (* $T0$ is the initial temperature. *)
 (* $\alpha$ is the cooling rate. *)
 (* $M$ is the time until next parameter update. *)
 (* $Maxtime$ is the total allowed time for the annealing process. *)
 (* $my\_rank$ is rank of current process;0 for master,!0 for slaves. *)
 (* $p$ is the total number of running processes. *)
**Begin**
 T = T0;
 CurS = S0; // only master has the initial Solution
 BestS = CurS;
 CurCost = Cost(CurS);
 BestCost = Cost(BestS);
 Time = 0;
 **If** (my_rank == 0) // i.e. Master process
     Broadcast(CurS);
 **Endif**

 **If** (my_rank ! = 0) // i.e. Slave process
     **Repeat**
        Call Metropolis(CurS, CurCost, BestS, BestCost, T, M);
        Time = Time + M;
        T = $\alpha$ T;
        M = $\beta$ M;
        Send_to_Master(BestCost);
        Receive_frm_Master(verdict);
        **If** (verdict == 1)
           Send_to_Master (BestS);
        **Else**
           Receive_frm_Master(BestS);
        **EndIf**
     **Until** (Time $\geq$ Maxtime);
 **EndIf**

 **If** (my_rank == 0) // i.e. Master process
     **Repeat**
        Receive_frm_Slave(BestCost);
        Send_to_Slave(verdict);
        **If** (verdict == 1)
           Receive_frm_Slave(BestS);
        **Else**
           Send_to_Slave (BestS);
        **EndIf**
     **Until** (All Slaves are done);
     **Return**(BestS);
 **EndIf**
**End.** (*Parallel_Simulated_Annealing*)
```

Figure 1.2: Procedure for Parallel Simulated Annealing using Asynchronous MMC.

possibly even recombination[1]. The calculated fitnes values are then collected at the assigned Master which then applies the GA operators and moves to the next generation. In such approaches, also referred to as 'Global Parallel Models', the algorithm characteristics are left undisturbed as the decision processes such as population selection and mutation are done by a single processor. However, such strategies have predictable and limited scalability [5]. A more popular approach is the Multi-Deme parallel model, where subpopulations on independent nodes communicate and collectively navigate the search space. These come under the Type-3 or Multi-threaded parallelization strategies.

Multithreaded Parallel Search Strategies: Multiple-Population GAs provide a more sophisticated parallelization strategy wherein several subpopulations evolve independently on individual processors and exchange individuals periodically. This exchange of solutions is called *migration* and is a core aspect of this parallel model. Multi-population GAs are known with different names. They are referred to as Multi-deme parallel GAs (drawing on the analogy of natural evolution), Distributed GAs (as they are often implemented on distributed parallel architectures), and Coarse-grained GAs (since the computation to communication ratio is usually high). This model of parallel GAs is very popular, but also the most difficult to understand due to the effect of migration and a large number of influential parameters.

The first systematic study of parallel GAs with multiple populations was Grosso's work in the 1980s [21]. The objective was to simulate the interaction of several parallel subcomponents of an evolving population. The population was divided into five demes, each of exchanging individuals with the other after a fixed interval. The effect of migration on the search process and population convergence was documented, and the findings were later on bolstered by the further work of Grefenstette [4]. From these studies it was seen that favorable traits spread faster when the population is divided into smaller demes. However, when the demes were isolated, the rapid rise in fitness stops at a lower fitness value than with the complete population i.e., the quality of the solution found after convergence was worse. This is expected as the quality of solution is heavily dependent on the initial population size. However, the controlled movement of solutions between these subpopulations

[1]Genetic operators such as crossover, selection and mutation are often trivial in terms of runtime compared to the more computation intensive fitness calculations. As such, the runtime gain by distributing these operators may not be justifiable with the associated communication latencies.

called migration can significantly alter this behavior. The migration policy associated with a multi-deme GA can be a function of the following parameters:

1. Migration Frequency: This is defined as the time scale between the movement of individual solutions between demes. Migration Frequency is also referred to as *Epoch Length*, and the interval between migrations is called an *Epoch*.

2. Migration Rate: This is the number of individual solutions that will migrate from one deme to another per epoch. Migration Frequency and Rate are closely related, and can be either static i.e., a fixed number of solutions moving between populations after a fixed number of generations or dynamic, wherein both parameters are controlled by the rate of convergence in individual subpopulations.

3. Migrant Absorption: This parameter defines how migrants are absorbed into destination demes. Though the usual policy for identifying migrant solutions in the source demes is to select from the best, the absorption policy within the destination deme can vary. Common models advocate replacing worst solutions, replacing random solutions or roulette-wheel based probabilistic replacement.

4. Communication Topology: This defines the connectivity between individual demes. Again, this can be static, wherein movement of solutions is predefined by the network topology, or dynamic, wherein movement of migrants to and from demes depends on their population diversity, average fitness and convergence.

The tuning of these parameters has often been on an intuitive basis, with different values reported for distinct problems. In the case of migration rate and frequency, most models adopt a synchronous approach, triggering movement of solutions at periodic intervals. However, an alternative policy is to be asynchronous, wherein demes communicate only when near convergence [3, 17]. The purpose of this model is most often to prevent premature convergence by restoring diversity into the demes. It is found that there is a certain critical migration rate and frequency which allows the communicating subpopulations to achieve solutions of almost the same quality as panmictic populations. Lower values of these parameters would not allow proper mixing of solutions, thus achieving the same as isolated demes.

Higher values of migration and frequency may show no improvement, thus wasting time and resources or worse, may actually hinder population diversity among processors.

Communication topology is an important factor controlling the spread of good solutions among different demes. It defines the direction of movement of migrant solutions from one deme to another. Although a dense topology would encourage rapid spread of good solutions throughout all demes, however it might also prevent each subpopulation from following a separate evolutionary path. A more sparse connectivity will achieve certain amount of isolation between demes, thus allowing different solutions to be found and exploring different areas of the search space. Another possible classification of topologies distinguishes them into static, where the communication is already defined, and dynamic, wherein migrant destination is decided by factors such as average fitness or population diversity of candidate destination demes [17].

An extensive study of the effect of these parameters, especially of the influence of migration and population sizes was done by Cant*ú* Paz. Inspite of its complexities, this Multi-Deme approach shown in Figure 1.3 has often been favored over simplistic data distribution as in the earlier model. The initial population constructor on the master (root) processor creates the initial population which is then distributed to all non-root processors. Following this, all nodes, including the root execute the serial GA on their allocated population for a predefined number of iterations called the Migration Frequency (MF). Then each node sends a certain number of its best solutions to the root. The number of solutions sent is controlled by the Migration Rate (MR) parameter. The root determines the MR best solutions from the collective $MR * (N)$ solutions and broadcasts it to all processors. These migrants if not already present on the processors, are then absorbed into the existing population by weeding out and replacing the weakest solutions. Each processor then continues with the serial GA for another MF number of generations. Every interval between migrations, i.e., the length of time defined by MF number of generations is called an *Epoch.* The stopping criterion is a predefined number of Epochs.

It is important to note that the migrant absorption policy dictates the replacement of worst solutions with incoming migrants only if the migrants already do not exist within

```
ALGORITHM $Multi - Deme\_Parallel\_GA$
$NOTATION$
$RANK$ : $ROOT$= Root Processor designated by Rank=0
$RANK$ : $NON - ROOT$= All other Processors designated by rank>0
$RANK$ : $ANY$= All processors, including Root
$MF$= Migration Frequency
$MR$= Migration Rate
$N$= Number of Processors
$Epoch$ = Instances of Migration
$EPOCH\_MAX$ = Maximum Number of Migrations Stopping Criteria
Begin
($Multi - Deme\_Parallel\_GA$)

FOR RANK:ROOT
Initial Population Constructor
Distribute Initial Population
ENDFOR RANK:ROOT

FOR RANK:ANY
  Receive Allocated Population
ENDFOR RANK:ANY

LOOP-A
  FOR RANK:ANY
    LOOP-B
      Serial GA on Allocated Population:
        Choice of Parents
        Crossover and Offspring Generation
        Fitness Calculation
        New Population Selection
    END LOOP-B IF [Num_Iterations >= MF]
    Send MR Best Solutions and Costs to ROOT
  ENDFOR RANK:ANY

  FOR RANK:ROOT
    Collect the best MR*N solutions
    Determine best MR distinct solutions
    Broadcast MR solutions
  ENDFOR RANK:0

  FOR RANK:ANY
    Receive MR Best Solutions
    IF [Received Migrants not present in existing Population]
      Replace Worst Solutions with Received Solutions
    ENDIF
  ENDFOR RANK:ANY

END LOOP-A IF [Epoch >= EPOCH_MAX]

FOR RANK:0
  Return Best solution.
ENDFOR RANK:0

End ($Multi - Deme\_Parallel\_GA$)
```

Figure 1.3: Structure of the Multi-Deme Parallel GA.

the population. Also, logically this model could represent a fully connected topology of non-hierarchical processing elements which cooperate to determine the best MR solutions among themselves and absorb these into their existing populations.

1.5 Tabu Search

Conceptually, Tabu Search (TS) is an elegant combinatorial optimization method, which belongs to the class of local search techniques. It enhances the performance of a local search method by using memory structures, to control navigation of the search space. It uses a local or neighborhood search procedure to iteratively move from a solution x to a solution x' in the neighborhood of x, until some stopping criterion has been satisfied. In order to explore regions of the search space that would be left uncovered by the local search procedure and - by doing this - escape local optimality, TS modifies the neighborhood structure of each solution as the search progresses. The solutions admitted to N*(x), i.e. the new neighborhood, are determined through the use of special memory structures. The search now progresses by iteratively moving from one solution to another in N*(x).

One of the commmon types of short-term memory structures to determine which solutions comprise the neighborhood, is the use of a *tabu list.* In its simplest form, a tabu list contains solutions that have been visited in the recent past (less than n moves ago, where n is the tabu tenure). Therefore, solutions in the tabu list are excluded from N*(x). However, tabu lists containing attributes are much more effective, although they raise a new problem. With forbidding an attribute, probabilistically more than one solution might be matched and declared tabu. Some of these solutions that must now be avoided might be of excellent quality and may not yet have been visited. To overcome this problem, aspiration criteria are introduced which allow the overriding of the tabu state of a solution and thus including it in the allowed set. A commonly used aspiration criterion is to allow solutions which are better than the currently best known solution.

Parallel Tabu Search Taxonomy

Of the various heuristics covered, Tabu Search stands out uniquely being the only algorithm that employs memory systems to control navigation of the search space. However with increasing problem sizes TS shows rapidly increasing runtimes, and as such can benefit from intelligent parallelization efforts.

Parallel TS has drawn the attention of many researchers, especially in comparison with similar acceleration strategies applied to other heuristics. Unlike GAs however, the TS algorithm with its structure and process flow is highly sequential. The first reported studies were published in the early 90's [9, 26, 11]. Crainic et. al [8] classified the different parallel tabu search heuristics based on a taxonomy along three dimensions as enumerated below.

- The first dimension is **Control cardinality**, where the algorithm is either *1-control*, where one processor executes the search and distributes tasks to other processors or *p-control*, where each processor is responsible for its own search and communicates with other processors.
- The second dimension is **Control and communication type**, where the algorithm can either follow a *synchronous rigid* or *knowledge synchronization (KS)* approach, or it can be asynchronous *Collegial (C)*, or *Knowledge Collegial (KC).* In a synchronous operation mode the processes are forced to establish communication and exchange information at specific, explicitly defined points. In an asynchronous operation mode the processes can indpendently decide on communication depending on the global characteristics of good solutions, the search strategy, and the possible content of that communication.
- The third dimension is **Search differentiation** where the algorithm can be *SPSS (Single Point Single Strategy), SPDS (Single Point Different Strategies), MPSS (Multiple Point Single Strategy)*, or *MPDS (Multiple Point Different Strategies).*

In addition to this type of classification, a more general category based on processor communication is also used. This divides various approaches as either *Synchronous* or *Asynchronous.* In the former, various processors working with the same solution, communicate in a synchronous manner, where the managing processor orchestrates the activities of all others. In asynchronous strategies, each processor communicates independently of other nodes using either the master-slave or peer-to-peer model.

Synchronous Parallel Tabu Search:

In this approach, the master is primarily in-charge of controlling movement in the search process, while the slaves are used for executing their assigned workload. Depending on the variants of this strategy, slave processors may start with either the same or different initial solution. After searching its allocated part of the current neighborhood, each slave process reports its best move back to the master. The master process selects the best among these, subject to tabu conditions. If the stopping criteria are met then the search stops; otherwise the master determines a new set of moves and distributes them among the slaves which continue with the search.

A more detailed view of this approach, from the both perspectives of the master and slave processors is given in Figure 1.4 and Figure 1.5 respectively.

```
Algorithm MasterProcess;
Begin
        Initialize parameters and data structures;
        $S_0$ = Initial solution;
        $BestS = S_0$;
        $CurS = S_0$; /* Current solution */
        Send $CurS$ to all slave processes;
        While   not-time-to-stop
             Begin
                  Wait for best moves from all slaves;
                  Select the best move subject to tabu restrictions;
                  Send the selected move to all slaves;
             End
        Force all slaves to stop;
        Return ($BestS$) /* of slave running on same machine */
End. /* MasterProcess */
```

Figure 1.4: Synchronous Parallel Tabu Search: The Master Process.

Asynchronous Parallel Tabu Search:

In this approach, each processor explores a subset of the neighborhood of its current solution. Each of these is competing with its neighbors (its adjacent processors) in finding a superior

```
**Algorithm** SlaveProcess;
**Begin**
        Initialize parameters and data structures;
        Wait for initial solution $S_0$ from master process;
        $BestS = S_0$;
        $CurS = S_0$; /* Current solution */
        **Repeat**
            Wait for selected move from the master;
            Perform the move;
            Update tabu_list;
            Update $BestS$ and $CurS$;
            Try all moves in partial neighborhood;
            Select best move and send it to the master;
        **Until** stop;
**End**. /* SlaveProcess */
```

Figure 1.5: Synchronous Parallel Tabu Search: The Slave Process.

solution. When the stopping criteria are met, every processor reports its best solution. The general outline of this parallelization approach is given in Figure 1.6. Similar asynchronous parallel tabu search implementations for the traveling salesman and quadratic assignment problems have been reported in [9].

```
**Algorithm** AsynchronousParallelTabuSearch;
**Begin**
1.      Construct initial solution and initialize parameters;
2.      Explores own neighborhood;
3.      Select best move subject to tabu restrictions;
4.      Update tabu list;
5.      Exchange current best solution with neighbors;
6.      Update current solution based on received neighbor solutions;
7.      **If** time-to-stop **Then Return** best solution;
8.      **Goto** step 2
**End**.
```

Figure 1.6: Pseudo Code for Asynchronous Parallel Tabu Search Algorithm.

Acceleration of Tabu Search through parallelization has been proved to be an effective strategy in numerous areas. Continuing with the literature on parallel TS, we cover its various applications and classify these according to Crainic's taxonomy.

Malek et. al [18] compared the performance of serial and parallel implementations of simulated annealing and Tabu Search for Traveling Salesman Problem (TSP). The reported experiments were performed on a 10 processor Sequent Balance 8000 computer. The authors reported that the parallel version of Tabu Search outperforms not only its sequential counterpart, but produced comparable or better results than the serial and parallel version of simulated annealing. Their strategy is synchronous following a 1-control, KS, SPDS approach.

In order to solve the flow shop sequencing problem, Taillard [26] employed a parallel implementation of the Tabu Search algorithm using a search space decomposition strategy. It is a 1-control, RS, SPSS algorithm. Battiti et. al [2] used Tabu Search to solve the Quadratic Assignment Problem (QAP) with hashing procedures. The scheme used is p-control, RS, MPSS. The authors report that the parallelization strategy is efficient and the average success time decreases with an increase in the number of processors.

Taillard [28] used parallel Tabu Search for vehicle routing problem. The parallelization is based on domain decomposition strategy, which is p-control, KS, MPSS. It was implemented on a Silicon Graphics 4D/35 workstation with 4 processors. Another effort by Taillard [27] to apply parallel Tabu Search to quadratic assignment problem follows a 1-control, RS, SPSS strategy. A ring of 10 transputers were used, but no implementation details were given. Chakrapani et al. [7] also used parallel Tabu Search to solve QAP, which is 1-control, RS, SPSS. The search is performed sequentially, while the move evaluation is done in parallel. The implementation is specifically designed for Connection Machines CM-2: a massively parallel SIMD machine. The authors report that the best known solutions were obtained in a lesser number of iterations. Furthermore, they were able to determine good suboptimal solutions to bigger problems in reasonable time.

Another effort to parallelize Tabu Search for TSP by Fiechter [10] used the p-control, KS, MPSS strategy. Intensification and diversification steps were implemented in the synchronous version. The algorithm was implemented on a network of transputers arranged in a ring structure. The authors report near-optimal solutions to large problems and almost linear speed-ups. TS was also applied for the vehicle routing problem by Garica et. al [11] also using search space decomposition strategy. It was a 1-control, RS, SPSS algorithm. The authors reported a noticeable improvement in solution quality over one of the best constructive algorithms for vehicle routing problem, with substantial reduction in runtime.

In order to improve parallel Tabu Search using evolutionary principles, the algorithm presented by Falco et. al [9] used multi-search thread strategy for the traveling salesman and quadratic assignment problems. It is a p-control, C, MPSS algorithm. The results reported indicate a marked improvement in solution quality as well as convergence speedup. Parallel TS for the 0-1 multidimensional knapsack problem was demonstrated by Nair [20], who used a multi-search threads strategy in a p-control, RS, MPDS algorithm. Taillard's extensive work with TS continued where he applied a p-control, RS, MPSS strategy to the job sequencing problem [25]. Several parallelization ideas which focussed on speeding up computations related to neighborhood evaluation didn't yield good results, either because the communication overtook computation, or the available computing platform (a ring of transputers, and a 2-processor Cray) were not suitable.

Crainic et. al [8], the authors who put forth the taxonomy of classification of Tabu Search, presented several of the strategies for both synchronous and asynchronous TS for multi-commodity location-allocation problems with balancing requirements. It was implemented on a heterogenous network of 16 SUN Sparc workstations. The results show that the average gap improved in most of the cases, when the number of processors increased. Mori and Hayashim [19], used parallel Tabu Search algorithm for voltage and reactive power control in power systems. Of the two schemes, one of them used the domain decomposition strategy, while the other scheme followed a multi-search threads strategy. The first one is 1-control, RS, SPSS and the second one is p-control, RS, MPDS algorithm.

A more recent work by Yamani et.al [1], parallelized Tabu Search for VLSI cell placement on heterogenous cluster of workstations, using PVM. The algorithm was parallelized on two levels simultaneously. The higher parallelization level can be classified as p-control while the lower level was 1-control. The synchronization strategy was RS and MPSS search differentiation strategy was used for both the levels. The authors reported obtaining proportional speed-up in most of the cases.

1.6 Simulated Evolution

Simulated Evolution (SimE) is a powerful general iterative heuristic for solving combinatorial optimization problems. It starts from an initial assignment, and then, following an evolution-based approach, it seeks to reach better assignments from one generation to the next. It is stochastic because the selection of which components of a solution to change is done according to a stochastic rule. Already well located components have a high probability to remain where they are. The probabilistic feature gives Simulated Evolution its hill-climbing property.

SimE assumes that there exists a population P of a set M of n (movable) elements. There is a cost function $Cost$ that is used to associate with each assignment of movable element m a cost C_m. The cost C_m is used to compute the goodness g_m of element m, for each $m \in M$. This goodness value is closely related to the overall target fitness value of the solution.

SimE algorithm proceeds as follows. Initially, a population[2] is created at random from all populations satisfying the environmental constraints of the problem. The algorithm has one main loop consisting of three basic steps, *Evaluation*, *Selection*, and *Allocation*. The three steps are executed in sequence until the population average *goodness* reaches a maximum value, or no noticeable improvement to the population *goodness* is observed after a number of iterations.

Parallelization of SimE hasn't attracted much attention from practitioners, with very few reported schemes mostly by the inventors of the algorithm themselves [16].

Domain Decomposition: Classified under the Type-II scheme, this approach in SimE involves the partitioning of a complete solution into smaller domains to be optimized in parallel. This implies concurrent execution of all its operators including *Allocation*. Hence the search behavior of this approach would differ from that of the serial algorithm - Such a parallelization strategy was reported by Kling and Banerjee [15] for VLSI cell placement, where alternating sets of rows are distributed among processors in every iteration. With

[2]In SimE terminology, a population refers to a single solution. Individuals of the population are components of the solution; they are the movable elements.

each processor limited to applying the SimE steps of Evaluation, Selection and Allocation on its assigned set of rows, this alternating distribution scheme would allow each cell to move to to any position in the placement within two iterations. A variation of this scheme was reported much later, which proposed random assignment of rows to processors [23].

```
ALGORITHM TypeII_Parallel_SimE_Master_Process
NOTATION
 (* $k_s$: Set of row indices for each process $s$. *)
 (* $\Phi$: The complete current solution. *)
INITIALIZATIONS;
 Read_User_Input_Parameters
 Read_Input_Files
 Construct_Initial_Placement
Begin
    Repeat
       ForEach $s \in m$ Generate_Row_Indices $k_s$ EndForEach;
    (* For each slave process. *)
          ParFor
             Slave_ Process($\Phi, k_s$)
    (* Broadcast cur. placement and row-indices. *)
          EndParFor
          ParFor
             Receive_Partial_Placement_Rows
          EndParFor
       Construct_Complete_Solution
    Until (Stopping Criteria is Satisfied)
 Return Best_Solution.
End. (*Master_Process*)
```

Figure 1.7: Outline of Master Process for Type II Parallel SimE Algorithm.

Figures 1.7 and 1.8 give the outlines of this parallelization strategy from both the Master and Slave perspectives. The slaves communicate back their assigned rows, modified by their respective allocation scheme to the Master after every iteration. The master reconstructs the complete solution, computes overall fitness and reassigns the rows for the next iteration.

This model is capable of achieving significant speedups for large problem sizes where such a row distribution scheme would provide a fair work distribution. However, the overall fitness values achieved by this method may vary from the serial values reached. In some cases, such as multi-objective VLSI design, a loss in solution quality has been reported along with reduced runtimes with increasing number of processors [22].

```
**ALGORITHM** TypeII_Parallel_SimE_Slave_Process($\Phi, k_s$)
*NOTATION*
 (* $B$ is the bias value. *)
 (* $\Phi^s$ are the rows assigned to slave $s$. *)
 (* $m_i$ is module $i$ in $\Phi^s$. *)
 (* $g_i$ is the goodness of $m_i$. *)
**Begin**
 Receive Placement_ And_ Indices
 *EVALUATION:*
     **ForEach** $m_i \in \Phi^s$ evaluate $g_i$ **EndForEach**;
 *SELECTION:*
     **ForEach** $m_i \in \Phi^s$ **DO**
        **Begin**
           **If** $Random > Min(g_i + B, 1)$
           **Then**
              **Begin**
                 $S^s = S^s \cup m_i$; Remove $m_i$ from $\Phi^s$
              **End**
        **End**
 *Sort the elements of* $S^s$
 *ALLOCATION:*
     **ForEach** $m_i \in S^s$ **Do**
        **Begin**
           Allocate($m_i, {\Phi_i}^s$)
     (* Allocate $m_i$ in local partial solution rows ${\Phi_i}^s$. *)
        **End**
 Send_Partial_Placement_Rows
**End.** (*Slave_Process*)
```

Figure 1.8: Outline of Slave Process for Type II Parallel SimE Algorithm.

Multithreaded Parallel Search: As part of the Type-III approach, this scheme implements parallel, independent search threads executed concurrently by each processor which may communicate periodically to exchange information and collectively navigate the search space. Such models have been reported with excellent results for other optimization algorithms such as Simulated Annealing, and Genetic Algorithms.

Figure 1.9 demonstrates an example of such a parallel model, which is very similar to the Asynchronous Multiple Markov Chains strategy discussed earlier for Simulated Annealing. After a fixed number of user-defined iterations, a slave returns back its fitness value to the master to compare against the global best received thus far. This central processor then either provides a better solution, or directs the slave to provide the complete solution if the latter has higher fitness.

Inspite of the success of Markov Chain models with Annealing and GAs, this scheme fails to perform with SimE [22]. The reason behind this is the nature of the heuristic and its intelligence. The primary concept behind this approach is to force each thread to explore a non-overlapping part of the search space near the global best solution reached so far. However, with the Selection operator heavily dependent on element goodness, and the deterministic Allocation process, achieving such random behavior on individual processors is highly unlikely.

1.7 Conclusion

The increasing computational power of generic PCs today represents a fantastic opportunity for cluster systems wherein high performance computing environments can be assembled from off-the-shelf hardware. With growing standardization of parallel communication and computation librariES, Out-of-the-box clustering solutions, and inexpensive, low latency networks, these computing platforms provide excellent avenues for accelerating performance and devising highly efficient algorithms.

In this chapter, we focussed on four popular heuristics, that have been extensively used for optimization in numerous areas - Simulated Annealing, Genetic Algorithms, Tabu Search and Simuated Evolution. Different parallel strategies were discussed under the context of 'Domain Partitioning' and 'Multithreaded Parallel Search' methods.

Acknowledgement

The authors express their gratitude to the King Fahd University of Petroleum & Minerals for support under the project code # COE/CELLPLACE/263.

```
ALGORITHM TypeIII_Parallel_SimE_Process
NOTATION
 (* $Count$ is the current retry value. *)
Begin
 INITIALIZATIONS:
     Read_User_Input_Parameters
     Read_Input_Files
     Construct_Initial_Placement
 Repeat
     EVALUATION:
       ForEach $m_i \in \Phi$ evaluate $g_i$;
     SELECTION:
       ForEach $m_i \in \Phi$ DO
         Begin
           If $Random > Min(g_i + B, 1)$
           Then
             Begin
                $S = S \cup m_i$; Remove $m_i$ from $\Phi$
             End
         End
     Sort the elements of $S$
     ALLOCATION:
       ForEach $m_i \in S$ Do
         Begin
         (* Allocate $m_i$ in $\Phi_i$. *)
           Allocate($m_i, \Phi_i$)
         End
     Calculate_Costs;
     If $Cur_{Cost} > Best_{Cost}$
     Then
       Begin
       Inform_Master;
       Count = 0
       End
     Else
       Count = Count + 1
     EndIf
     If $Count > Retry\_Threshold$
     Then
       Begin
         If $Cost_{master} < Cost_{cur}$
           Then Get_New_Placement
       End
 Until (Stopping Criteria is Satisfied)
End.
```

Figure 1.9: Structure of the Type III Parallel Simulated Evolution Algorithm.

References

[1] Ahmad Al-Yamani, Sadiq M. Sait, Habib Youssef, and Hassan Barada. Parallelizing tabu search on a cluster of heterogenous workstations. *Journal of Metaheuristics*, 8:277–304, May 2002.

[2] R. Battiti and G. Tecchiolli. Parallel biased search for combinatorial optimization: Genetic algorithms and tabu. *Microprocessors and Microsystems*, 16(351-367), 1992.

[3] Braun H. C. On solving traveling salesman problems with genetic algorithms. *Parallel Problem Solving by Nature*, pages 129–133, 1990.

[4] Pettey C. C., Leuze M., and Grefenstette J. J. A parallel genetic agorithm. *Proceedings of the Third International Conference on Genetic Algorithms*, pages 155–161, 1987.

[5] Erick Cantú-Paz. Designing efficient master-slave parallel genetic algorithms. *Genetic Programming 1998: Proceedings of the Third Annual Conference*, page 455, 1998.

[6] V. Cerny. Thermodynamical approach to the traveling salesman problem: An efficient simulated algorithm. *Journal of Optimization Theory Application*, 1(1):41–45, 1985.

[7] J. Chakrapani and Skorin-Kapov. Massively parallel tabu search for quadratic assignment problem. *Annals of Operation Research*, 41(327-341), 1993.

[8] T.G. Crainic, M. Toulouse, and M. Gendreau. Towards a taxonomy of parallel tabu search heuristics. *INFORMS Journal of Computing*, 9(1):61–72, 1997.

[9] I. De Falco, R. Del Balio, E. Tarantino, and R. Vaccaro. Improving search by incorporating evolution principles in parallel tabu search. In *Proc. of the first IEEE Conference on Evolutionary Computation- ICEC'94*, pages 823–828, June 1994.

[10] C.N. Fiechter. A parallel tabu search algorithm for large travelling salesman problems. *Discrete Applied Mathematics*, 51(243-267), 1994.

[11] Bruno-Laurent Garica, Jean-Yves Potvin, and Jean-Marc Rousseau. A parallel implementation of the tabu search heuristic for vehicle routing problems with time window constraints. *Computers & Operations Research*, 21(9):1025–1033, November 1994.

[12] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Publishing Company, Inc., 1989.

[13] Holland and J. H. *Adaptation in Natural and Artificial Systems.* Univ. of Michigan Press, Ann Harbor, Michigan, 1975.

[14] S. Kirkpatrick, Jr. C. Gelatt, and M. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):498–516, May 1983.

[15] R. M. Kling and P. Banerjee. ESP: Placement by Simulated Evolution. *IEEE Transaction on Computer-Aided Design*, 3(8):245–255, March 1989.

[16] R. M. Kling and Prithviraj Banerjee. Concurrent ESP: A placement algorithm for execution on distributed processors. *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 354–357, 1987.

[17] Munetomo M., Takai Y., and Sato Y. An efficient migration scheme for subpopulation-based asynchronously parallel genetic algorithms. *Proceeedings of the fifth International Conference on Genetic Algorithms*, page 649, 1993.

[18] M. Malek, M. Guruswamy, M. Pandya, and H. Owens. Serial and parallel simulated annealing and tabu search algorithm for the travelling salesman problem. *Annals of Operation Research*, 21:59–84, 1989.

[19] H. Mori and T. Hayashim. New parallel tabu search for voltage and reactive power control in power systems. *In Proc. of the 1998 IEEE International Symposium on Circuits and Systems - ISCAS'98*, pages 431–434, May 1998.

[20] S. Nair and A. Freville. A parallel tabu search algorithm for the 0-1 multidimentional knapsack problem. *11th International Parallel Processing Symposium*, April 1997.

[21] Grosso O.B. *Computer Simulations of Genetic Adaptation: Parallel subcomponent interaction in a multilocus model.* PhD thesis, The University of Michigan, 1985.

[22] Sadiq M. Sait, Mustafa I. Ali, and Ali M. Zaidi. Evaluating parallel simulated evolution

strategies for vlsi cell placement. In *The 9th International Workshop on Nature Inspired Distributed Computing (NIDISC'06), Rhodes Island, Greece*, April 2006.

[23] Sadiq M. Sait, Mustafa I. Ali, and Ali Mustafa Zaidi. Multiobjective vlsi cell placement using distributed simulated evolution. *International Symposium on Circuits and Systems, ISCAS*, May 2005.

[24] Sadiq M. Sait and Habib Youssef. *Iterative Computer Algorithms with Applications in Engineering: Solving Combinatorial Optimization Problems.* IEEE Computer Society Press, California, December 1999.

[25] Taillard. Parallel tabu search techniques for the job sequencing problem. *ORSA: Journal on Computing*, 6(2)(108-117), 1994.

[26] E. Taillard. Some efficient heuristic methods for the flow shop sequencing problem. *European Journal of Operational Research*, 417:65–74, 1990.

[27] E. Taillard. Robust tabu search for the quadratic assignment problem. *Parallel Computing*, 17:443–455, 1991.

[28] E. Taillard. Parallel iterative search methods for the vehicle routing problem. *Networks*, 23:661–673, 1993.