Brock University

Department of Computer Science

An Algorithm for Adaptive Maximization of Speedup

W. Wojcik and J. Martin
Technical Report # CS-03-09
September 2003

Brock University
Department of Computer Science
St. Catharines, Ontario
Canada L2S 3A1
www.cosc.brocku.ca

AN ALGORITHM FOR ADAPTIVE MAXIMIZATION OF SPEEDUP

W. Wojcik, J. Martin
Dept. of Computer Science
Brock University
St.Catharines, Ontario L2S 3A1
Canada

Abstract

We describe an algorithm for workload evaluation and auto-adaptive maximization of speedup. Emphasis is put on exploitation of medium- and fine-grain algorithmic parallelism. The simulation results suggest that significant speedups are achievable for communications-bound algorithms. The implications for design of programming languages and supercomputer design are suggested.

Key Words

Speedup, Simulation, Fine-Grain Parallelism, Adaptation.

1. Issues in Speedup Maximization

Current supercomputing practice is rather successful in obtaining demonstrable speedups when executing coarse-grain parallel algorithms of "embarrassingly parallel" nature. By "coarse-grain" we mean algorithms consisting of a number of compute-intensive steps, in which the steps are only slightly data-interdependent. The predominant supercomputing architecture is that of Beowulf, in which there is exactly one path between every two processing elements (PEs). Frequently this path contains gateways or switching elements, which introduce delays (latencies) and block other paths.

The traditional definition of algorithm defined as a finite sequence of steps leading from input data to the results (see Fig.1a) does not offer opportunities of parallelization. The "embarrassingly parallel" algorithms we successfully run today (see Fig.1.b) usually consist of executing the same procedures for different data sets.

We argue that supercomputer architectures in which the designers put the emphasis on the computing power of PEs while neglecting the issue of PE-to-PE communications are not suitable for extracting mid-grain or fine-grain algorithmic parallelism.

Indeed, Nature figured it out a long time ago. The PEs of a mammalian brain (neurons) are relatively slow; the power of the brain is predominantly due to their interconnect network. A typical neuron is connected to

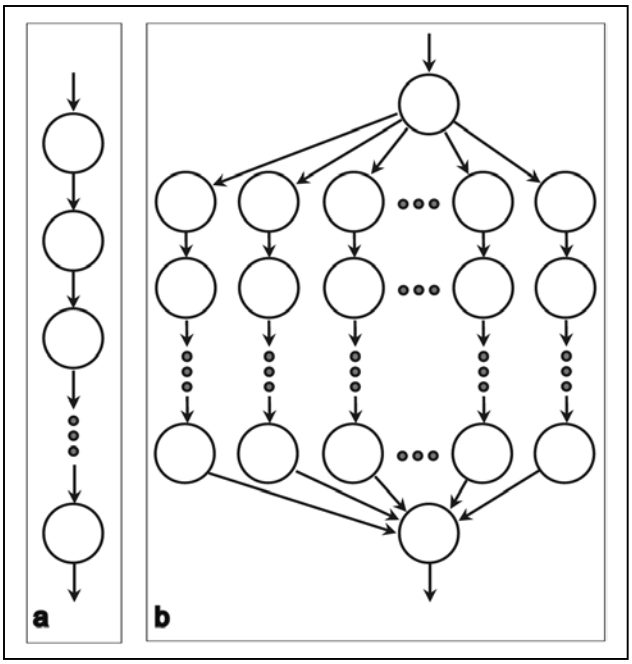

Fig.1. (a) A sequential algorithm represented as a directed acyclic graph (DAG) according to the traditional definition; (b) an "embarrassingly parallel" algorithm.

approx. 15 to 20 thousand other neurons, while the more specialized Purkinje cells have between 150 to 200 thousand connections to other neurons [1].

Our current supercomputing technology is nowhere that sophisticated in terms of PE-to-PE interconnectivity. Most disturbingly, however, we seem to be heading in the wrong direction: We put emphasis on increasing the computing power of individual PEs, while paying only secondary attention to the issue of interconnectivity. We persevere in this approach while being aware of the limits to the computing power that can be extracted from a single PE, due to the laws of physics. A more detailed analysis of this situation follows.

In order to suggest how to attain speedups when executing mid-grain and fine-grain parallel algorithms, we represent here all algorithms using directed acyclic

graphs (DAGs) as per Fig.1. Circles represent the computable steps there, while the arcs represent the data dependencies.

Given a particular algorithm and a parallel computer we may assign the algorithm's computable steps to different PEs. Such assignment, however, may result in delays due to the necessary data transfers (represented by arcs in our DAGs). On the other hand, assigning all computable steps to a single PE reduces to zero the times of all intermediate data transfers, while possibly resulting in delays due to overloading the PE.

The goal of this paper is to present the algorithm useful in determining which computable steps should be assigned different PEs in order to maximize speedup.

For a particular algorithm we define the speedup S as

$$S = T_1 / T_N$$

where T_1 is the time needed to execute the algorithm on a single-CPU machine, while T_N is the time needed to execute the same on a supercomputer with N PEs. There exist other definitions of speedup; one of the more formal definitions involves the comparison of the execution time of the best sequential algorithm solving the given problem to T_N. We find it impractical: the best algorithm of today may not be the best tomorrow; furthermore, users are unlikely to write two different programs solving the same problem only to know the details of speedup obtained.

2. The Simulation Approach

The algorithms we run today are far more sophisticated than those shown in Fig.1, resulting in much more complex DAGs. Fig. 2 shows a more representative DAG.

Observe that the nodes in a DAG can be seen as belonging to various levels, beginning with a single node at level 0. Once the execution of that node on a given PE terminates, execution of all nodes belonging to the level 1 may begin, as soon as the PEs are allocated to these nodes, and the data are transmitted to these PEs. The issue at hand is: Given a supercomputer of many interconnected PEs, which nodes should be allocated which processors, so that speedup is maximized? This allocation depends not only on the current load of all PEs, but also on the load of the interconnecting channels.

We may think of algorithms as having particular "shapes", corresponding to the shape of their DAGs. "Tall" and "slim" algorithms (i.e. with DAGs of many levels, with few nodes per level) are most suitable to run on uniprocessor machines; for supercomputers with many PEs we would prefer algorithms that are "short" and "fat". Sadly, our current programming languages make it exceedingly difficult to create medium- or fine-grained parallel programs that would be "fat" and "short".

A fine-grained parallel algorithm (regardless of its "shape") contains nodes that execute fast in comparison to the data transmission times of the arcs emerging from these nodes.

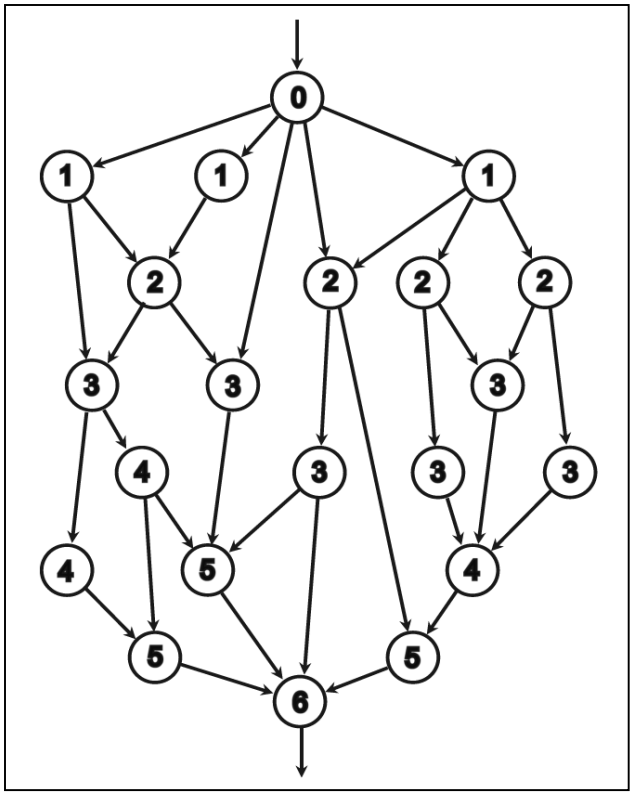

Fig.2. A typical DAG. Note the node precedence levels.

Is it possible to obtain significant speedups for fine-grained algorithms that are "fat" and "short"? Our simulation results (to follow) suggest that it is so.

Consider an elementary DAG and a simple, dual-PE machine shown on Fig.3 below.

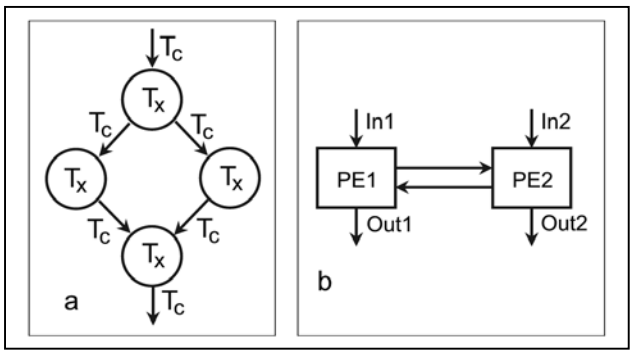

Fig.3. (a) A simple DAG to run on (b) a dual-PE machine.

All nodes of this DAG bear the same execution time T_x, all arcs bear the same transmission time T_c. Should this DAG be executed using two PEs? Not necessarily. The total execution time for this DAG, when ran on a single PE is

$$2T_c + 4T_x$$

(because the transmission time of all intermediate arcs reduces to zero), while if we allocated the two intermediate nodes to separate PEs, the parallel execution time would be

$$4T_c + 3T_x$$

These times would be equal if $T_x = 2T_c$ and for such a DAG it would not matter whether it were executed using one PE or two. However, if $T_x < 2T_c$ then execution on the single PE would be preferable. Conversely, if $T_x > 2T_c$ then the DAG would be suitable for parallel execution.

We note that the suitability for parallel execution is not the property of the DAG only, but it also depends on the machine available. A machine upgrade making the PEs faster while keeping the speed of the comms. channels constant reduces the suitability of DAGs for parallel execution (by reducing the effective T_x values), while bandwidth improvements to channels lead to greater DAG parallelizability.

3. The Parallel Simulator of Parallel Systems

Our parallel simulator, code named **SKULD**, is written in Ada95. It is capable of modeling a class of MIMD supercomputers depicted in Fig.4.

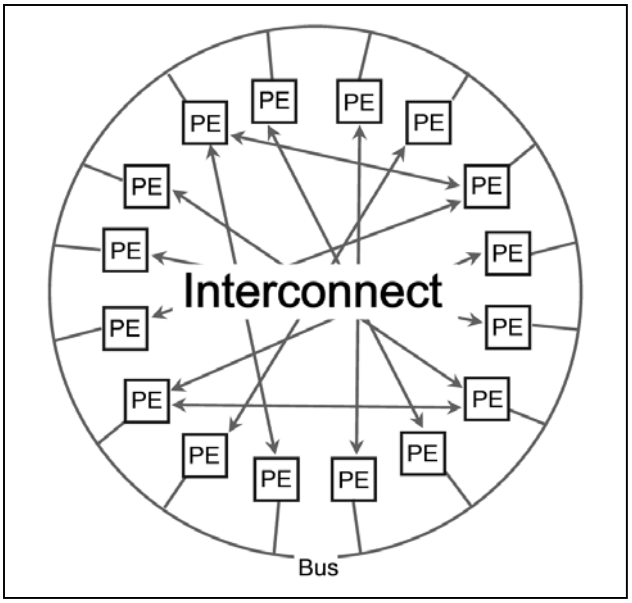

Fig.4. The class of MIMD supercomputers simulated by SKULD.

The simulated distributed-memory MIMD machines can consist of any number of PEs connected to the shared data bus. The PE-to-PE interconnect network consists of a number of unidirectional channels. A channel can connect any two PEs. A bi-directional connection between any two PEs (shown in Fig.4 as a double arrow) is obtained by using two unidirectional channels of opposite orientation. Each machine has its own clock showing simulated time.

The synthetic programs these machines execute are presented in form of DAGs of appropriate statistical characteristics. For each machine we can set the maximum number of DAGs that it can execute simultaneously (we call it the global MPX limit). If the number of DAGs submitted exceeds the machine's MPX limit, the extra DAGs are queued. Their execution will commence as soon as it is possible without exceeding the global MPX limit.

The supported standard architectures include full connect, data bus only, mesh, torus, hypercube (n-cube), tree, (un)shuffle exchange, star, pyramid, De Bruijn, butterfly, etc. Also supported are all kinds of random topologies, obtained by selectively providing the interconnect channels.

Not supported are shared-memory supercomputers. We decided not to support blocking interconnect networks with gateways and switches. After all, at a given moment a switch of n positions keeps one way open while keeping n-1 ways closed. It also introduces unnecessary latency and inhibits the use of multiplexing channels.

The shared data bus can transmit one message at a time, in any direction. It is used to perform the initial and final data transfer to and from each DAG. An attempt to transmit several messages overlapping in time results in the immediate transmission of the first message, while all other messages are being queued. With the transmission of the first message completed, the next message is de-queued and its transmission begins immediately.

We can set the transmission speed of the bus, as well as its overhead (measured as the percentage of transmission time lost on administrative tasks). We can set bus status to be active or inactive. An active bus can be used to execute any arc in a DAG; an inactive bus is allowed to execute only the input and output arcs of a DAG.

The processing elements (PEs) contain their own memories and have time-sharing capabilities (i.e. more than one node of a DAG can be executed on a given PE simultaneously). We can set the processing speed of each PE, its overhead (measured as the percentage of execution time lost on context switching), and overhead type (as one of O(n), O(n log n), $O(n^2)$, where n is the number of DAG nodes executed simultaneously). In order to limit the overhead time wasted by each PE, we can set the maximum number of nodes (MPX limit) a PE can execute simultaneously. If the number of nodes allocated to a given PE exceeds its MPX limit, the extra nodes are queued.

Thus there are three queues associated with every PE:

1. The execute queue, containing the DAG nodes currently in execution;

2. The ready queue, containing the nodes ready to execute as soon as the number of executing nodes drops below the PE's MPX limit;
3. The wait queue, containing nodes allocated to the PE but unable to execute until they receive all needed data.

The unidirectional channels also have multiplexing capabilities. One can set their speed, the overhead value and type (as for PEs), and their MPX limit.

There are two queues associated with the data bus and each channel:

1. The transmission queue, containing DAG arcs currently in progress;
2. The wait queue, containing arcs ready to transmit as soon as the number of concurrent transfers drops below the MPX limit (the MPX limit for the bus is 1).

4. The Simulation Experiment

It is rather obvious that speedups are easily attainable for compute-bound DAGs, i.e. for DAGs with $T_c << T_x$. We therefore decided to seek speedups for the comms-bound DAGs. Five families of DAGs were considered, differing in exponentially distributed T_c values with means of 0.25, 0.5, 1.0, 2.0, 4.0. Each family consisted of two sets of DAGs of shared statistical characteristics. The first set of 60 DAGs was intended to train the simulated supercomputers. The second set of 110 work DAGs was then used to measure the speedups.

The remaining DAG parameters were constant:

1. Mean $T_x = 1.0$ (exponentially distributed)
2. Mean number of node levels = 12.0 (Poisson distributed)
3. Mean number of nodes per level = 10.0 (again Poisson distributed)
4. Mean in-degree of a node = 2.0 (again Poisson distributed)
5. Mean difference between the child node level and its parent node level = 1.0 (Poisson distributed)

Therefore the typical "shape" of these DAGs was "square" (the number of interior levels in a typical DAG was equal to the number of nodes per level). Observe that with the mean T_c =0.25 and the in-degree of nodes being 2 the DAGs of the first family were balanced, while the DAGs of all other families were comms-bound.

Our reference simulated machine in the experiments was a computer with a single PE sitting on a bus.

Three experimental supercomputers had 16 PEs each. The first computer was the data-bus machine as per Fig.4 but without any interconnect channels. We had no choice but to use the active bus on this computer. The other two computers were a 4D hypercube and a fully connected machine, both with inactive buses.

The remaining settings for all four machines were:

1. Global MPX limit of 10
2. PE speed, bus speed and channel speed of 1.0
3. PE overhead, bus overhead and channel overhead of 1%
4. All overheads of type O(n)
5. PE MPX limit and channel MPX limit of 10
6. Bus MPX limit of 1

Each machine was first subject to training using the first set of DAGs from each family. The objective of the training was to minimize the execution time of first 50 DAGs in the training set, using our algorithm. In this way we assured that the training of the machine took place when the machine was working under full load.

Following training, each machine was made to process under full load the first 100 DAGs from the work set, and that execution time was noted and compared to the execution time of the reference machine in order to calculate speedup.

5. The Training Algorithm

Execution of a DAG on a simulated machine involves making a number of scheduling decisions. At the beginning we can use the data bus to assign a PE to the DAG's first node. Ideally that PE should be a member of a group of relatively idle PEs in our machine. We will then keep assigning neighbouring PEs to the subsequent nodes of our DAG. In this way we will create a region in our machine executing our DAG, an analogy to the regions of the brain that execute specialized algorithms like speech recognition, vision, etc.

In our simulation, when assigning interior nodes to PEs the candidate PEs are immediate neighbours to the PE which just finished executing a parent node (including that very PE). The least loaded PE is chosen from among the candidates, the load L of a PE being calculated after the tentative allocation of a node to the PE, viz.:

$$L(PE) = card(XQ)*(1+ovh) + x_2*card(RQ) + x_3*[card(WQ)+1]$$

where card( ) means the number of elements in a given queue; XQ , RQ and WQ stand for execute queue, ready queue and wait queue correspondingly, ovh is the overhead coefficient of that PE, and x_2, x_3 are weight coefficients that will be subject to optimization.

Similarly, the load of an idle channel CH, or a channel CH running under its MPX limit is

$$L(CH) = [card(TQ)+1]*(1+ovh)$$

while the load of a channel operating at its MPX limit is

$$L(CH) = MPX*(1+ovh) + x_5*[card(WQ)+1]$$

Again, x_5 is a weight coefficient, while WQ and TQ are wait and transmission queues, correspondingly.

The load on a data bus B is calculated analogously using coefficient x_4, and with MPX limit being 1.

The total cost of allocating a new PE to a node of a DAG is

$$L(PE) + x_0*L(B) + x_1*L(CH)$$

This expression involves six weight coefficients $x_0 \ldots x_5$, which we will treat as a vector $\mathbf{x}$ in $\mathbf{R}^6$.

Our algorithm searches for the value of $\mathbf{x}$ minimizing the execution time of the list of training DAGs. It is a generalization of the Rosenbrock [2] procedure, which was intended for general non-linear programming purposes. In order to assure convergence to the solution and maximize the actual execution speed of the simulator, we had to modify the procedure significantly.

Firstly, our goal function (being the execution time of the set of training DAGs) is nonlinear, but not smooth: it consists of a number of flat "terraces" of various elevations. The original Rosenbrock procedure notes only the improvements of the goal function. Our procedure notes these improvements too, but if found itself on a vast "terrace", it will continue searching (for limited number of iterations) hoping to "fall off" onto a lower terrace. The procedure stops after a prescribed number of consecutive unsuccessful iterations (i.e. iterations failing to obtain improvement in the goal function).

Secondly, every coordinate of the vector $\mathbf{x}$ has its own flag which can have one of the values (OFF, NUMB, DONE, ON). Not all coordinates of the vector $\mathbf{x}$ are relevant for a particular machine. For example, the coordinates pertaining to the bus when it is inactive are irrelevant. Similarly irrelevant are those pertaining to channels in a machine without channels. We set the flags of these coordinates to OFF and the optimization routine will not search for optimal values for them.

Furthermore, the optimization routine may find some more coordinates of $\mathbf{x}$ to be irrelevant for particular set of DAGs. For example, should queues not form on channels of the fully connected machine, then the value of x_5 is irrelevant. If no improvement to the goal function is found within a prescribed number of iterations from the start of the search, the routine sets the $flag_5$ = NUMB and stops searching for better values of x_5.

Lastly, the routine may find some improvement for particular values of coefficient x_i, but may subsequently fail to find further improvements for a prescribed number of iterations. The routine will then set $flag_i$ = DONE.

The routine searches only for values of coefficients x_i for which $flag_i$ = ON.

Rosenbrock has recommended that his optimization routine performs periodic rotations of the system of coordinates, perhaps using the Gram-Schmidt method or Powell mathod [3]. We abandoned rotations, having noticed that, although they work, in our case they slow down the process of convergence to a solution.

The pseudocode (Ada style) of our routine is shown in the listing at the end of this paper.

6. Simulation Results

After approx. 20 to 30 iterations the optimization routine stops. The speedups obtained are shown in Fig.5 below.

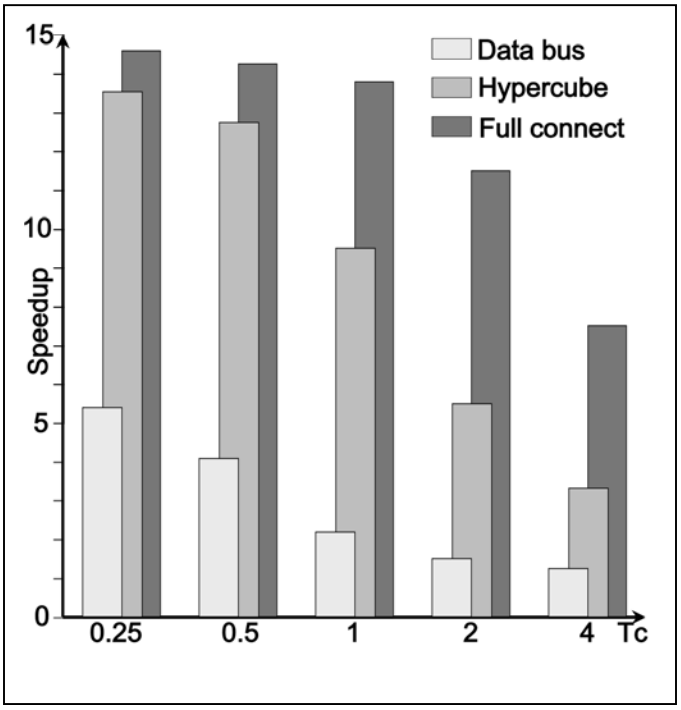

Fig.5. Speedup results as a function of mean comms. time

Observe that for the comms-bound DAGs the data bus machine performs the worst, and its speedup is nowhere near the theoretical maximum of 16 and decreases rapidly with the increase of the mean transfer time T_c.

Perhaps it is not that surprising that the fully connected machine performs best (close to the theoretical speedup limit of 16) until the values of T_c significantly exceed T_x.

We find most encouraging the performance of the 4D hypercube. It shows that a machine does not have to be fully connected to perform well under a comms-bound load. We knew this intuitively: a human brain contains approx. 100 billion neurons. If it were fully connected, it would be too heavy to carry it around.

It is likely that in future we will be able to build adaptive supercomputers capable of exploiting fine-grained parallelism. Although their training process will be computationally expensive, we could make these machines extract suitable **x** values from tables. Missing **x** values for new workload profiles could be computed and then shared across the network.

Acknowledgement

The authors would like to thank Mr. Colin Attlesey for his collaboration in the development of **SKULD**.

References

[1] J.M. Bower, L.M. Parsons: Rethinking the lesser brain, *Scientific American*, Aug 2003, p.53.

[2] Rosenbrock, H.H.: An automatic method for finding the greatest value of a function, *The Computer Journal, 3*, 1960, p.175.

[3] Powell, M.J.D.: On the calculation of orthogonal vectors, *The Computer Journal*, *7,* 1965, p. 303.

[4] Wojcik, W.: *The Structure of the Brain and of the Adaptive Computer Architectures,* www.cosc.brocku.ca/~vwojcik/dreams.htm, 1998.

[5] S. G. Akl, M. Cosnard, and A. G. Ferreira, Data-movement-intensive problems: two folk theorems in parallel computation revisited, *Theoretical Computer Science 95* (1992) 323-337

[6] G. S. Almasi and A. Gottlieb, Highly Parallel Computing, Benjamin /Cummings, 1989


```
procedure MINIMIZE (X0     : in     VECTOR;  -- start point
                    G0     :    out SCALAR;  -- value of GOAL function at X0
                    X      :    out VECTOR;  -- solution point (minimum of GOAL)
                    G      :    out SCALAR;  -- value of GOAL function at X
                    VERSOR : in out MATRIX;  -- basis of orthogonal versors
                    STEP   : in out VECTOR;  -- directional step lengths
                    ITERS  : in out NATURAL; -- number of requested / done iter's
                    MASK   : in out FLAG_MASK;
                    ALPHA  : in     SCALAR;  -- successful step factor
                    BETA   : in     SCALAR)  -- unsuccessful step factor (negative)
is
    ITER : NATURAL := 0; -- iteration counter
    BADS : NATURAL := 0; -- count of consecutive unsuccessful directional attempts
    BAAD : NATURAL := 0; -- count of consecutive unsuccessful iterations
    BAADMAX : constant NATURAL := 5; -- consecutive unsuccessful iteration limit
    NEWX    : VECTOR;
    NEWG    : SCALAR;
    NUMOFFS : NATURAL := COUNT(VALUES => OFF, WITHIN => MASK);

    type COUNTS is array (INDEX) of NATURAL;
    FAIL : COUNTS  := (others => 0);
--------------------------------------------------------------------------------
begin
    G0 := GOAL(X0); X := X0; G := G0

    while (ITER < ITERS) and then (BAAD < BAADMAX) and then (NUMOFFS < DIM)
    loop
        ITER := ITER+1;  BADS := 0;
        for DIR in INDEX'RANGE
        loop
            if MASK(DIR)=ON then
                NEWX := X + (STEP(DIR)*COL(DIR, FROM => VERSOR));
                NEWG := GOAL(NEWX);
-- mask admin:
                if NEWG < G then
                    FAIL(DIR) := 0;
                else
                    FAIL(DIR)   := FAIL(DIR)+1;
                    if FAIL(DIR)=2*BAADMAX then
                        if FAIL(DIR)=ITER then MASK(DIR) := NUMB;
                                          else MASK(DIR) := DONE; end if;
                        NUMOFFS := NUMOFFS+1;
                    end if;
                end if;
-- optimization:
                if (NEWG <= G) then
                    X := NEWX; G := NEWG; STEP(DIR) := ALPHA*STEP(DIR); BADS := 0;
                else
                    STEP(DIR) := BETA*STEP(DIR); BADS := BADS+1;
                end if;
            end if;
        end loop;
        if BADS>=DIM-NUMOFFS then BAAD := BAAD+1; else BAAD := 0; end if;
    end loop;
    ITERS := ITER;
end MINIMIZE;
```