Scheduling Using Genetic Algorithms

Ursula Fissgus
Computer Science Department
University Halle-Wittenberg
06099 Halle (Saale), Germany
fissgus@informatik.uni-halle.de

Abstract

We consider the scheduling of mixed task and data parallel modules comprising computation and communication operations. The program generation starts with a specification of the maximum degree of task and data parallelism of the method to be implemented. In several derivation steps, the degree of parallelism is adapted to a specific distributed memory machine. We present a scheduling derivation step based on the genetic algorithm paradigm. The scheduling takes not only decisions on the execution order (independent modules can be executed consecutively by all processors available or concurrently by independent groups of processors) but also on appropriate data distributions and task implementation versions. We demonstrate the efficiency of the algorithm by an example from numerical analysis.

Key words*: scheduling, distributed memory machines, task parallelism, data parallelism, genetic algorithms.*

1 Introduction

Several applications from scientific computing, e.g., from numerical analysis [5], signal processing [15], and multidisciplinary codes like global climate modeling [1] enclose different kinds of potential parallelism: task parallelism and data parallelism. *Task parallelism* results from the structure of the program. Independent program parts can be executed in parallel on disjoint groups of processors. *Data parallelism* is the result of data independences. It allows that the same operations can be applied concurrently on different data.

The detection and efficient exploitation of different kinds of parallelism often leads to efficient parallel programs. Right from start, when generating a parallel program, it is difficult to determine how to exploit the parallelism detected in order to obtain an optimal implementation on a specific parallel machine. Using pure data parallelism, we have small computation costs and possibly high collective communication costs. Using task parallelism, we reduce the number of participating processors for a task and this on its part reduces the internal communication costs. On the other hand, a reduced number of processors increases the pure computation time. The management of processor groups involved, however, also causes some overhead.

Because of non-linear execution times for the tasks and the communication operations, it is usually difficult to select the most efficient execution order and to determine the size of processor groups accordingly. The general problem is NP-complete [4], several approaches have been proposed. Recently, Rauber and Rünger presented a scheduling algorithm based on the principle of dynamic programming [13]. There are also various genetic algorithm based approaches working on heterogeneous [18] or homogenous [9] computing environments and using measured, i.e., known and fixed, costs. This approaches work with tasks with known computation behaviour and non-varying data distribution.

We present an algorithm based on a genetic algorithm (GA) approach that provides a static scheduling solution. Our algorithm not only provides a scheduling of a given set of tasks, but also selects for each particular task the suitable implementation (taken, e.g., from a predefined set of library functions) and the appropriate data distributions, and specifies the data redistributions, if necessary. The algorithm is based on runtime prediction formulas proposed by Rauber and Rünger [12], [14] and Dierstein [3], and adapted (Sect. 2.3, [8]) for the requirements of the scheduling algorithm.

In the following, we describe how the scheduling algorithm works. First, we outline the programming model and describe the elements needed for scheduling in Section 2. Especially, this section goes into the GA coding of the problem and the details of the runtime estimation. Section 3 presents the GA approach. The structure of the GA chromosomes is explained, a detailed specification of the GA operations mutation and crossover is given, and the fitness function, i.e., the function which evaluates the quality of a solution, is described. Section 4 shows experimental results.

Finally, Section 5 discusses some future research directions.

2 Model Overview

The TwoL model [11], [5] clearly separates task parallelism from data parallelism. This clear separation is used to divide the work between programmer and compiler system. The programmer specifies an algorithm as a module specification and provides a specification of the data parallel basic modules. The module specification expresses the maximum degree of task parallelism of the method to be implemented, basic modules express parallel operations on data. The compiler adapts this degree of parallelism to the facilities of a target machine by translating the specification into a parallel program, e.g., a C program with MPI calls. The compiler translation consists of several transformation steps, including scheduling, see [5] for more details. The scheduling decides on the actual execution order of the tasks and on the processor groups mapped to the tasks. The transformation steps are guided by a runtime estimation of the resulting program.

2.1 Module Specification

A *module specification* is a non-executable program which (only) indicates the maximal degree of parallelism without giving an exact execution order of tasks or specifying a data distribution of variables. A module specification is hierarchically structured and consists of a set of composed modules. *Composed modules* only specify the structure of the task parallelism, direct operations on data are hidden within basic modules. Tasks in composed modules may be executed concurrently (|| operator, parallel loops) or sequentially (∘ operator, sequential loops). There is no dependence between modules executed in parallel. Data parallelism is expressed by *basic modules* provided as executable parallel programs operating on arbitrary data.

As an example, see in Fig. 1 a fragment of the program of the conjugate gradient method (cf. Section 4.1). The composed module `iter_loop` has an I/O behaviour which is specified by `IN`/`OUT` parameters. The basic modules are `mv_prod`, `vv_prod`, and `op_assign`. The `mv_prod` module must be executed first, before the `vv_prod` modules can be started. The different instances of `vv_prod` may be executed in parallel. The computation ends with the execution of the module `op_assign`.

2.2 Module Graphs

From the scheduling point of view, a composed module contains two significant information: the possible task execution order and the parameter interdependence. For our purposes, we concentrate this information into an extended module graph (a directed acyclic graph) which we use in the scheduling step, later on. Fig. 2 illustrates the *module graph* corresponding to the composed module of Fig. 1. The computation order is expressed by the structure edges, the parameter interdependence is expressed by the data edges.

```
module iter_loop (IN A : MAT, p : VEC, w : VEC, w0 : VEC,
                  r0 : VEC, OUT p : VEC)
mv_prod (IN  A : MAT, p : VEC, OUT  w : VEC)
∘ (vv_prod (IN p : VEC, r0 : VEC, OUT tmp1 : SCAL)
    || vv_prod (IN w : VEC, p : VEC, OUT λ1 : SCAL)
    || vv_prod (IN w : VEC, w : VEC, OUT tmp2 : SCAL)
    || vv_prod (IN w : VEC, w0 : VEC, OUT tmp3 : SCAL) )
∘ op_assign (IN tmp1 : SCAL, op : SCAL, λ1 : SCAL,
             OUT ξ : SCAL)
```

Figure 1. Composed module.

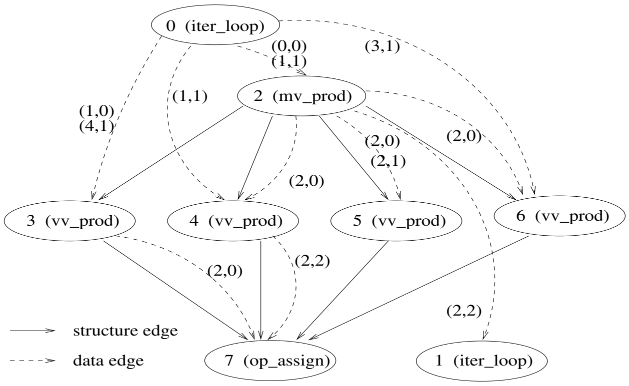

Figure 2. Module graph.

The *nodes* of the module graph correspond to the inner modules of the composed module expression (nodes 2 to 7 of Fig. 2). For technical reasons, a composed module `C` is represented by two nodes: the node C_0 represents the *input node* of the graph, the node C_1 represents the *output node* of the graph (Fig. 2, nodes 0 and 1, respectively).

The module graph contains two kind of edges: structure edges and data edges. *Structure edges*, drawn in Fig. 2 by solid lines, result from the structure of the module expression. Structure edges illustrate the precedence relation between modules, expressing whether a module have to be executed before another one. For example, for a module expression $A \circ B$ with a sequential compose operator, the module graph contains the oriented edge (A, B). Modules connected via a || operator do not have any connecting edges, alike there are no structure edges from/into the nodes representing the composed module.

Data edges, drawn by dashed lines, describe the parameter data interdependence. We use the usual validity rules for variables in nested blocks, the notions of *successor* and *predecessor* have to be applied accordingly. If module $\mathtt{A}$ is a predecessor of $\mathtt{B}$ and $\mathtt{B}$ has an input parameter which is an output parameter of $\mathtt{A}$, then there is a directed data edge $(\mathtt{A}, \mathtt{B})$ in the module graph. Annotations to data edges store additional information about the corresponding parameters. In Fig. 2 the annotations store the index of the parameters in the corresponding parameter list, e.g., annotation (2,0) of edge (2,4) means that parameter 2 of module 2 and parameter 0 of module 4 correspond.

There are special rules to be applied to the parameters of the composed module $\mathtt{C}$. If $\mathtt{a}$ is an input parameter of $\mathtt{C}$ and $\mathtt{A}$ is an inner module having $\mathtt{a}$ as an input parameter, then an oriented data edge $(\mathtt{C}_0, \mathtt{A})$ is generated. Analogously, for an output parameter of the composed module an oriented edge $(\mathtt{A}, \mathtt{C}_1)$ is generated.

Note that a valid schedule of the composed module corresponds to a *topological sort* of the module graph and vice versa. (If there is a structure edge $(\mathtt{A}, \mathtt{B})$, then the task implementing module $\mathtt{A}$ has to finish its execution, before the task implementing module $\mathtt{B}$ is started. If $(\mathtt{A}, \mathtt{B})$ is a data edge, then the data transfer has to be completed, too.)

2.3 Runtime Estimation

The most important evaluation criteria of a parallel programm is the global execution time, i.e., the time between start and termination of the computation of a program. This time consists of time needed for task computation and time needed for communication between tasks, i.e., for data (re)-distribution. Task computation time depends on the algorithm implemented and on the data distribution used. Data (re)distribution time only depends on the data distributions used. The choice of a suitable data distribution is an important issue of an efficient global execution time.

Technically, data distribution is accomplished by means of a (bijective) map from a global index set to a set of local index sets. In order to determine an optimal data distribution that leads to a minimal global execution time, we consider theoretical execution time functions that include information about various data distributions as parameters. For describing data distribution we adapt the *parametrized data distribution* proposed in [3] and [14]. According to this description, each data distribution can be described by a *distribution vector*. We use this description in order to estimate the volume of data to be transfered for (re)distributing data. Because of given practical facts, we assume that data is evenly distributed among the processors and, thus, the communication costs are evenly distributed. Each one of the initial processors communicates with each one of the processors of the final distribution. The communication itself is realized by single-to-single transfer, which depends on the parameters of the target machine model used: number of processors, execution time for an arithmetic operation, startup time for point-to-point communication, byte transfer time, bandwidth of the interconnection network, see [2], [12], [8] for details.

A composed module consists of tasks which represent basic modules, i.e., modules which are computed in a data parallel manner. The computation time of a basic module can be a measured time, e.g., for already available library functions, or it can be an analyticaly derived estimation function. We use a runtime estimation model for SPMD programs in message-passing style [11]. Investigations for several applications from numerical analysis show that the runtime prediction formulas describe the execution time accurately enough to compare different execution schemes of the same application [14], [8].

3 Genetic Algorithms

The usual form of genetic algorithms (GAs) was described by Goldberg [7]. GAs are stochastic search techniques based on the mechanism of natural selection and natural genetics. They start with an *initial population*, i.e., an initial set of random solutions which are represented by chromosomes. The chromosomes evolve through successive iterations, called *generations*. During each generation, the solutions represented by the chromosomes of the population are evaluated using some measures of *fitness*. To create the next generation, new chromosomes are formed. The chromosomes involved in the generation of a new population are selected according to their fitness values. Fitter chromosomes have higher probabilities of being selected. After several generations, the algorithm converges to the best chromosome which represents the (sub)*optimal solution* of the problem.

3.1 The Chromosomes

An individual (*chromosome*) represents a solution of the scheduling problem by giving the relative execution order of the tasks, by selecting for each task an appropriate version, and by associating to each task a group of processors. A chromosome consists of 3 substrings of the same length: task, version, and processor part. Three genes (t, v, p) belong together. The meaning is: a group of p processors is available for the execution of version v of task t.

A gene of the *task part* represents a task (a basic module). Each task occurs exactly once. Remember, that there are interdependencies between tasks expressing whether two tasks have to be executed sequentially or concurrently which have to be met by a valid schedule. Thus, the tasks in a valid schedule are topologically sorted.

A gene of the *version part* represents a version of the corresponding task. Each task has one or more versions,

0	1	2	3	4	5	6	7	position
0	2	6	5	4	3	7	1	task part
14	8	3	2	1	0	0	14	version part
4	4	1	4	4	4	1	4	procs part

mutation on version part position 2

0	1	2	3	4	5	6	7	position
0	2	6	5	4	3	7	1	task part
14	8	4	2	1	0	0	14	version part
4	4	2	4	4	4	1	4	procs part

Figure 3. Version part mutation operation.

0	2	3	6	5	4	7	1	task part
12	10	1	1	0	3	0	12	version part
4	4	4	4	4	1	1	4	procs part

mutation on task part position 2, valid range is 2..5
new position (randomly selected) = 5

0	1	2	3	4	5	6	7	position
0	2	6	5	4	3	7	1	task part
12	10	1	0	3	1	0	12	version part
4	4	4	4	1	4	1	4	procs part

Figure 4. Task part mutation operation.

e.g., different library functions, which differ by the algorithm implemented, by the data distribution of parameters, or by the number of processors needed for execution. A gene of the *processor part* represents the number of processors available for executing the corresponding task.

Note that a chromosome specifies unambiguously each task by two genes: one gene for the task id and another one for the version used. However, the chromosome stores only the number of processors associated to a task. An exact specification of the processor group actually used is made by additional operations, see 3.4 for more details.

All examples in the next subsections are chromosomes which code a schedule of the composed module of Fig. 1. The task ids refer to the nodes of the module graph representation from Fig. 2.

3.2 Mutation Operation

The mutation operation creates a new individual by randomly changing a randomly selected individual of the population. The position where the operation is performed is randomly selected, too. There is no guarantee, that a mutation operation really achieve a change. For some values there is no valid alternative. A scheduling chromosome fulfills some rules and it is not always possible to make a change at a randomly selected position without violating these rules.

Different mutation operations are developed for different parts of the chromosome. The *processor part mutation* operation randomly changes the number of processors of a task. This operation is possible only when the corresponding task version does not expect a well defined number of processors.

The *version part mutation* operation randomly choose a new version, if available, of the corresponding task. Accordingly, the corresponding number of processors is adapted, if necessary. Fig. 3 shows the mutation operation on version part position 2. The new version 4 of task 6 is randomly selected; this version needs 2 processors.

The *task part mutation* operation is more complex. Let $\mathtt{t_i}$ be a task. Let $\mathtt{inlist(t_i)} = \{\mathtt{t_j} : \mathtt{t_j}$ task with $(\mathtt{t_j}, \mathtt{t_i})$ edge$\}$ be the set of direct predecessors of task $\mathtt{t_i}$, and $\mathtt{outlist(t_i)} = \{\mathtt{t_j} : \mathtt{t_j}$ task with $(\mathtt{t_i}, \mathtt{t_j})$ edge$\}$ be the set of direct successors of $\mathtt{t_i}$.

The *valid range* of a task $\mathtt{t_i}$ on position $\mathtt{i}$ in a chromosome is the set of positions $[\mathtt{pos_1..pos_2}]$, $\mathtt{pos_1} \leq \mathtt{i} \leq \mathtt{pos_2}$, at which this task can be placed without violating the interdependence rules, i.e.,

$$pos_1 = \begin{cases} 0, \text{ if inlist}(t_i) = \phi \\ \max\{j : \ t_j \in \text{inlist } (t_i)\} + 1 \end{cases}$$

$$pos_2 = \begin{cases} \text{max\_position, if outlist}(t_i) = \phi \\ \min\{j : \ t_j \in \text{outlist } (t_i)\} - 1 \end{cases}$$

In particular, the valid range starts behind all tasks which have an outgoing edge ending at the selected task $\mathtt{t_i}$ and it ends before any task having an ingoing edge which starts at $\mathtt{t_i}$. The valid range contains only tasks which do not have interdependence relations with the selected task $\mathtt{t_i}$ and, thus, can be executed concurrently with it.

The task part mutation operation randomly moves the selected task within the valid range, the corresponding versions and processor numbers are moved accordingly. The valid range of a task can consist of only one element (the initial position of the task) and the chromosome does not change. Fig. 4 illustrates the task part mutation operation. The valid range of task 3 from position 2 is between the positions 2 and 5. The new position 5 is randomly selected and the genes are moved accordingly.

3.3 Crossover Operation

The crossover operation creates a new individual by combining parts taken from two parent individuals. At first, the child is a copy of its first parent. Then we randomly select the crossover position and perform the actual crossover operation by introducing information from the second parent into the child chromosome. Different crossover operations are developed for different parts of the chromosome.

The *processor part crossover* and the *version part crossover* operations substitute the processor and version information of the child by the corresponding information of first parent

0	2	3	5	4	6	7	1	task part
22	12	0	2	0	0	0	22	version part
2	4	4	4	4	4	1	2	procs part

second parent

0	2	6	5	3	4	7	1	task part
14	8	3	2	0	1	0	14	version part
4	4	1	4	4	4	1	4	procs part

crossover on version part position 3

0	1	2	3	4	5	6	7	position
0	2	3	5	4	6	7	1	task part
14	12	0	2	1	3	0	14	version part
4	4	4	4	4	1	1	4	procs part

Figure 5. Version part crossover operation.

first parent

0	2	3	5	4	6	7	1	task part
22	12	4	2	2	0	0	22	version part
2	4	2	4	4	4	1	2	procs part

second parent

0	2	6	5	4	3	7	1	task part
21	18	4	0	2	4	0	21	version part
4	1	2	4	4	2	3	4	procs part

crossover on task part position 4

0	1	2	3	4	5	6	7	position
0	2	3	5	6	4	7	1	task part
22	12	4	2	0	2	0	22	version part
2	4	2	4	4	4	1	2	procs part

Figure 6. Task part crossover operation.

the second parent. In order to obtain a consistent individual, both belonging information, i.e., version and processor number, are taken. The operations are performed starting at the selected position i and continuing until the end of the chromosome is reached. Fig. 5 shows a version part crossover operation on position 3. Starting at this positon, the version and processor information from the second parent is copied to the child chromosome, i.e., the information for the tasks 5, 4, 6, 7, and 1. Due to the special relation between task 0 and 1 (both represent the composed module `iter_loop`), the information of task 0 is also changed, accordingly to the values corresponding to task 1.

The *task part crossover* operation put the tasks of the child in the relative order of the tasks of the second parent. Only the tasks between the selected crossover position and the end of the list are considered. For a consistent individual, the belonging information, i.e., version and processor number, are moved accordingly without changing the values. In Fig. 6 the tasks 4, 6, 7, and 1 are put in the relative order of the second parent, i.e., 6, 4, 7, 1. Note that the resulting chromosome maintains the tasks topological sort.

3.4 Fitness Function

Fitness is the driving force of the Darwinian natural selection and, likewise, of GAs. It may be measured in many different ways. The accuracy of the fitness function with respect to the application is crucial for the quality of the results produced by the GA. Our fitness function evaluates the runtime of a program whose task schedule is represented by a chromosome. The costs include all computation and communication costs. Due to the structure of a schedule, the fitness evaluation process is not uniquely determined. Therefore we made the following arrangements.

- The evaluation function considers the tasks in the order they appear on the chromosome. Let $\mathtt{t_i}$, $\mathtt{t_j}$ be two tasks with $\mathtt{i} < \mathtt{j}$, then first the group of processors mapped to task $\mathtt{t_i}$ is computed before processor mapping for task $\mathtt{t_j}$ is performed.
- For each task $\mathtt{t_i}$, its input data has to be locally available, i.e., data transfers have to be completed, before the task execution starts: $\mathtt{start\_time(t_i)} \geq \mathtt{max}\{\mathtt{end\_time(t_j)} :\ \mathtt{i} > \mathtt{j}, (\mathtt{t_j}, \mathtt{t_i})\ \mathtt{data\ edge}\}$. Thus, communication costs are added completely to the source task of the communication.
- When starting the execution of a task $\mathtt{t_i}$, all processors executing $\mathtt{t_i}$ have to be available and they start the execution at the same time: $\mathtt{start\_time(t_i)} \geq \mathtt{max}\{\mathtt{time(proc\ q\ available)} : \mathtt{q\ executes\ t_i}\}$.
- Interdependent tasks, i.e., tasks with connecting module graph structure edges, are executed sequentially in the order specified by the chromosome, i.e., a task has to complete its execution before the successor can be started: $\mathtt{start\_time(t_i)} \geq \mathtt{max}\{\mathtt{end\_time(t_j)} : \mathtt{i} > \mathtt{j}, (\mathtt{t_j}, \mathtt{t_i})\ \mathtt{structure\ edge}\}$.
- Similarly for independent tasks computed by the same processors: $\mathtt{start\_time(t_i)} \geq \mathtt{max}\{\mathtt{end\_time(t_j)} : \mathtt{i} > \mathtt{j}, \mathtt{procs(t_j)} \cap \mathtt{procs(t_i)} \neq \phi\}$.
- There is no rule to be respected for independent tasks working on different processors.
- Computation costs which accrue when executing a task have to be considered when evaluating a task.
- Communication costs are associated to data edges. These costs are completely/partially added to the source or/and end task of a data edge, depending on the strategy used.

A chromosome codes the relative order of tasks, their versions, and knows the number of processors used for each task. Task id and version number exactly define an implementation of a task with a well defined data distribution. Thus, the computation costs can properly be evaluated.

In order to obtain accurate communication cost evaluations, an exact specification of the processors assigned to each task is necessary. This assignment is either randomly or we use different heuristics to do it. Fig. 7 gives an example of evaluating the fitness function. It uses 4 processors and refers to the example of Fig. 1 and 2. The communication costs are completely added to the costs of the task which initiates data transfer. We try to minimize costs by taking for a task the processors earliest available.

chromosome

0	2	4	6	5	3	7	1	task part
15	17	4	4	1	3	0	15	version part
4	4	2	2	4	1	1	4	procs part

computation costs

costs(task, version, nbr_of_procs)

t, v, p	costs	t, v, p	costs
0, 15, 4	0	4, 4, 2	29.931
1, 15, 4	0	5, 1, 4	50.475
2, 17, 4	688.046	6, 4, 2	29.931
3, 3, 1	10.1	7, 0, 1	0.1

communication costs

costs((task$_1$, version$_1$), (task$_2$, version$_2$))

t$_1$, v$_1$ \ t$_2$, v$_2$	1, 15	2, 17	3, 3	4, 4	5, 1	6, 4	7, 0
0, 15	–	0 33.25	4.3852 4.3852	8.7704	–	8.7704	–
1, 15	–	–	–	–	–	–	–
2, 17	0	–	–	8.7704	17.5408 17.5408	8.7704	–
3, 3	–	–	–	–	–	–	4.34015
4, 4	–	–	–	–	–	–	4.34015
5, 1	–	–	–	–	–	–	–
6, 4	–	–	–	–	–	–	–
7, 0	–	–	–	–	–	–	–

costs evaluation

t, v	procs	comp+ comm costs	start time	end time	max time
0, 15	0 1 2 3	63.9464	0	63.9464	63.9464
2, 17	0 1 2 3	758.209	63.9464	822.155	822.155
4, 4	0 1	34.2711	822.155	856.427	856.427
6, 4	2 3	29.931	822.155	852.086	856.427
5, 1	0 1 2 3	50.475	856.427	906.902	906.902
3, 3	2	14.4401	906.902	921.342	921.342
7, 0	3	0.1	921.342	921.442	921.442
1, 15	0 1 2 3	0	921.442	921.442	921.442

Figure 7. Fitness evaluation.

3.5 Initialization and Termination

There are several methods to initialize the GA. At the beginning of our experiments we generate each chromosome of the first population randomly. Experiments showed, that the results are better if the initial population includes 'good' individuals, e.g., if a part of the initial population is overtaken from a previous run of the GA.

The GA stops after having considered a predefined number of generation or if no improvement is obtaind during a predefined number of generations.

4 Experimental Results

As example, we consider the conjugate gradient method which has a potential of task and data parallelism.

4.1 Conjugate Gradient Method

The conjugate gradient (CG) method, a Krylow-space method to solve a system of linear equations $\mathtt{Ax} = \mathtt{b}$ with a symmetric, positive-definite coefficient matrix $\mathtt{A} \in \mathbb{R}^{\mathtt{M}\times\mathtt{M}}$ [17]. In each step $\mathtt{k}$, $0 \le \mathtt{k} \le \mathtt{K}-1$, the method chooses a search direction $\mathtt{p_k} \in \mathbb{R}^{\mathtt{n}}$ and computes a new approximation $\mathtt{x_k} \in \mathbb{R}^{\mathtt{n}}$.

Fig. 8 shows a possible module specification of an iteration step. First, a mv_prod() call is specified (line 5). Then, four independent vv_prod() calls can be computed in parallel (lines 6-9). After these computations, two groups of computations can be executed in parallel. One group (lines 10-14) computes the next approximation x and a residual r, the other group (lines 15-23) computes the next search direction p. Both computation groups have additional potential of parallel execution which is expressed by the corresponding || operators.

The information contained in the composed module specification is transformed into a module graph with 21 nodes (2 nodes for the composed module and 19 basic modules), 37 structure edges, and 38 data edges. For each basic module there are various variants which basically differ on the data distribution used for each parameter. We used for our experiments 28 different versions for the composed module, 28 versions for the mv_prod module, 5 versions for each one of the other vector operations (vv_prod, etc., a total of 14 modules), and only one possibility for the scalar operations (assign and op_assign, 4 modules). Theoretically, there are $28^2 \cdot 5^{14}$ possibilities to combine the module versions. Additional possibilities are caused by the different task permutations. The number of these permutations is determined by the number of operands of the || expressions; in our example there are $4! \cdot 3! \cdot (2!)^4$ possibilities.

Despite the volume of the search space, the GA run shows a good convergence to an optimal solution. Fig. 9

```
module iter_loop (IN A : MAT, p : VEC, p0 : VEC,
                  w : VEC, w0 : VEC, r : VEC, r0 : VEC, x : VEC,
                  λ0 : SCAL, OUT p : VEC, x : VEC, r : VEC)
for(k = 0..K)
(mv_prod (IN A : MAT, p : VEC, OUT w : VEC)
∘ (vv_prod (IN p : VEC, r0 : VEC, OUT tmp1 : SCAL)
   || vv_prod (IN w : VEC, p : VEC, OUT λ1 : SCAL)
   || vv_prod (IN w : VEC, w : VEC, OUT tmp2 : SCAL)
   || vv_prod (IN w : VEC, w0 : VEC, OUT tmp3 : SCAL))
∘ ((op_assign (IN tmp1 : SCAL, . . . , OUT ξ : SCAL)
     ∘ ((sv_prod (IN ξ : SCAL, . . . , OUT tmp4 : VEC)
          ∘ vv_add (IN x : VEC, . . . , OUT x : VEC))
        || (sv_prod (IN ξ : SCAL, . . . , OUT tmp5 : VEC)
             ∘ vv_sub (IN r : VEC, . . . , OUT r : VEC)) ) )
   || (((op_assign (IN tmp2 : SCAL, . . . , OUT μ : SCAL)
          ∘ sv_prod (IN μ : SCAL, . . . , OUT tmp6 : VEC))
        || vv_assign (IN w : VEC, OUT w0 : VEC)
        || (op_assign (IN tmp3 : SCAL, . . . , OUT ν : SCAL)
             ∘ (assign (IN λ1 : SCAL, OUT λ0 : SCAL)
                || sv_prod (IN ν . . . , OUT tmp7 : VEC)) ) )
      ∘ (vv_add (IN tmp6 : VEC, . . . , OUT tmp6 : VEC)
         || vv_assign (IN p : VEC, OUT p0 : VEC))
      ∘ vv_sub (IN w : VEC, tmp6 : VEC, OUT p : VEC)
) ) )
```

Figure 8. Composed module for the conjugate gradient iteration.

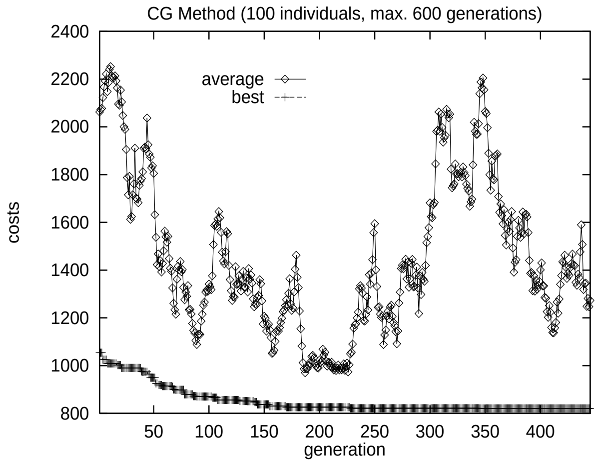

Figure 9. Evolution of costs.

illustrates the evolution of costs during a GA run with 100 chromosomes per generation and maximal 600 generations. In this example the GA run stops after about 500 generation because no improvement is achieved in the last 100 generations. A GA run with 100 individuals and 500 generations needs about three and half hours on a Sun UltraSparc. The problem size (vector size) was 3000 and we assumed a target machine with 4 processors. The GA has found the

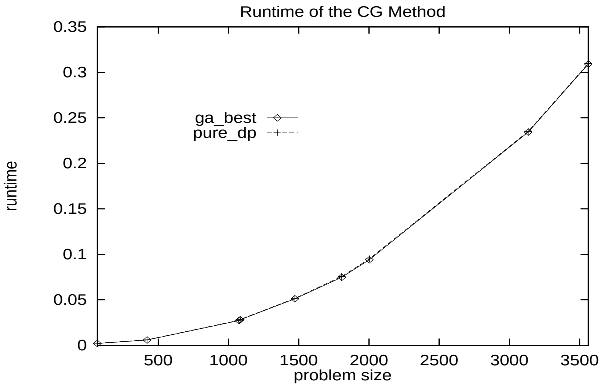

Figure 10. Runtime of the CG method for 4 processors on Cray T3E.

best implementation of the CG method, as an analysis of the possible implementations showed. The solution found mixes task and data parallelism and is slighly better than a pure data parallel solution. Fig. 10 illustrates the runtime for 4 processors on a Cray T3E; due to the low resolution and the small differences (less that 2%) between, the two graphs are overlapping. Fig. 11 illustrates the structure of this solution: the first four `vv_prod` calls and some scalar operations are executed in a task parallel manner, the remain of the operations are pure data parallel calls.

5 Conclusions and Future Work

The simultaneous exploitation of task and data parallelism can lead to significantly faster programs than sole exploitation of data parallelism. In this paper, we outlined a GA based scheduling algorithm that takes important design decisions in order to obtain efficient implementations.

The advantage of this approach is not only that it provides a scheduling algorithm, but also that it allows choosing the appropriate implementation from a set of functions and the convenient data distributions. The scheduling algorithm is part of the prototype of the TwoL system [11], [5]. The result of the scheduling step can be translated by a syntax directed pass in any imperative language augmented by a message passing library supporting groups [5]. For our experiments we chosed C as the imperative programming language and MPI as the message passing interface.

Future research includes scheduling for hierarchically structured module specifications and the exploitation of parallelism for the GA computations.

6 Acknowledgement

I would like to thank A. Thüring for making available the framework for generating genetic algorithms [16].

```
module iter_loop (IN A ROWBL, p ROWBL, p0 ROWBL,
          w ROWBL, w0 ROWBL, r ROWBL, r0 ROWBL,
          x ROWBL, λ0 : SCAL REPL,
          OUT p ROWBL, x ROWBL, r ROWBL, PROC 0..3)
   ... /* data redistribution */
for (k = 0..K)
( mv_prod (IN A ROWBL, p REPL, OUT w ROWBL, PROC 0..3)
  ∘ redist (IN w ROWBL, OUT w REPL(0..3), PROC 0..3)
  ∘ ( vv_prod (IN p LOC, r0 LOC, OUT tmp1 LOC, PROC 0)
      || vv_prod (IN w LOC, p LOC, OUT λ1 LOC, PROC 2)
      || vv_prod (IN w LOC, w LOC, OUT tmp2 LOC, PROC 1)
      || vv_prod (IN w LOC, w0 LOC, OUT tmp3 LOC, PROC 3)
    )
  ∘ redist (IN tmp1 LOC(0), OUT tmp1 LOC(2), PROC 0, 2)
  ∘ redist (IN λ1 LOC(2), OUT λ1 REPL, PROC 1..3)
  ∘ ( op_assign (IN tmp1 LOC, ..., OUT ξ LOC, PROC 2)
      || (op_assign (IN tmp3 LOC, ... OUT ν LOC, PROC 3)
          ∘ assign (IN λ1 LOC, OUT λ0 LOC, PROC 3))
      || op_assign (IN tmp2 LOC, ..., OUT μ LOC, PROC 1)
    )
  ∘ redist (IN ξ LOC(2), OUT ξ REPL, PROC 0..3)
  ∘ redist (IN μ LOC(1), OUT μ REPL, PROC 0..3)
  ∘ redist (IN ν LOC(3), OUT ν REPL, PROC 0..3)
  ∘ sv_prod (IN ξ REPL, ..., OUT tmp4 ROWBL, PROC 0..3)
  ∘ vv_assign (IN w ROWBL, OUT w0 ROWBL, PROC 0..3)
  ∘ sv_prod (IN ξ REPL, ..., OUT tmp5 ROWBL, PROC 0..3)
  ∘ sv_prod (IN μ REPL, ..., OUT tmp6 ROWBL, PROC 0..3)
  ∘ sv_prod (IN ν REPL, ..., OUT tmp7 ROWBL, PROC 0..3)
  ∘ vv_sub (IN r ROWBL, ..., OUT r ROWBL, PROC 0..3)
  ∘ vv_assign (IN p ROWBL, OUT p0 ROWBL, PROC 0..3)
  ∘ vv_add (IN x ROWBL, ..., OUT x ROWBL, PROC 0..3)
  ∘ vv_add (IN tmp6 ROWBL ... OUT tmp6 ROWBL, PROC 0..3)
  ∘ vv_sub (IN w ROWBL, ..., OUT p ROWBL, PROC 0..3)
  ... /* data redistribution (for looping) */
)
```

Figure 11. Sketch of the GA solution for the conjugate gradient iteration.

References

[1] H. Bal and M. Haines. Approaches for Integrating Task and Data Parallelism. *IEEE Concurrency*, 6(3):74-84, July-August 1998.

[2] D.P. Bertsekas, J.N. Tsitsiklis. *Parallel and Distributed Computation*. Prentice Hall, 1989.

[3] A. Dierstein, R. Hayer, and T. Rauber. The ADDAP System on the iPSC/860: Automatic Data Distribution and Parallelization. *Journal of Parallel and Distributed Computing*, 32(1):1-10, 1996.

[4] D. Fernandez-Baca. Allocating modules to processord in a distributed system. *IEEE Trans. on Software Engineering*, SE-15, 11:1427:1436, 1989.

[5] U. Fissgus, T. Rauber, and G. Rünger. A Framework for Generating Task Parallel Programs. In *Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation*, Annapolis, USA, 1999.

[6] I. Foster and K.M. Chandy. Fortran M: A Language for Modular Parallel Programming. *Journal of Parallel and Distributed Computing*, 1995.

[7] D. Goldberg. *Genetic Algorithms in Search, Optimisation, and Machine Learning*. Addison Wesley, 1989.

[8] L. Heinrich-Litan, U. Fissgus, St. Sutter, P. Molitor, and T. Rauber. Modeling the Communication Behavior of Distributed Memory Machines by Genetic Programming. *Proceedings of the 4th International Euro-Par Conference*, Southhampton, UK, 1998.

[9] E.S.H.. Hou, N. Ansari, and H. Ren. A Genetic Algorithm for Multiprocessors Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 5(2):113-120, 1994.

[10] S. Ramaswamy, S. Sapatnekar, and P. Banerjee. A Framework for Exploiting Task and Data Parallelism on Distributed-Memory Multicomputers. *IEEE Transactions on Parallel and Distributed Systems*, 1997.

[11] T. Rauber and G. Rünger. The Compiler TwoL for the Design of Parallel Implementations. In *Proc. 4th Int. Conf. on Parallel Architectures and Compilation Techniques*, IEEE, pages 282-301, 1996.

[12] T. Rauber and G. Rünger. PVM and MPI Communication Operations on the IBM SP2: Modeling and Comparison. *Proc. of the HPCS'97*, 1997.

[13] T. Rauber and G. Rünger. Scheduling of Data Parallel Modules for Scientific Computing. *Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing*, San Antonio, Texas, 1999.

[14] T. Rauber and G. Rünger. Deriving Array Distributions by Optimization Techniques. *Journal of Supercomputing*, 15(3), 2000.

[15] J. Subhlok and B. Yang. A New Model for Integrating Nested Task and Data Parallel Programming. *8th ACM SIGPLAN Symp. on Principles & Practice of Parallel Programming*, 1997.

[16] A. Thüring. Skriptbasierte Steuerung für parallele evolutionäre Algorithmen. Diploma thesis, University Halle, 2000.

[17] E.F. van de Velde. *Concurrent Scientific Computing*. Springer, 1994.

[18] L. Wang, H.J. Siegel, V.P. Roychowdhury, and A.A. Maciejewski. Task Matching and Scheduling in Heterogeneous Computing Environments Using a Genetic-Algorithm-Based Approach. *Journal of Parallel and Distributed Computing*, 47:8-22, 1997.