Parallel complex systems simulation

Schoneveld, A.

Citation for published version (APA):

General rights
It is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), other than for strictly personal, individual use, unless the work is under an open content license (like Creative Commons).

Disclaimer/Complaints regulations
If you believe that digital publication of certain material infringes any of your rights or (privacy) interests, please let the Library know, stating your reasons. In case of a legitimate complaint, the Library will make the material inaccessible and/or remove it from the website. Please Ask the Library: https://uba.uva.nl/en/contact, or a letter to: Library of the University of Amsterdam, Secretariat, Singel 425, 1012 WP Amsterdam, The Netherlands. You will be contacted as soon as possible.
Chapter 4

Task Allocation by Parallel Evolutionary Computing

"You cannot teach a [man] anything, you can only help him to find it within himself."

-Galileo

"You do not destroy an idea by killing [people], you replace it with a better one."

-Unknown

4.1 Introduction

In the previous chapter the fundamental properties of the Task Allocation Problem (TAP) on random task graphs instances and thermo-dynamical aspects of the optimization process using Simulated Annealing have been discussed. In this chapter we will discuss in more detail the optimization of TAP instances using parallel evolutionary algorithms: Simulated Annealing, Genetic Algorithms and Steepest Descent. A so called Abstract Cellular Genetic Algorithm is implemented using the P-CAM framework to study performance and problem specific aspects of TAP optimization.

Heuristic search methods have been formulated as a compromise between computational search cost and the final cost of (sub) optimal solutions for problems which are hard to optimize. In order to reduce the execution time associated with a heuristic search, parallel versions of the corresponding search strategies must be developed. Two typical natural solvers are Simulated Annealing (SA)[83] and Genetic Algorithms (GA)[72]. A fundamental problem is that both methods are difficult to parallelize to a level of high scalability. Classical GAs use global knowledge for their selection process. An essential problem in SA is that the method is inherently sequential.

In this work we will apply a GA, SA and a steepest descent search (SD), to find (sub) optimal allocations of the TAP as introduced in Chapter 3. Both a realis-
tic task graph as well as random task graph instances will be studied. A finite element mesh of a car is used as a representative of a realistic task graph. In Section 4.3, a unifying evolutionary optimization framework is introduced in which we can embed parallel versions of GA, SA and SD. For the parallel GA, we discuss the consequences of the imposed parallelization strategy for its selection scheme, and a time complexity formula of the algorithm, expressing its scalability, is derived.

Problem specific enhancements for each of the three algorithms are presented in Section 4.5. Experimental results, relating to problem specific enhancements and individual performance of the optimization methods, are given in Section 4.6. Furthermore, we investigate the trade-off between solution quality and optimization time (wall clock). Finally, Section 4.7 is used to reflect on the various aspects and consequences of this work.

4.2 Genetic Algorithms

Genetic Algorithms[60, 72] (GA) are “blind” search algorithms and refer to a family of computational models inspired by evolution. They are based on the mechanics of natural selection and natural genetics. Potential solutions to specific problems are encoded as simple chromosome-like data structures. The evolution of the algorithm proceeds in combining chromosomes or individuals of a previous generation in order to form a next generation. In a sense a GA can be regarded as a guided random walk, using historical information to effectively exploit new search points. A GA can be applied to a variety of combinatorial, i.e., NP-hard and function optimization problems.

An implementation of a GA begins with a population of random individuals, i.e., possible solutions. Next, the individuals are evaluated according to a so called objective function (or cost function), which in turn is used to calculate the fitness of the individual in the current population (the higher the fitness the “better” the solution). The individuals are given reproductive opportunities related to their fitness. These reproductive opportunities are actually probabilities assigned to individuals to indicate the chance to get selected in the next generation. Individuals with a high fitness are given high probabilities. The fitness of a solution is often defined with respect to the current population.

Roughly there are only two components of a GA that are problem dependent: the encoding of a solution and the fitness-function. A solution to a problem can be described by a set of parameters which are given some value. If we take for example TAP, the parameters consist of the allocation of tasks to a given processor. Most often the variables representing the parameters are represented by bit strings in a GA. As such the variables are discretized in an a priori fashion and the range of discretization corresponds to some power of 2. If we use for example 10 bits per parameter, we obtain a range of 1024 discrete values. Sometimes it can be quite difficult to properly encode some parameter. What if a parameter can only be assigned an exact number of \( X \) values and this \( X \) is not a power of 2. For example, our TAP problem, which consist of a \( n \) parameters rep-
representing the number of tasks, which can take $P$ different values per parameter. Of course it is possible to code this also in bit strings (which is naturally always the final coding scheme in a digital computer), but a different representation can often be more useful, in this case $P$-ary bit strings. An important criterion for the fitness function is that it must be relatively fast to calculate. This is especially important for genetic algorithms, since they work with a large population of potential solutions.

It is useful to view the execution of a genetic algorithm as a two stage process. We start with the current population. Selection is applied to form an intermediate population. Subsequently reproduction is used to form the next generation. This process is schematically depicted in fig. 4.1.

Let's call $f_i$ the value of the objective function of individual $i$ and $f$ the average value of the fitness function over the total population. We can now define the fitness of an individual as $\frac{f_i}{f}$. We can regard this fitness as the probability (compare to $p_i$ in Markov Chain simulation) of an individual to be selected in the intermediate generation. After the selection the recombination step is applied, which in turn can be divided in a crossover and a mutation step. During the recombination selected individuals are pairwise crossed with a certain large probability which results in two offspring, which both have features of their parents. Also there is a chance that some bits in the bit string are flipped with a low probability, this is called mutation. The resulting individuals are placed in the new generation. This is repeated until the new population is full, for example if it is as large as the previous one. Then the cycle starts all over until some stop-criterion is fulfilled.

Summarizing, the main cycle of the GA consists of three operators:

- Selection
- Crossover
- Mutation

These are the main operators in a genetic algorithm, they are found almost in any GA. These operators jointly determine the efficiency of the algorithm. It is possible to add other operators in this cycle, but they often only have marginal differences in the original operators. An example can be a restriction in the selected offspring for the next generation, where only the best of the two individuals is selected. Next we will discuss the complete GA cycle in more detail in order to get a deeper understanding of the functioning of a GA, such that we can outline more clearly the issues in parallelizing this natural solver.

4.2.1 Selection

Before entering the next generation there is a stage in which the GA keeps a temporary or intermediate population of individuals that may survive the next generation. This temporary population is called the mating pool. The mating pool consists of copies of individuals from the previous generation. A selection process is responsible for selecting those individuals. Individuals with a higher fitness have a larger probability to get in the mating pool. Different selection strategies are possible: roulette wheel, stochastic remainder and tournament selection [60]. Other methods are possible, for example combinations with SA[98], but the first three strategies represent the bulk of the selection methods used within the GA-community.

Given an encoded solution of a function, how can one decide if it is near the optimum, or far from the optimum, if you have no knowledge about this optimum at all? The solution to this problem in GA, is to compare individuals within the current population. Instead of absolute fitness, GA uses relative fitness, better individuals receive a higher fitness. In order to obtain the fitness, a fitness function must be designed, which quantifies the essential features of a solution to a problem.

4.2.2 Recombination

Recombination is the most essential feature of a GA. Without recombination operators there would be no exploration of the phase space. Recombination in GA is essentially different from the random walks in SA. In GA the information in the individuals is used to guide the search for new exploration points. By combining information from individuals the offspring are "hoped" to receive the best of both worlds. This combining of individuals is called crossover. An other essential operator is mutation, which has to ability to reintroduce previously extinct individuals by a noise generation mechanism.
4.2.3 Convergence

An optimization algorithm terminates when the optimum is found. This argument is of course only realistic in the case of known global optima. But it is obvious that we do not know these solutions when dealing with a "real" application. Known local optima can however by used when it is the GA itself we are interested in. A stopping criterion has to be found to prevent the GA from running indefinitely. A possible scheme is to wait for a number of $X$ consecutive generations where the highest fitness in the population does not change. Another version is to wait for the moment when the difference between the current best optimum and the previous best optimum gets smaller than some small value $\varepsilon$. There is no way to deterministically establish how many generations it will take for a GA to terminate, because of the stochastic nature of the algorithm itself. Maybe it is possible to find statistical proofs for convergence, like the ones known for Monte Carlo methods. Because of the lack of a proper theory for GA, research in this direction can become quite difficult.

4.3 Parallel Evolutionary Algorithms

Despite a GA's capability to implicitly process $N^3$ so called schemata at the same time [60], which makes it implicitly parallel, we would like to exploit some explicit parallelism in order to efficiently execute a GA on parallel architectures. At a first glance it seems difficult to exploit this explicit parallelism, because of the global knowledge that is needed for the reproduction step. In order to select individuals for the mating pool one has to calculate the average fitness of the entire population. In other words, it is difficult to localize a GA. A solution could be a selection scheme that does not depend on global knowledge, one could use a local selection scheme. An example of a local selection scheme, i.e., one that does not depend on global knowledge, is tournament selection. Other techniques include the introduction of a neighborhood structure, where individuals can only mate with other individuals in their direct neighborhood. Several methods using the latter method have been suggested in the literature:

- Island models
- Stepping stone models
- Isolation by distance models

The situation is even worse for the Simulated Annealing algorithm. The Metropolis algorithm, which composes SA, is inherently sequential. A new step can only be started if the previous has finished since the new step does not know the state of the system until the previous step is completed.

But still there are a number of possibilities for parallelism for both GA as well as SA. Several techniques exist for parallelizing these algorithms [1, 26, 19, 10, 62, 130, 27, 63, 8, 45, 15, 61]. In this work a generic parallelization approach
is taken by employing a parallel evolutionary algorithm in which several evolutionary optimization strategies can be embedded. In the remainder of this section we will discuss this approach which is mainly based on the Isolation by distance model of parallel GA.

### 4.3.1 An Abstract Cellular Genetic Algorithm

In [2] a generic algorithm, the so called Abstract Genetic Algorithm (AGA), for both SA and GA was introduced. However, the AGA was not designed to facilitate parallelization. We describe an Abstract Cellular Genetic Algorithm suitable for parallelization. In this ACGA, locality is introduced by mapping both GA and SA onto Cellular Automata. Examples in which a GA is mapped onto Cellular Automata can be found in [99, 62, 154]. In general, it is not possible to map SA on Cellular Automata. However, locality can be imposed to SA by applying a population based algorithm [60]. Another approach is to use simultaneous independent searches, which is equivalent to the previous method apart from the interactions between the Markov chains [8]. Other approaches to parallelize SA are discussed in [155].

To avoid the use of global information, which is necessary in the Abstract Genetic Algorithm of Aarts et al. [2], we introduce a local spatial neighborhood structure. The main idea behind the ACGA is to make an analogy between the chromosome (or solution vector) and a cell in Cellular Automata. Each chromosome is assigned to a cell, which explicitly defines its neighborhood structure. All communication is local and cells can only interact with direct neighbors. In Fig. 4.2 the ACGA is formulated in pseudo code.

### 4.3.2 ACGA instances

From the ACGA framework, a Cellular GA (CGA) with local selection can be derived straightforwardly. We only have to select the various genetic operators. First the selection operator. A conventional GA uses a global method to select the parents, for example, roulette wheel selection. Within a CGA the parents are selected from a neighborhood of size \((2r+1)^2\), where \(r\) is the radius (see Fig. 4.3), using a fitness function \(F\). A cell is chosen as a parent if a uniformly distributed random number \(\xi \in [0,1)\) satisfies the following rule:

\[
\xi \leq \frac{F(x_m)}{\sum_{x_j \in A_{k(r)}} F(x_j)}.
\]  

(4.1)

where \(A_{k(r)}\) is the neighborhood with radius \(r\) of cell \(x_k\), including \(x_k\), and \(x_m \in A_{k(r)}\). We call this selection mechanism Local Roulette Wheel (LRW).

Another option is tournament selection, which we call Local Tournament Selection (LTS) in the case of CGA, since selection is restricted to a local region. There is an advantage in using LTS over LRW in small neighbourhoods, because LRW suffers from sampling errors when used on small populations. For
Initialize

Do

For each cell in the population

Do

Choose a parent list (choice)
Recombine parent list (production)
Mutate the offspring
Evaluate offspring

If offspring meets some criterion (selection)

Then
accept offspring

Else
leave the current chromosome in its place

Endif

Endfor

Until maximum number of generations (iterations)

Figure 4.2: Algorithmic specification of the Abstract Cellular Genetic Algorithm

Figure 4.3: An ACGA population of 20x20 chromosomes. The black cell indicates the center cell of a neighborhood (grey cells) with radius r = 2.

this reason we have decided to apply LTS selection in our optimization experiments. In LTS two individuals are randomly chosen from the neighborhood. The individual with the lowest fitness is discarded. As a recombination operator the usual GA-crossover operator can be chosen.
Crossover between two cells is realized by taking a uniformly distributed random number as the splitting location. Mutation is implemented by “bit-flipping” every bit of a chromosome with a fixed probability, where a bit is a P-ary number \( P \geq 2 \). The fitness of the new chromosome is calculated at the evaluation step. Since one child out of two is accepted, a child selection criterion must be applied, e.g., discard the child with the highest cost. Note that a maximum radius for the CGA is identical to a conventional GA with child selection.

Another instance of the ACGA is a special variant of Simulated Annealing (see Chapter 3): Cellular Simulated Annealing (CSA). To introduce locality in the SA algorithm we use an alternative approach where several configurations exist together on a 2-dimensional grid. These configurations only know of the existence of other configurations in their direct neighborhood. If a new configuration has to be evaluated for acceptance, not only the previous configuration is taken into account, but also its neighborhood set. Rejection of a new configuration can cause any of the configurations in a neighborhood set to take over the current spatial grid location. In previous work we studied a CSA with interactions, applied to the Traveling Salesperson Problem [144]. In this work we will use CSA without interactions, also known as simultaneous independent searches [8]. At the end of the annealing process, the allocation with the lowest energy value is selected.

```
Allocate := Random Allocation
Lowest := CalculateCost(Allocation)

Do

  Previous Allocation := Allocation
  For each Task j do for all Processors i
    Mutated Allocation := Allocation with Task j
      on Processor i
    Cost := CalculateCost(Mutated Allocation)
    If (Cost < Lowest)
      Stored Allocation := Mutated Allocation
      Lowest := Cost
    Endif
  Endfor
  Allocation := Stored Allocation

Until Allocation <> Previous Allocation
```

**Figure 4.4:** Pseudo code for the steepest descent algorithm

The steepest descent (SD) algorithm (see Fig. 4.4) is comparable to gradient based optimization methods in continuous optimization. It evaluates the cost
of each mapping that differs a single mutation with the present solution vector.
The mutation that gives the highest decrease in cost (steepest descent) is ac­
cepted. This procedure is repeated and continued, until no one-step mutations
are left that decrease the cost. Such a mechanism is also known as a greedy
method. Although the method is fast in comparison with SA and GA (for small
chromosomes), and straightforward to implement, it gets stuck in the firstly encoun­
tered local minimum, and consequently is the least attractive method of the
three, for global optimization of complex combinatorial optimization prob­
lems. SD can be embedded into the cellular framework, leading to a parallel
implementation of the algorithm with multiple non-interacting SDs.

4.3.3 An ACGA implementation: Parallel MAP

In our research group a task allocation optimization tool, MAP, has been de­
veloped based on a sequential genetic algorithm solver [36]. MAP was intended
to be used in conjunction with a domain decomposition tool in order to create a
coarse grained initial partitioning of finite element grids, based on the concept of Redundant Decomposition [38, 35]. The modular design of the MAP tool, i.e., a graphical user interface separated from the optimization kernel, enabled us to
enhance the kernel using a parallel optimization algorithm based on the ACGA.
In addition, the topological structure of the ACGA method is well suited for im­
plementation in the P-CAM framework. The computational task graph is sim­
ply a two dimensional grid. The mapping between a P-CAM virtual particle
and the basic simulation entity, a solution vector or chromosome, of the ACGA
method can also be one to one.

4.3.4 Locality in CGA

We already mentioned that LTS is used as the selection mechanism for the CGA
experiments. Here it is shown that the neighborhood size has no significant
influence on this selection process, implying that significant differences in the
convergence behavior can only be caused by recombination operators acting on
localized parts of the population.

Consider the expected number of current best chromosomes in the next popula­
tion, using the LTS mechanism. Let \( P_r(\text{best}) \) be the probability of choosing the
best chromosome in one selection round (choose two chromosomes) given a cer­
tain radius \( r \). Assuming that there is only one best chromosome present in the
current population, the following formula can be derived:

\[
P_r(\text{best}) = \frac{2(2r+1)^2 - 1}{(2r+1)^4}.
\]  \hspace{1cm} (4.2)

Furthermore the expected number of best chromosomes in the next generation,
is given by:

\[
E_r[\text{best}] = (2r+1)^2 \times P_r(\text{best}) = \frac{2(2r+1)^2 - 1}{(2r+1)^2} = 2 - \frac{1}{(2r+1)^2}.
\]  \hspace{1cm} (4.3)
Note that $\lim_{r \to \infty} E_r[best] = 2$. Furthermore, if $r = \frac{N-1}{2}$ we have global tournament selection, with a population of $N \times N$ chromosomes. The convergence of the above limit is fast, indicating that only for small $r$ a deviation of the normal GA tournament selection should be expected. Similar arguments can be given for the reproduction of other chromosomes in the population. Therefore, it can be expected that the composition of next generations is not influenced significantly by the radius size. Experimental validation of this result is given in the next section.

4.3.5 Allocating a Finite Element grid

Figure 4.5: A finite element mesh of a car. Courtesy provided by ESI-SA (Paris) in the framework of the ESPRIT-III project CAMAS (nr. 6756).

Initial work has been done with the CGA on the "de Jong's test functions"[144]. The results were very promising from the perspective of solution quality and convergence speed. Here we apply the CGA to a real world problem occurring in many industrial applications: load balancing of a decomposed mesh for parallel finite element simulations. As a test case we have used a car mesh (see Fig. 4.5). First domain decomposition was applied, using recursive spectral bisection (RSB, see e.g., [141]), to partition the mesh in 32 clusters. As a target architecture we employed a 16 processor grid topology, which is a model of a 16 processor partition of a 32-node Parsytec PowerXplorer. With the resulting parallel execution- and processor topology graph, a series of experiments has been performed. Although only one specific instance of a load balancing experiment is shown, the results are general with respect to load balancing for finite element applications. The experiments only discuss the relative solution quality for different mutation rates and neighborhood sizes. Actual solution quality is discussed elsewhere [36].

In the following experiments the crossover probability was set to 1, and the population size to 400 (20x20). Subsequently the sensitivity to parameter changes has been measured for the (i) population diversity, (ii) the convergence of the
fittest chromosome and (iii) the number of convergence steps. The algorithm is assumed to be converged when the best chromosome has not changed for 300 iterations.

*Population diversity* is defined as the mean Hamming distance, $<h>$, over the population. The Hamming distance between two chromosomes is defined as the number of different genes. Figs. 4.6 and 4.7 show the evolution of the diversity for varying radii with fixed mutation rate $\mu$.

![Figure 4.6: Diversity evolution for different $r$ (1, 2, 3, 7, 9) and $\mu = 0.02$.](image)

Next, Table 1 shows the influence of $r$ and $\mu$ on the solution quality. Table 2 depicts the dependence of the convergence length on $r$ and $\mu$. All experiments have been repeated 25 times.

<table>
<thead>
<tr>
<th>$\mu$, $r$</th>
<th>1</th>
<th>5</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.015</td>
<td>606 ± 12</td>
<td>611 ± 12</td>
<td>611 ± 10</td>
</tr>
<tr>
<td>0.020</td>
<td>618 ± 18</td>
<td>612 ± 13</td>
<td>608 ± 9</td>
</tr>
<tr>
<td>0.025</td>
<td>618 ± 17</td>
<td>610 ± 6</td>
<td>612 ± 13</td>
</tr>
<tr>
<td>0.030</td>
<td>646 ± 19</td>
<td>619 ± 13</td>
<td>612 ± 14</td>
</tr>
<tr>
<td>0.040</td>
<td>683 ± 24</td>
<td>786 ± 86</td>
<td>770 ± 84</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\mu$, $r$</th>
<th>1</th>
<th>5</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.015</td>
<td>804 ± 222</td>
<td>657 ± 155</td>
<td>650 ± 165</td>
</tr>
<tr>
<td>0.020</td>
<td>473 ± 154</td>
<td>692 ± 181</td>
<td>681 ± 165</td>
</tr>
<tr>
<td>0.025</td>
<td>1014 ± 374</td>
<td>756 ± 214</td>
<td>848 ± 172</td>
</tr>
<tr>
<td>0.030</td>
<td>684 ± 194</td>
<td>772 ± 262</td>
<td>752 ± 256</td>
</tr>
<tr>
<td>0.040</td>
<td>624 ± 261</td>
<td>619 ± 250</td>
<td>651 ± 221</td>
</tr>
</tbody>
</table>

**Table 4.1:** Cost of final solution for varying $\mu$ and $r$.

**Table 4.2:** Convergence length for varying $\mu$ and $r$.

Fig. 4.6 shows that the evolution of the population diversity is not significantly dependent of $r$. Except for $r = 1$, the behavior differs from the global selection mechanism. This is in correspondence with our theoretical predictions from Section 4.3.4. This difference is even more manifest for larger $\mu$ (see Fig. 4.7).
According to the formal derivation in Section 4.3.4 the behavior of the selection process cannot be notably influenced by radius size. Therefore, observable differences must be due to local recombination operations. This is the motivation for studying the dependence on $r$ and $\mu$ of the solution quality (Table 1) as well as the convergence length (Table 2).

The diversity is inherently higher for $r = 1$ (see Fig. 4.6 and Fig. 4.7), therefore it is expected that, for $r = 1$, a mutation probability must be chosen which is lower than the optimal values for $r > 1$ in order to reach comparable convergence behavior.

The final solution quality in Table 1 appears to have a lower best $\mu$ in the case of $r = 1$ than for larger, near global $r$, which supports our previous assumption. The convergence lengths shown in Table 2 tend to be highly sensitive to variations in $\mu$ for $r = 1$.

A preliminary conclusion is that the behavior of the CGA is identical to GA. Except for $r$ near 1 a small difference exists. It should be noted that this has only been validated for GAs based on tournament selection. Furthermore the results show that both solution quality as well as convergence length of CGA (for $r > 1$) are comparable to GA.

### 4.3.6 Time complexity

Next, the time complexity for executing one iteration of a parallel 2D CGA, implemented on a distributed memory parallel architecture, is considered. The CGA is implemented straightforward using the P-CAM framework.

The total parallel time $T_r(N, p)$ ($p$ processors, radius $r$ and population size $N$) is

---

**Figure 4.7:** Diversity evolution for different $r$ (1, 5, 7, 9) and $\mu = 0.03$. 

---

...
modeled by:

\[ T'(N, p) = T'_{\text{calc}}(N, p)\tau_{\text{calc}} + T'_{\text{comm}}(N, p)\tau_{\text{comm}}, \]  

(4.4)

where \( T'_{\text{calc}}(N, p) \) corresponds to the basic steps performed in the calculation process, \( T'_{\text{comm}}(N, p) \) corresponds to the number of basic units communicated every iteration step. Furthermore, \( \tau_{\text{calc}} \) and \( \tau_{\text{comm}} \) can be respectively associated with the flop rate and the communication bandwidth. \( T'_{\text{calc}}(N, p) \) can be written as:

\[ T'_{\text{calc}}(N, p) = (T'_{\text{select}}(N, p) + T'_{\text{cross}}(N, p) + T'_{\text{eval}}(N, p)) \frac{N}{p} + \frac{N}{p} T'_{\text{copy}}(N, p). \]  

(4.5)

Both \( T'_{\text{eval}} \) and \( T'_{\text{cross}} \) are \( O(1) \) in \( p \) and \( r \), \( T'_{\text{select}} \) is \( O(1) \) for LTS selection and \( T'_{\text{copy}} \) corresponds to copying the children to the current population. The communication term, \( T'_{\text{comm}}(N, p) \), can be written as:

\[ T'_{\text{comm}}(N, p) = 4r(\sqrt{\frac{N}{p}} + 2r). \]  

(4.6)

Combining Eq. 4.5 and Eq. 4.6 results in:

\[ T'(N, p) = \left( \frac{2N}{p} \right)\tau_{\text{calc}} + (4r \sqrt{\frac{N}{p}} + 8r^2)\tau_{\text{comm}}. \]  

(4.7)

In the equations above \( \sqrt{\frac{N}{p}} \) is integer, otherwise appropriate floor and ceiling functions must be introduced. We will use Eq. 4.7 to verify parallel timings. Note that, due to the absence of any global communication, the parallel algorithm is theoretically scalable.

In Fig. 4.8 timing results as well as theoretical predictions are displayed for running 100 iterations of the CGA on 1, 4, 9 and 16 processors. For these experiments we have used fixed mutation (0.02) and crossover (1.0) probabilities and population sizes of 576 (24 x 24) and 2304 (48 x 48) chromosomes.

Fig. 4.8 shows that the measured values behave conform the theoretical predictions of Eq. 4.7. Similar experiments were performed for larger radii, resulting in comparable behavior. Again correspondence with the theoretical predictions was obtained. From the experiments it can be concluded that the CGA is scalable, according to the theoretical prediction given by Eq. 4.7.

Since CGA allows for (theoretically) highly scalable implementations, and shows good convergence behavior, it appears to be a very fruitful approach for large optimization problems.

### 4.4 The Task Allocation Problem

After discussing the evolutionary framework and a sample application, let us turn to the theoretical model of random task graphs. In this section we will
study the application of different instances of the ACGA to the allocation of random task graphs on fully connected machines, based on the models and the cost function (Eq. 3.3) as discussed in Chapter 3.

The cost function in Eq. 3.3 has the locality property, which means that single task changes in an allocation can be propagated into the cost without having to recalculate the whole cost function. Only a difference has to be calculated instead [37]. This is specifically useful if an optimization algorithm is applied that is based on incremental changes, and as such can exploit the direct calculation of these mutations. A disadvantage of using Eq. 3.3 is the fact that it is not a correct model for the absolute cost. However, it has been shown that its optima approximately coincide with those of more realistic equations [36].

An allocation of tasks to processors is coded as a sequence, where each letter in the sequence is a number from the alphabet \{1, 2, ..., P\}. The index of this sequence corresponds to the vertex number of the task in the task graph, while the letter in the sequence corresponds to the processor allocation number of the given task. Each of the three optimization methods manipulates this solution encoding in its own characteristic manner in the optimization process.

### 4.4.1 Sequential and Parallel Optimal Allocation

If we consider the spectrum of possible task allocations, we already observed a transition, within the optimal allocations, from sequential to parallel, when \(\beta\) is increased from 0 to 1 (or equivalently, if \(\gamma\) is decreased from 1 to 0) (see Chapter 3). For example the case \(\beta = 0\) corresponds to the situation where allocation is optimal if all tasks reside on one processor. This can be the result of the fol-
following hypothetical situations: either the processor speed is infinite or the network bandwidth is zero or the task weights are zero or the inter-task communication is infinite. On the other hand if $\beta = 1$, the optimal allocation corresponds to an equal distribution of all tasks over the available processors. In this case the above situation is simply reversed, that is, infinite network bandwidth, etc.

### 4.5 Problem specific adjustments

The power of evolutionary methods is mainly manifest in their general applicability. Unlike specialized optimization methods, they do not rely on domain specific knowledge. However, neglecting a priori information regarding the optimization problem may seriously hamper the performance of such methods. In this section, specific properties of the TAP are exploited to enhance the application of the various evolutionary strategies.

#### 4.5.1 CGA: cluster crossover

![Diagram](image)

**Figure 4.9:** Schematic representation of the structured crossover vs. the uniform crossover. Two task allocations, C1 and C2, are selected as children for a recombination operation. The left hand side of the figure depicts the cluster crossover: a sub cluster is chosen in C1, which is recombined with the complementary cluster in C2. The right hand side of the figure depicts the uniform crossover: an arbitrary crossover point (dotted line) is chosen and the two sections of C1 and C2 are combined. The grey-values in the chromosomes represent the processor to which a task is allocated.

An important aspect of Genetic Algorithms is the exploitation of so called building blocks or schemata [60]. Building blocks represent partially known gene values, which are most significant for the cost associated with a complete allocation. Recombination operators are applied to mix optimal building blocks from chromosomes in the current population. An important property, for efficient sampling of optimal building blocks, is their compactness. This means that the defining length in the chromosome must be short enough, such that destruction of building blocks by uniform crossover is minimized [60]. For graph
problems, the defining length of the schemata can often be very long, because the inherent locality of the graph can not be represented by a one dimensional encoding. However, the disruption process can be avoided if the internal structure of the problem domain is used as additional knowledge. Two approaches can be taken in order to reach this goal. Either the coding of solutions must somehow represent the inherent structure of the problem domain or the genetic operators must be able to exploit this. In this work we take the latter approach for reasons of generality, i.e., the same coding must be used for different optimization methods.

The internal structure in a random task graph is obvious, namely the connectivity between the vertices. In order to use this information, we propose a recombination method, which exploits connected sub graphs instead of sub chromosomes. The procedure is as follows: First choose a random vertex \( v \) and a random size \( S \in [0, n] \), where \( n \) corresponds to the total number of vertices. Subsequently, a cluster of \( S \) connected vertices is chosen. The cluster is constructed by taking vertex \( v \) as the center and first choosing all 1-connected neighbors, followed by all 2-connected neighbors. This process continues until a cluster of size \( S \) is reached. We toss the name cluster crossover for this structured recombination operator. The process is illustrated in Fig. 4.9, where both cluster crossover and 1-point uniform crossover are depicted.

### 4.5.2 CSA and CSD: single task mutation

The computational overhead that is introduced by calculating the difference in cost between successive configurations, that differ only by a single task mutation, involves calculation of the cost corresponding to the previous \( (H) \) and the next \( (H') \) allocation. The complexity of this calculation grows with the size of the task graph. Fortunately, Eq. 3.3 allows us to formulate the change in cost \( \delta H = H - H' \) as follows:

\[
\delta H = 2w_k(W_m - W_n - w_k)\beta + 2R(1 - \beta)
\]  \hspace{1cm} (4.8)

if the allocation number of task \( k \) changes from \( m \) to \( n \) \((m \neq n)\), else \( \delta H \) is 0; \( w_k \) is the work associated with task \( k \), \( m \) is the previous allocation number, \( n \) the new one, \( W_m \) is the execution time due to the work on processor \( m \) and equivalently, \( W_n \) for processor \( n \). Both \( W_m \) and \( W_n \) are computed before the mutation. The term \( R \) denotes the change in the communication cost (communication cost before - communication cost after). CSA and CSD both rely on a single task mutation mechanism and therefore can benefit from the direct computation of \( \delta H \) (Eq. 4.8).

### 4.6 Experimental results and discussion

The purpose of this section is two-fold. Firstly, the effect on the CGA’s convergence behavior, caused by the cluster crossover operator, is studied. A broad
range of TAP instances is optimized using both the uniform and cluster crossover. The convergence of the best individual (allocation with the lowest cost) in the population is used for comparison.

The second objective of this section is to examine the performance (in terms of solution quality and turn around time) of the three optimization strategies applied to the TAP.

In all the experiments below we assume to have a fully connected, homogeneous parallel machine consisting of \( P \) processors.

### 4.6.1 Uniform vs. Cluster crossover

In this section experimental results are presented with respect to the convergence rate of both cluster and uniform crossover. Both \( \beta \) and \( \gamma \) (see Section 4.4) values are taken from the interval \([0,1]\). The plots contain the evolution of the cost of the best chromosome in the population, averaged over 25 CGA runs, for both the cluster- and the uniform crossover. All experiments have been carried out with a parallel CGA with \( r = 2 \), a crossover probability of 0.7, and a mutation probability of \( 1/n \), where \( n \) is the number of tasks. The initial population is generated by randomly assigning tasks to processors for each chromosome. A population is considered to be converged if the cost of the best individual has not changed over 100 generations. All optimization runs were performed for the same instance of a specific random task graph (parameterized by \( n \) and \( \gamma \)). The initial populations of the different runs were generated randomly, which accounts for small cost differences in the initial generation. The CGAs have been executed on a 16-node partition of a Parsytec CC-40 machine (40 PowerPC 604s, each having 96 Mbyte of RAM).

First, we consider an experiment in which the calculation term has been disabled, by setting \( \beta = 0 \) (see Fig. 4.10). Furthermore, \( n = 64, P = 8 \) and \( \gamma = 0.2 \). The optimum of this specific case corresponds to a sequential allocation with \( H = 0 \). For two higher values of \( \beta \), keeping the other parameters fixed, comparable experimental data are displayed in Figs. 4.11 and 4.12. In Fig. 4.11 a horizontal line is plotted, indicating the lowest average cost value in the uniform crossover convergence plot.

In Fig. 4.13, the convergence behavior on a highly connected task graph (\( \gamma = 0.8 \)) is shown, using the same \( \beta \) as in Fig. 4.12. Comparable experiments have been carried out using larger task graphs and more processors, leading to similar results.

The experimental results presented in this section indicate that the cluster crossover operator performs structurally better than uniform crossover over a wide range of task graph parameters. The results are indicative for a larger series of experiments which have been performed (data not shown). Some parameter settings can not be used to discriminate between cluster- and uniform crossover, for example if there are no connections at all (\( \gamma = 0.0 \)) or if the communication term is neglected (\( \beta = 1.0 \)) (data not shown). In Chapter 3 we have shown that it is possible to identify, for a given fixed \( \gamma \) or \( \beta \), a corresponding value of either \( \gamma \) or \( \beta \), where the calculation and communication terms of the
Figure 4.10: Cost evolution of the best individual in the population for cluster- and uniform crossover, using the following parameters: $n = 64, P = 8, \gamma = 0.2, \beta = 0.0$.

Figure 4.11: Cost evolution of the best individual in the population for cluster- and uniform crossover, using the following parameters: $n = 64, P = 8, \gamma = 0.2, \beta = 0.15$. The horizontal dotted line indicates the lowest average cost of the uniform crossover.

cost function are of comparable magnitude. The approximate location of this transition region can be obtained by Eqs. 3.52 and 3.53. The figures indicate that the difference in convergence rate between cluster and uniform crossover, becomes smaller for decreasing dominance of the communication term (increas-
4.6 Experimental results and discussion

Figure 4.12: Cost evolution of the best individual in the population for cluster- and uniform crossover, using the following parameters: \( n = 64, P = 8, \gamma = 0.2, \beta = 0.3 \).

Figure 4.13: Cost evolution of the best individual in the population for cluster- and uniform crossover, using the following parameters: \( n = 64, P = 8, \gamma = 0.8, \beta = 0.3 \).

Furthermore, we observe that outside the transition region the final value of the average cost is approximately equal for both methods. However, in the neighborhood of this region the final value of the average cost is lower if cluster crossover is used. This is a consequence of the fact that both terms are of comparable magnitude. Hence, both connectivity as well as work load balance
are important. The selection mechanism tries to filter optimal building blocks, followed by the destructive effect of uniform crossover on optimally connected sub clusters. This combined operation accounts for the poorer solution quality in the case of uniform crossover. Besides the algorithmical convergence aspects, it is important to consider the additional overhead associated with the cluster crossover. One would expect severe computational costs for constructing sub clusters from the task graphs. However, the extra overhead is only manifest in additional memory requirements, in terms of an extra data structure of size $n^2$, which contains for each task a list of task numbers associated with cluster sizes up till $n - 1$. The cluster crossover utilizes this data structure to construct clusters of arbitrary sizes.

### 4.6.2 Evolutionary strategies compared

Although both SA and GA can be mapped onto the same computational model, they are inherently different. For example, SA deals with cooling schedules, the length of the Markov chains and starting temperatures, whereas GA entails population sizes, crossover, mutation probabilities, etc. As a consequence, it is difficult to compare the methods even qualitatively. An additional problem is that for each method a proper balance between turn around time and solution quality must be defined. This balance is hard to realize, other than by using empirical methods. Besides these considerations, the actual performance of each algorithm depends on implementation issues. However, despite all these peculiarities, we will compare the performance of the three strategies.

Below, the solution qualities of a CGA, CSA without interactions, and independent CSD optimization experiments are compared. The population size for the CGA is fixed, while both for CSA and CSD, the number of independent individuals is equal to the number of processors. After the CGA and CSA have reached convergence, a full SD procedure follows to get to the nearest local optimum. The ACGA algorithm has been executed on 16 processors of a 40-node Parsytec CC machine using the MPI message passing library [50]. Since CSA and CSD instances are used without interactions, a single individual can be assigned to each processor, in order to exploit maximum concurrency. Both CSA and CSD have been performed with 16 individuals, the best of which is chosen at the end of each run. The results are averaged over 25 runs.

For the CGA a population of 256 individuals is used, along with a “standard” mutation probability of $1/n$ and a cluster crossover probability of 0.7. The initial CGA population consists of a uniform mix of chromosomes with both randomly assigned processor IDs, and fixed processor IDs, leading to a balanced representation of both parallel and sequential allocations. This can be interpreted as yet another kind of problem specific enhancement, because it is known from Section 4.4.1, that sequential as well as parallel optimal allocation regimes exist. Experiments performed with only random chromosomes in the initial population, resulted in significantly higher average costs in the intermediate parallel ($\beta \approx \beta_c$ and $\gamma \approx \gamma_c$) and sequential allocation regimes (data not shown).
4.6 Experimental results and discussion

<table>
<thead>
<tr>
<th>$\beta$</th>
<th>CGA</th>
<th>CSA</th>
<th>CSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.10</td>
<td>640.0(0.0)</td>
<td>640.0(0.0)</td>
<td>640.0(0.0)</td>
</tr>
<tr>
<td>0.12</td>
<td>768.0(0.0)</td>
<td>768.0(0.0)</td>
<td>768.0(0.0)</td>
</tr>
<tr>
<td>0.14</td>
<td>893.3(0.2)</td>
<td>892.4(0.0)</td>
<td>893.4(0.4)</td>
</tr>
<tr>
<td>0.16</td>
<td>983.6(0.9)</td>
<td>982.3(1.0)</td>
<td>994.7(7.6)</td>
</tr>
<tr>
<td>0.18</td>
<td>1041.8(1.9)</td>
<td>1037.7(2.2)</td>
<td>1051.6(6.5)</td>
</tr>
<tr>
<td>0.20</td>
<td>1063.6(4.9)</td>
<td>1052.9(1.8)</td>
<td>1074.1(6.3)</td>
</tr>
<tr>
<td>0.22</td>
<td>1066.0(5.6)</td>
<td>1057.2(1.9)</td>
<td>1070.9(6.7)</td>
</tr>
<tr>
<td>0.24</td>
<td>1064.3(5.4)</td>
<td>1054.0(1.2)</td>
<td>1066.8(3.6)</td>
</tr>
<tr>
<td>0.26</td>
<td>1057.3(6.0)</td>
<td>1046.8(1.4)</td>
<td>1061.6(6.1)</td>
</tr>
<tr>
<td>0.28</td>
<td>1045.2(5.5)</td>
<td>1035.6(1.1)</td>
<td>1047.2(3.4)</td>
</tr>
<tr>
<td>0.30</td>
<td>1030.1(4.6)</td>
<td>1023.7(1.6)</td>
<td>1034.9(3.1)</td>
</tr>
</tbody>
</table>

Table 4.3: Average cost (and standard deviation) of optimal allocations found by CGA, CSA and CSD. A random task graph is used with $n = 80$, $P = 16$ and $\gamma = 0.25$.

For CSA, Markov chains are generated of length $10n$. The starting temperature was chosen such that the acceptance ratio started at 70%. Geometrical cooling is applied with an exponent of 0.95, that is $T(i+1) = 0.95T(i)$, with $T(i)$ the temperature at the $i$-th Markov chain.

The first series of optimization experiments has been performed on random task graph instances with $n = 80$, $P = 16$, and $\gamma = 0.25$. For this specific value of $\gamma$, the break even point from optimal sequential to optimal parallel allocation (see Section 4.4.1) is approximately $\beta_c = 0.2$ (using Eq. 3.52). In Chapter 3 we have argued that the most “difficult” TAP instances for fully connected random task graphs are found near this break even point. As shown in Figs. 4.10, 4.11 and 4.12, the convergence characteristics differ for each of the three regions: $\beta < \beta_c$, $\beta \approx \beta_c$ and $\beta > \beta_c$. Therefore, the optimization methods are tested on instances for $\beta$ in all three regions. The following results (see Table 4.3) have been obtained for $\beta$ varied in the range $[0.10 - 0.30]$.

Table 4.3 summarizes the final solution quality for the three different algorithms depending on $\beta$. The average execution time (wall clock) is displayed in Fig. 4.14.

A second experiment, with the same parameters for the optimization algorithms has been carried out with $n = 128$, $P = 32$, $\beta = 0.1$, and $\gamma \in [0.04, 0.24]$. The results for the final average costs are displayed in Table 4.4, for $\gamma$ varied between 0.04 and 0.24. The turn around times are displayed in Fig. 4.15.

The optimal allocations for $\beta = 0.10, 0.12, 0.14$ in Table 4.3 and $\gamma = 0.20, 0.22, 0.24$ in Table 4.4, correspond to sequential allocations, i.e., $P = 0$. The highest $\beta$ values in Table 4.3, and the lowest $\gamma$ values in Table 4.4, correspond to parallel allocations, i.e., $P \approx 1$.

Tables 4.3 and 4.4 show that the overall convergence quality is best for CSA in almost all instances, which could be expected from the fact that the TAP energy landscape has self-similar properties (see Section 3.3).

Furthermore, the solution quality of the CGA method is better than that of CSD.
Table 4.4: Average cost (and standard deviation) of optimal allocations found by CGA, CSA and CSD. A random task graph is used with \( n = 128 \), \( P = 32 \) and \( \beta = 0.1 \).

<table>
<thead>
<tr>
<th>( \gamma )</th>
<th>CGA</th>
<th>CSA</th>
<th>CSD</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.04</td>
<td>402.7(4.6)</td>
<td>396.1(1.7)</td>
<td>411.0(3.3)</td>
</tr>
<tr>
<td>0.06</td>
<td>647.2(4.7)</td>
<td>636.7(2.8)</td>
<td>654.4(3.5)</td>
</tr>
<tr>
<td>0.08</td>
<td>890.7(6.4)</td>
<td>877.2(3.1)</td>
<td>899.0(4.1)</td>
</tr>
<tr>
<td>0.10</td>
<td>1115.7(7.7)</td>
<td>1107.4(3.5)</td>
<td>1135.5(6.0)</td>
</tr>
<tr>
<td>0.12</td>
<td>1329.8(10.0)</td>
<td>1318.7(6.7)</td>
<td>1353.9(6.8)</td>
</tr>
<tr>
<td>0.14</td>
<td>1480.7(6.2)</td>
<td>1460.2(7.2)</td>
<td>1501.5(12.7)</td>
</tr>
<tr>
<td>0.16</td>
<td>1606.1(1.0)</td>
<td>1617.0(7.8)</td>
<td>1615.8(9.9)</td>
</tr>
<tr>
<td>0.18</td>
<td>1638.2(0.0)</td>
<td>1639.9(3.7)</td>
<td>1638.5(1.3)</td>
</tr>
<tr>
<td>0.20</td>
<td>1638.4(0.0)</td>
<td>1638.8(1.5)</td>
<td>1638.4(0.0)</td>
</tr>
<tr>
<td>0.22</td>
<td>1638.4(0.0)</td>
<td>1638.4(0.0)</td>
<td>1638.4(0.0)</td>
</tr>
<tr>
<td>0.24</td>
<td>1638.4(0.0)</td>
<td>1638.4(0.0)</td>
<td>1638.4(0.0)</td>
</tr>
</tbody>
</table>

Figure 4.14: Average execution time for a single optimization run for CSA, CSD and CGA (\( n = 80 \), \( P = 16 \) and \( \gamma = 0.25 \)). \( \beta \) is on the horizontal axis.

On the other hand, the computational overhead is significantly higher for CGA, specifically in the parallel allocation regime (see Figs. 4.14 and 4.15).

The standard deviation of the final solution costs is a measure for the difference in cost between the various local optima. Tables 4.3 and 4.4 indicate that these deviations are large in the proximity of the predicted transition region, which is a result of the fact that the difference between task mutations is most sensitive in this region. This phenomenon is most pronounced for the CSD method, because it is a direct measure for the number of local optima (see Chapter 3).

The fact that this deviation is shifted from the predicted values of \( \beta_c \) and \( \gamma_c \) can
be ascribed to finite size effects (see Chapter 3). That is, the theoretical values are only correct for infinite task graphs.

The increase of the CSD execution time, with $\gamma$, in Fig. 4.15, is a consequence of the fact that the starting configurations consist of randomly chosen genes. Because most of the genes will have to change to the same value, as the sequential regime is approached (increasing $\gamma$), resulting in more CSD steps. The calculation of one SD step is of order $O(nP)$, therefore this effect is less pronounced, but nevertheless present (decreasing $\beta$), in Fig. 4.14, where only 80 tasks and 16 processors are used.

4.7 Conclusions

Evolutionary algorithms have gained much attention over the last years as optimization strategies for hard combinatorial problems, because there are several advantages in using such methods. The main advantage comes from a biased selection scheme in combination with a heuristic perturbation operator. This induces a search mechanism which is less sensitive for getting stuck in local minima than deterministic methods. Also, Evolutionary methods can be applied to a large class of problems without altering the fundamental principles of the method. However, some care has to be taken when designing an evolutionary approach for a particular problem. In this chapter we have discussed the application of such methods to a well known problem in parallel computing: allocation of parallel tasks onto a parallel platform; the task allocation problem. A cellular framework for evolutionary algorithms has been formulated, in
which three different strategies have been embedded: GA, SA and SD. An advantage of this framework is that it allows for a generic way to exploit computational concurrency in the algorithms, allowing transparent implementation of the strategies into a single tool [36]. The actual implementation of the ACGA framework has been done using P-CAM.

Preliminary studies of a GA, instantiated in the ACGA framework, applied to mapping a finite element mesh onto a parallel machine have shown that the algorithm is highly scalable and is able to find good minima compared to GAs that use global selection methods.

Both the representation of the solutions as well as the structure of the cost function that is used to quantify the allocation quality, allow for problem specific enhancements. More specifically, in a CGA, a crossover mechanism exploits low cost solutions of sub problems in order to enhance the search, the application of which indeed showed an increased convergence rate over standard uniform crossover. The specific form of the TAP cost function can be used to enhance the performance of the calculation of mutated allocations. The cost of a local perturbation, that is, a transfer of a single task to another processor, can be computed with a constant time algorithmic overhead for task graphs and processor topologies of arbitrary sizes. This specific characteristic can be exploited for the local perturbation mechanisms in CSA and CSD.

In Chapter 3 it has been established that the TAP energy landscape is self-similar or AR(1). These landscapes are known from other fields like computational biology (e.g., RNA free energy [49]), theoretical physics (e.g., spin glasses [49]), and combinatorial optimization (e.g., TSP [150]). It has been shown that Simulated Annealing performs optimally on such landscapes [147]. Our experimental results indicate the superiority of Simulated Annealing over Genetic Algorithms and Steepest Descent as applied to the Task Allocation Problem.