Paper I

A Parallel Interval Method Implementation for Global Optimization using Dynamic Load Balancing

Jerry Eriksson and Per Lindström
Department of Computing Science, Umeå University
S-901 87 Umeå, Sweden.
E-mail: Jerry.Eriksson@cs.umu.se, Per.Lindstrom@cs.umu.se

Abstract

During the past few years the interest paid to global opt imization has rapidly increased. One of the main reasons is the new technology of parallel computers which offer computational power capable of solving global optimization problems in reasonable time. The method studied in this work is based on interval analysis which provides a reliable way for solving the problem. Despite the fact that the method contains a high degree of potential parallelism, it is not straight forward to parallelize due to its irregular and unpredictable computational behaviour. This paper deals with the problem of balancing the load dynamically, both with respect to the quantity and to the quality of the tasks. Efficient strategies are proposed and implemented on an Intel iPSC/2 hypercube. Since the sequential algorithm is used as a base it will be modified to suit the parallel algorithm.

Key words. Parallel computing, Global optimization, Interval computations, Dynamic load balancing.

1 Introduction

During the last two decades, a number of methods for solving global optimization problems has been proposed. One of the hardest issues for these problems is the computational requirements that are very high. Fortunately, modern computer technology has made significant progress which has brought the global optimization problem into new light. The global optimization problem is,

$$\min f(x)$$

$$s.t. \quad l_i \le x_i \le u_i \ ; \ i = 1, 2, ..., n.$$

In this paper, let I be the set of real compact intervals. A box is defined to be a closed rectangular parallelepiped with sides parallel to the coordinate axes. In this paper, a box is denoted with an underlined letter as in $\underline{X}$ where $\underline{X} \in I^n$.

The method studied here for solving the global optimization problem is named *the interval method*, see [8], [5], [1], and [11]. The method provides a reliable way for solving the global optimization problem. It is based on an exhaustive search in a given box. The box is dynamically sub-divided into smaller sub-boxes until they are manageable by different elimination phases. Throughout the algorithm, all sub-boxes not yet examined are ordered in a priority queue so that the most promising sub-box can be quickly extracted.

Here the interval method is implemented as a *branch-and-bound algorithm* which is an important class of methods for solving optimization problems. Since the parallelism in *branch-and-bound algorithms* is inherently irregular, it requires dynamic load balancing in its implementation on a distributed memory computer. In this study, different decentralized dynamic load balancing strategies are designed and implemented. The load balancers are based on both receiver and sender initiated approaches. In many implementations in this area, the scheduler is only dealing with balancing the load with respect to the quantity of tasks. In this paper, it is seen that it is also important to consider the quality of the tasks.

Initially, in the parallel execution, the box is split into p sub-boxes, where p is the number of processors. Since the original sequential algorithm does not include any initial sub-division, it is not fair to compare execution time between the sequential algorithm and the parallel algorithm. Instead, it is better to include an initial phase in the sequential algorithm. Additionally, in situations when the Krawczyk method (the local minimum search method) performs poorly a bisection is often too weak. A sub-dividing strategy based on the result of the Krawczyk method that may split the box into more than two sub-boxes is proposed.

The paper is outlined as follows. In Section 2, the interval method is described. If the interval method is unfamiliar, a sufficient introduction to interval analysis and a more detailed description of the interval method is found in [10].

The modifications to the sequential algorithm are given in Section 3. The parallel algorithm and its realization are discussed in Section 4. Section 5 deals with the dynamic load balancing problem. Last, in Section 6, the results and experiences are summarized.

2 The sequential algorithm

The algorithm is based on an exhaustive search in the multidimensional solution space. The initial box can arbitrarily be sub-divided into sub-boxes which can be investigated independently of each other. If a sub-box is too large to be manageable it is split further into smaller sub-boxes. Fortunately, it is not necessary to fully investigate all sub-boxes. Instead, many sub-boxes can quickly be discarded as uninteresting by different rules. Further a priority queue is used to order the sub-boxes by increasing lower bound. The lower bound is the lowest function value the objective function can obtain in a sub-box. During the execution, sub-boxes are dynamically queued and dequeued from the priority queue.

The first step in the algorithm is to compute the lower bound of the initial box, $\underline{X_0}$. The box $\underline{X_0}$ with its corresponding lower bound is inserted in the priority queue. As long as there exist sub-boxes that are not investigated, i.e., the priority queue is not empty, the execution continues.

Next, the first sub-box is dequeued from the priority queue. If its corresponding lower bound is higher than the current minimum (*cmin*)*, the algorithm terminates. Hence, it is possible to eliminate sub-boxes before any costly computations of the sub-boxes are carried out. This test is often called the mid-point test. In the second step it is investigated whether the sub-box is monotonous †. The purpose of this elimination phase, which is often called an acceleration device, is to make the algorithm faster and it does not seek a minimum explicitly. The algorithm would still be correct even if this phase was removed from the algorithm, but the time to solve some problems could be expected to increase dramatically.

If the sub-box passes these tests the Krawczyk method is invoked to explicitly seek a local minimum. The Krawczyk method can end up with one of three different results. If the Krawczyk method converges it has either made a significant or a non-significant improvement, otherwise the Krawczyk method diverges and the box is excluded. The Krawczyk iteration continues as long as it converges significantly. In the non-significant case the box is split into two new sub-boxes which are inserted in the priority queue. If a lower minimum is found, i.e., a value which is lower than *cmin*, then *cmin* is updated.

When all sub-boxes, with a width greater than a given epsilon, ϵ, have been investigated, the algorithm terminates. At this stage *cmin* holds the global

* *cmin* is initially set to $+\infty$.

†A box $\underline{X}$ is said to be monotonous if the function f is monotonous in $\underline{X}$.

minimum.

3 Modifications to the original algorithm

The purpose of this modification is to extend the sequential algorithm so it can serve as an efficient basis for the parallel algorithm. The original sequential algorithm does not contain an initial sub-dividing of the solution space. Since the eliminations phases are often inefficient in the beginning of the execution an initial phase is to prefer. The second sub-diving modification proposed uses the result of the Krawczyk operator.

The initial phase. We propose an extension so a number of bisections can be carried out initially. Here the term initial phase refers to the time before entering the sequence of cycles. The elimination phases, especially the Krawczyk method, are often inefficient in the first part of the execution and it is not motivated to invoke them too early. Ideally, it would be more efficient if the sub-boxes were small enough to be directly manageable by the elimination phases.

There is a trade-off in determining the number of sub-boxes to be generated. If too few sub-boxes are generated, the gain will not be large. On the other hand, if too many sub-boxes are generated, a lot of sub-boxes will be unnecessarily investigated. In order to keep the algorithm reliable, the generation of sub-boxes should be held at a moderate level.

Unfortunately, it is impossible to give an optimal formula on how a sub-box generation should be defined. Probably, the only practical way is to use common sense and experiences from computer tests. In order to design a useful practical algorithm, the following three parameters have been emphasized:

- the width of the initial box, $w(\underline{X}_0)$,
- the required precision, ϵ,
- the dimension of $\underline{X}_0$, n.

Here the quotient $w(\underline{X}_0)\ /\ \epsilon$ is an important factor since it is the upper bound of how many sub-boxes that must be investigated. However, this quotient could be a very large number. By applying log_{10} to the quotient a suitable number is often achieved. That is, it gives values not larger than 10 for reasonable sizes of the initial box and of the accuracy respectively. This leads to Formula (3.1).

$$s = \log_{10}(w(\underline{X}_0)/\epsilon) \tag{3.1}$$

However, when n is large s should be limited. That is, if $n \leq 10$, then s is be limited to 2, to avoid a too high sub-box generation frequency.

The next task is to decide how s should be used. If n is large, it is generally more complicated and should lead to a higher generation of sub-boxes. After some testing, the following strategy turned out to be the most promising.

Algorithm: INITIAL PHASE

1. Compute s by Formula (3.1).
2. For $i = 1, n$ do steps 3-5
3. Pick all sub-boxes in the priority queue.
4. Split all these sub-boxes into s sub-boxes for dimension i.
5. Insert all these in the priority queue.

In other words, the initial box is dequeued from the priority queue and is further split into s sub-boxes. These newly created sub-boxes are inserted in the priority queue. At this point the priority queue consists of s sub-boxes. This pattern is repeated for all dimensions, n. After n repetitions the priority queue consists of s^n sub-boxes.

There is no need to compute the lower bound of the sub-boxes generated except in the last loop. That is, in the first $n-1$ loops the value zero is assigned to the lower bound of each sub-box. Since all lower bounds are zero, the priority queue is essentially a simple list in these loops. Hence, this scheme avoids the calculation of s^{n-1} lower bounds. But more importantly, the elimination phases have not been invoked for these sub-boxes.

It could also be motivated to consider different mathematical operations and interval dependency of the object function. However, it would probably be an immense task to determine or calculate weights on these operations and it has not been dealt with.

A modified bisection The Krawczyk method [6], computes an operator $K(\underline{X})$, called the Krawczyk operator. In each iteration the sub-box $\underline{X}_{n+1} = \underline{X}_n \cap K(\underline{X}_n)$ is computed. If $\underline{X}_{n+1}$ is not an empty sub-box, there might exist a local minimum in the sub-box, otherwise the method diverges. A positive side effect lies in the inherent information in $K(\underline{X})$. If $w(K(\underline{X}_n)) \geq 0.9w(\underline{X}_n)$, then the Krawczyk method has made a non-significant improvement (otherwise significant improvement). The reason is probably that $w(\underline{X}_n)$ is too large to be efficiently manageable by the Krawczyk method. In these situations a bisection of $\underline{X}_n$ is required. If $w(K(\underline{X}_n)) \gg w(\underline{X}_n)$, there is a risk that a bisection is too powerless. If the sub-box could be split into more than two sub-boxes the elimination phases would be invoked on smaller sub-boxes.

In order to realize this approach another practical algorithm is designed. The following parameters are considered as important:

- the width of the current sub-box, $w(\underline{X}_n)$,

- the width of the Krawczyk operator, $w(K(\underline{X}_n))$. If $w(K(\underline{X}_n))$ is much wider than $w(\underline{X}_n)$ it could be expected the Krawczyk method has been very unsuccessful.

Basically, this idea is somewhat akin to the previous algorithm. In this case the quotient $w(K(\underline{X}_n))$ / $w(\underline{X}_n)$ is of great importance. That is, a high quotient should lead to a high number of sub-boxes generated. The quotient is applied to log_{10} in order to obtain a suitable number. Further, the upper bound of this number is limited to $uplimit = 6$ leading to Formula (3.2).

$$s = \max(\min(\log_{10}(w(K(\underline{X}_n))/w(\underline{X}_n)), uplimit), 2) \tag{3.2}$$

The sub-division is applied to the widest interval in the sub-box.

However, this solution is not fully satisfactory. If the Krawczyk method continuously performs poorly, a very high number of sub-boxes will be generated. Therefore, an adaptive mechanism is included. If the Krawczyk method performs poorly the number of sub-boxes to be generated is slightly decreased by a term *ad*. The term *ad*, initially equals zero, could also be increased during the execution if it turns out that the Krawczyk method is getting more efficient. In our tests, it has been shown that it is suitable to let *ad* be decreased by 0.25 for each non-significant improvement and increased by 0.25 for each significant improvement. By using Formula (3.3) the variable, s, varies between two (ordinary bisection) and *uplimit*(equal to six in our tests). Hence, if *ad* is getting too large or too small it will not be not adjusted.

$$s = \max(\min(ad + log_{10}(w(K(\underline{X}_n))/w(\underline{X}_n)), uplimit), 2) \tag{3.3}$$

Summary, these modifications consist of the following two parts; First, the algorithm INITIAL PHASE is supplied, which purpose is to avoid unnecessary computations in the beginning of the execution. Second, in the situations when the width of the Krawczyk operator, $w(K(\underline{X}_n))$, is much larger than the width of the computed sub-box, $w(\underline{X}_n)$, Formula (3.3) is used to determine the number of sub-boxes to be generated.

These modifications are solely based on common sense, but they have an empirical basis. It is most important to keep control over the number of generated boxes otherwise an extremely high sub-box generation might result.

Test results. The test functions selected are well-known global optimization problems. There are polynomial functions as well as functions which include both exponential operations or trigonometrical operations.

The selection of test functions is shown in Table 1. Six of the seven functions can be found in [13] and number 6 can be found in [12]. The corresponding initial boxes for the functions are given in Table 2. Throughout all tests the accuracy (ϵ) is set to 0.1. The computer used for the sequential implementation is a Sun Sparc 1+.

No.	Function
1	$\sum_{i=1}^{5} i\cos((i-1)x_1+1)\sum_{j=1}^{5} j\cos((j-1)x_2+1)$
2	$(1.5-x_1+x_1x_2)^2+(2.25-x_1+x_1x_2^2)^2+(2.625-x_1+x_1x_2^3)^2$
3	$\sum_{k=1}^{10} F_k^2, F_k = \exp(\frac{-kx_1}{10}) - \exp(\frac{-kx_2}{10})A_kx_3,\ A_k = \exp(\frac{-k}{10}) - \exp(-k)$
4	$(x_1+10x_2)^2+5(x_3-x_4)^2+(x_2-2x_3)^4+10(x_1-x_4)^4$
5	$100(x_2-x_1^2)^2+(x_1-1)^2$
6	$\sum_{i=1}^{10} \ln{(x_i-2)}^2+\sum_{i=1}^{10} \ln{(10-x_i)}^2-\prod_{i=1}^{10} x_i{}^2$
7	$\frac{(x^2+y^2)}{200}+1-\cos x\cos\frac{y}{\sqrt{2}}$

Table 1: The test problems.

No.	Initial box, $\underline{X}^0$	# local minima
1	[-10, 10][-10, 10]	760
2	[-10, 10][-10, 10]	1
3	[0, 5][8, 11][.5, 3]	2
4	[-4, 5][-4, 5][-4, 5][-4, 5]	1
5	[-10000, 10000][-10000, 10000]	1
6	$[3,4][3,4][3,4][3,4][3,4][3,6][3,6][3,6][3,6][3,6]$	Boundary value
7	$[-1000,1000][-1000,1000]$	1

Table 2: The initial boxes.

Function No.	Original algorithm	After modification	Improvement
1	5.81	4.26	1.55 (27 %)
2	1.56	1.46	0.10 (6 %)
3	12.45	8.46	3.99 (32 %)
4	0.36	0.19	0.17 (47 %)
5	0.82	0.58	0.24 (29 %)
6	551.48	388.61	162.87 (29 %)
7	2.38	1.46	0.92 (38 %)

Table 3: Comparison of the original sequential algorithm versus the modification (in seconds).

Table 3 summarizes the total execution times. The second column shows the improvements after the modification. The last column displays the improvement of the original algorithm versus the final proposed algorithm. For six of the functions an improvement of $27 - 47\%$ has been made. For Function 2 only a minor improvement is achieved. For a more detailed presentation of the test results, see [3].

We stress that the purpose of these modifications is to make the algorithm more suitable to the parallel algorithm. However, the test results show that these ideas might be successful in the sequential algorithm as well.

4 The parallel algorithm

The sequential algorithm has a high level of inherent parallelism. All sub-boxes are independent of each other and can be investigated in parallel. As mentioned earlier, the interval method is here formed as a *branch-and-bound algorithm*. The following algorithm is adapted for the interval method, but the structure also fits many other *branch-and-bound algorithms*, see [4].

1. **Initialization.** The original box is inserted in the priority queue and the *cmin* is set to ∞.

2. **Selection.** The most promising sub-box is selected and dequeued from the priority queue.

3. **Lower bound test.** If the lower bound of the new sub-box is higher than *cmin*, the sub-box is excluded from further consideration (The mid-point te st).

4. **Elimination test.** The sub-box is investigated by the different elimination phases. The sub-box is excluded if it is guaranteed not to lead to an optimal soluti on.

5. **Feasibility test.** Update *cmin* if a lower minimum has been found. If so, exclude all sub-boxes in the priority queue that have a higher lower bound than *cmin*.

6. **Branching.** Split the sub-box into new smaller sub-boxes and insert these into the priority queue.

7. **Algorithm termination.** If the priority queue is not empty then repeat 2-6, otherwise the algorithm terminates and the optimal value is hold by *cmin*.

4.1 Realization of branch-and-bound methods on iPSC/2

The parallel computer used in this work is an Intel iPSC/2 hypercube, supplied with 64 processors (nodes). The hypercube computer belongs to the category of distributed memory computers. Another name is message-passing computers since communication between nodes are carried out by sending messages to each other. A hypercube has always 2^d nodes where each node has d physical neighbours. The most interesting properties of the hypercube topology is the different mapping configuration possibilities, such as ring, torus, cubes, and trees, see [9]. Another nice feature is the efficient implementation of global operations. For example, if each node has a partly computed vector, it is possible to summarize all these in d steps.

In order to implement a parallel *branch-and-bound method*, or more precisely the parallel interval method on a distributed memory machine, several issues must be solved.

- How and where should the priority queue be stored in order to:
  - keep all nodes busy investigating sub-boxes (quantity),
  - investigate the most promising sub-boxes (quality),
  - utilize the total memory efficiently.
- How to broadcast the new minima as quickly as possible?

The first item refers to the dynamic load balancing problem. Our approach uses two processes on each node; a WORKER and a SCHEDULER . The first process, the WORKER, investigates sub-boxes. The second process, the SCHEDULER, tries to balance the load as fairly as possible among the nodes. It also deals with the distributed termination problem. The idea of using two processes is a matter of abstraction. To treat the two processes as separate makes the discussion conceptually cleaner and also serve to emphasize the independence of the processes. The WORKER should be free from all responsibility considering load balancing and, besides investigating sub-boxes, only request and receive sub-boxes from the SCHEDULER . In the next section, different load balancers are proposed.

The second item refers to the problem of broadcasting a new minimum. If a new lower local minimum is found the WORKER broadcasts the minimum to all other WORKERS. This is carried out relatively cheaply by just one communication call, *csend*(). All other nodes can receive this message by a corresponding *crecv*(). However, the problem is to determine when, during the execution, the WORKER should receive this message. It is expensive to probe for a message often. On the other hand, unnecessary computations might be carried out if the node probes too seldom.

Alternatively, the iPSC/2 offers an interrupt-driven message primitive, *hrecv*(), that solves this problem efficiently. In the beginning of the execution, a node can inform the operating system which messages (in our case a new minimum) that should have special treatment. Whenever a new minimum is pending it is immediately received and the execution continues in an user predefined subroutine. In this subroutine *cmin* is updated and then it is legal to jump to any place in the code. This feature enables the WORKER to receive a new minimum during an investigation of a sub-box. It is even possible that the current sub-box is excluded during the investigation if its lower bound is higher than the new received minimum.

It is certainly preferable to use this primitive instead of *crecv*(). The advantage is twofold. First, one does not have to bother when and where to probe for a new minimum. Second, *cmin* is updated immediately after it reaches the node which makes it possible to avoid some computations.

5 Dynamic load balancing

The amount of work of investigating a sub-box (task in the sequel) can differ widely. Due to the fact that no knowledge considering the behaviour of the execution is known *a priori*, it is not suitable to statically balance the load. The use of a static load balancing strategy would undoubtfully lead to an inefficient use of the parallel computer. In general, a *branch-and-bound algorithm* can never be efficiently implemented on a parallel machine without dynamic load balancing.

In an earlier work, [2], two different approaches were implemented and compared; a centralized load balancer based on the Master-and-slave concept and a hypercube topology based decentralized load balancer.

In the centralized load balancer the priority queue and *cmin* are stored at the Master node. The other nodes (the slaves) request work from the Master. The Master always deliver the most promising task to the requesting node. This is the main advantage of using centralized load balancers. Sadly, a lot of drawbacks occur using this approach. The Master very often becomes a serious bottleneck w.r.t. the processor capacity and worse, the communication unit will often be quickly overloaded.

The decentralized load balancer, on the other hand, avoids the bottlenecks mentioned above. But, since the priority queue is distributed among the p nodes and the load balancer only consider the quantity of the tasks, the p most promising tasks are seldom investigated. Test results showed that, when the number of nodes exceeds 8, the decentralized load balancer is superior to the centralized load balancer. Hence, in the sequel, only the decentralized load balancer will be considered and further developed.

The aim for a dynamic load balancer is to migrate work from the busy nodes to the lightly loaded or idle nodes during the execution. A decentralized dynamic load balancing scheme can be characterized as one of the following, see [7]:

- *Sender-initiated.* This means that the load balancer delivers tasks from the local node to lightly loaded nodes without any explicit request from the remote nodes. However the load balancer often has some knowledge of the load in the system.
- *Receiver-initiated.* In these schemes the load balancer requests tasks from a sub-set of nodes. A demand is initiated by an idle node or lightly loaded node.
- *Hybrid.* This scheme is a hybrid of the two earlier.

In the following sections, receiver-initiated as well as sender-initiated load balancers are proposed. In Section 5.1 a receiver-initiated quantity load balancer is described. This load balancer considers the load with respect to the quantity of the tasks. The goal of this load balancer is to reduce the inefficient parts of our previous load balancer based on the hypercube topology. Two sender-initiated load balancers are proposed in Section 5.2. These load balancers strive to balance the load with respect to the quality of the tasks. The purpose is to capture the advantages of the centralized load balancer. Together, the receiver-initiated and sender-initiated load balancer form a hybrid load balancer. In Section 5.3 test results for an Intel iPSC/2 hypercube are given.

5.1 The receiver-initiated quantity load balancer

One of the drawbacks of using the hypercube topology is that the protocol requires a lot of messages. Consider the following scenario for a d-dimensional hypercube: A node has been idle[‡]. The idle node requests work from all its immediate neighbors (d messages). Upon receiving a task from one or more abun-

[‡]A node that is out of tasks is called idle.

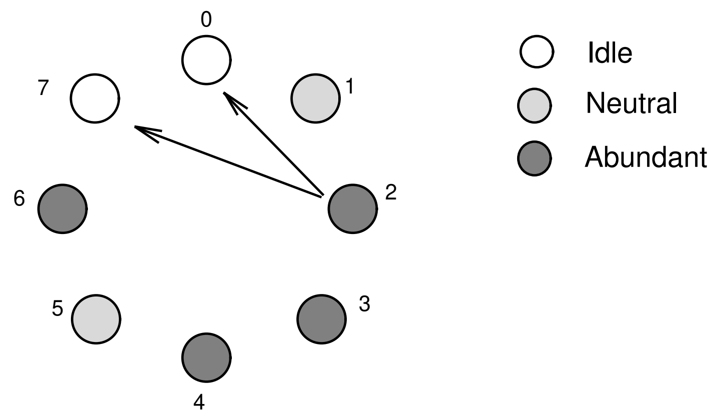

Figure 1: Node 7 has been idle. It sends a request to the next node in the ring. Just before node 0 receive the request from node 7 it has also become idle. Node 0 packs both requests into one message and sends it to node 1. Unfortunately, node 1 has no tasks to deliver so it sends the request further to node 2. Since, this node has tasks, it sends one task directly to both node 0 and node 7 respectively, without interfering node 1.

dant [§] node it sends a acknowledgement to the same set of nodes(d messages). This implies a total of $2d$ messages for the requests and acknowledgements, plus one or more messages consisting of tasks from abundant nodes. Another disadvantage of using this scheme is that the powerful cut-through network is not fully utilized. Here, a new load balancer algorithm based on a ring topology that solves the two earlier drawbacks is proposed (see Figure 5.1). The scheduler proposed has the following properties.

First, it is fully asynchronously and decentralized which means that balancing is carried out whenever necessary and not at specific intervals. The term decentralized means that the responsibility for load balancing is decentralized among all nodes. Second, it reduces the communication for the SCHEDULER to a very low level. An idle node sends a request to the next node in the ring and upon receiving a task it sends **no** acknowledgement. For a 64-nodes configuration, this approach reduces the number of messages significantly. Third, if an idle node sends a request to another idle node, the request is passed further packed into one message. The last feature is that an abundant node does not send the task back to the ring. Instead it sends the task directly to the target node thereby utilizing the cut-through communication network. For a more detailed description, see [3].

[§] A node that has a surplus of tasks is called abundant

5.2 The sender-initiated qualitative load balancer

To this point the quality of the tasks has not been considered. By the term quality, we mean how promising the tasks are (the lower bound of the sub-boxes). The purpose of the quality balancer is both to speed-up the execution and to reduce the number of tasks, thereby capturing the advantages of the centralized load balancer. In other words, its purpose is to overcome the problem of investigating non-promising tasks. Several approaches based on heuristics have been tested. The two schemes presented below turned out to give the best results. The receiver-initiated quantity load balancer is included as a base in both schemes.

5.2.1 Random distribution

The first scheme acts as follows. Each SCHEDULER counts the number of tasks generated by splittings. Each time the counter exceeds a *generator limit* the SCHEDULER sends its first task in its priority queue to a randomly chosen node. In this scheme the *generator limit* is fixed to 5. This strategy tends to keep the most promising tasks scattered among all nodes so all WORKERS are likely to investigate one of the most promising task. Each separate local SCHEDULER has no knowledge about the load on other nodes. So, in the worst case, a node with a lot of promising tasks can unnecessarily receive non-promising tasks. This is not too bad since the non-promising tasks will be inserted at the end of the receiver's priority queue.

Notice that, assuming that the random generator works well (which it does), this scheme also implicitly balances the load with respect to quantity.

5.2.2 Adaptive random distribution

This scheme is an improvement of the previous. Instead of having the *generator limit* fixed it can be varied during the execution. The scheme acts as follows. As previously, a SCHEDULER that receives a task inserts it in its priority queue. If the task is inserted **first** in the priority queue, the SCHEDULER respond a *good* message back to the sending node. Otherwise, if the task is inserted in another place in the priority queue, a *bad* message is responded. The sending node then receives the responded message and its local *generator limit* is adjusted depending on the value of the message. If a *good* message is received, the *generator limit* is decreased by one. This means that the SCHEDULER node will distributes tasks more frequently. The opposite holds if a *bad* message is received. For example, let the *generator limit* initially be set to 5. If a SCHEDULER has distributed 15 *bad* and 3 *good* tasks, the *generator limit* is 17 (5 + 15 - 3). In this scenario the SCHEDULER is distributing tasks rather seldom due to all *bad* messages it has received.

The advantages of using this approach are that a SCHEDULER with promising tasks tend to distribute more tasks than others. This leads to less distributions and investigations of non-promising tasks.

5.3 Test results and comparisons

In the context of parallel computers, the efficiency is measured as:

$$E(p) = \frac{time(s)}{time(p) * p},$$

where $time(s)$ is the sequential execution time and $time(p)$ is the parallel execution time using p nodes. $E(p) * 100$ expresses the efficiency in percentages. . Naturally, the goal is to achieve efficiency as close as possible to 100%.

The test results show comparisons of the different load balancers w.r.t. the efficiency. Due to some hardware problems, we have been forced to use software interval rounding. By that fact, no execution times are presented. The three most time consuming (on the iPSC/2) functions from the seven test functions presented in Table 1 have been selected.

Figure 2 shows the results for Function 3. The efficiency is over 60 % up to 16 nodes. Using more nodes, it is hard to achieve high efficiency due to the very short execution time. It is clear that a problem must be large if it should be fruitful to use a parallel computer.

In Figure 3 the results for Function 4 are displayed. The efficiency is over 80% for all load balancers using up to 32 nodes. It is clear that the hybrid load balancer including the adaptive random distribution is more efficient than the others for all node configurations. Due to the very irregular parallel algorithm it is possible to achieve an efficiency over 100%. This is due to the fact that the parallel algorithm does not investigate the sub-boxes in the same order as the sequential algorithm.

The best results are achieved using Function 6 as displayed in Figure 4. The differences between the different load balancers are obvious. The use of adaptive random distribution is very successful for large problems. These results confirm the advantages of using qualitative properties in the load balancer.

Last, the number of tasks generated for Function 6 are presented. Figure 5 shows that if only a quantitative load balancer is used, the number of tasks can increase dramatically. In this example the difference is around 19000 tasks (35000-16000) using 64 nodes.

6 Conclusions

In this paper we have developed different dynamic load balancers for implementing a global optimization method on an Intel iPSC/2 hypercube. Decentralized load balancers has several advantages; they use the memory efficiently, the communication unit will not be overloaded. Another advantage is scalability. These

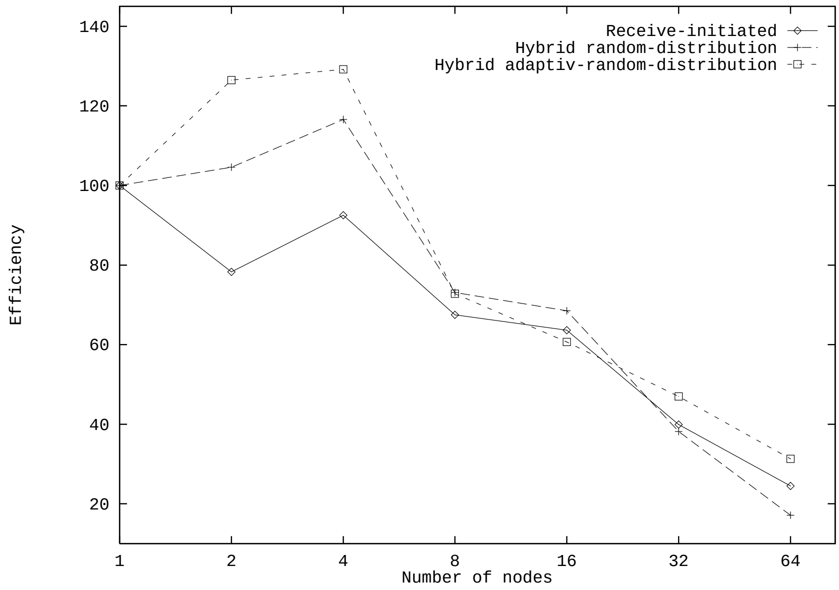

Figure 2: Comparison of the efficiency using Function 3.

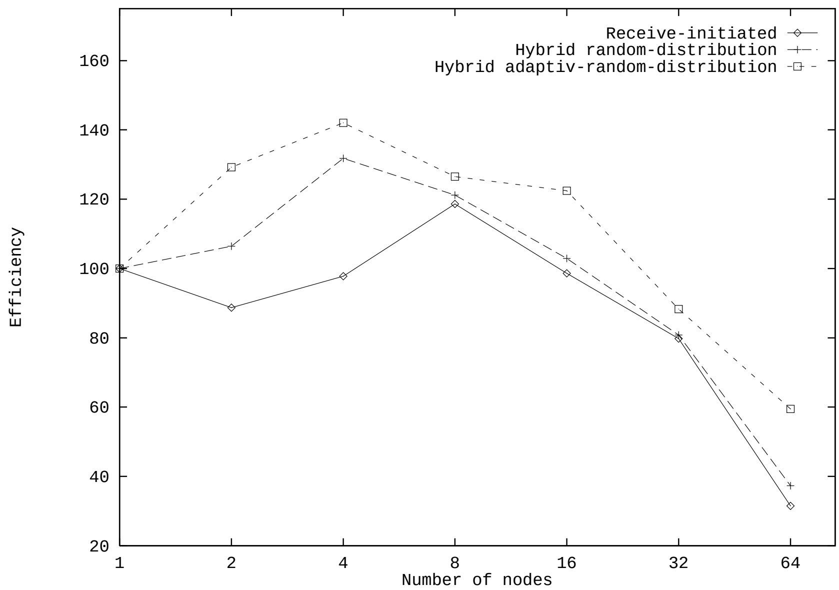

Figure 3: Comparison of the efficiency using Function 4.

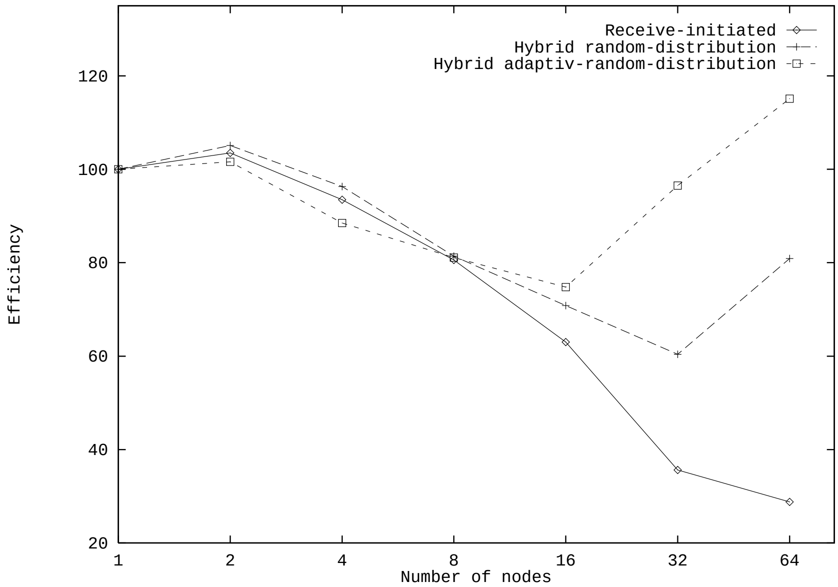

Figure 4: Comparison of the efficiency using Function 6.

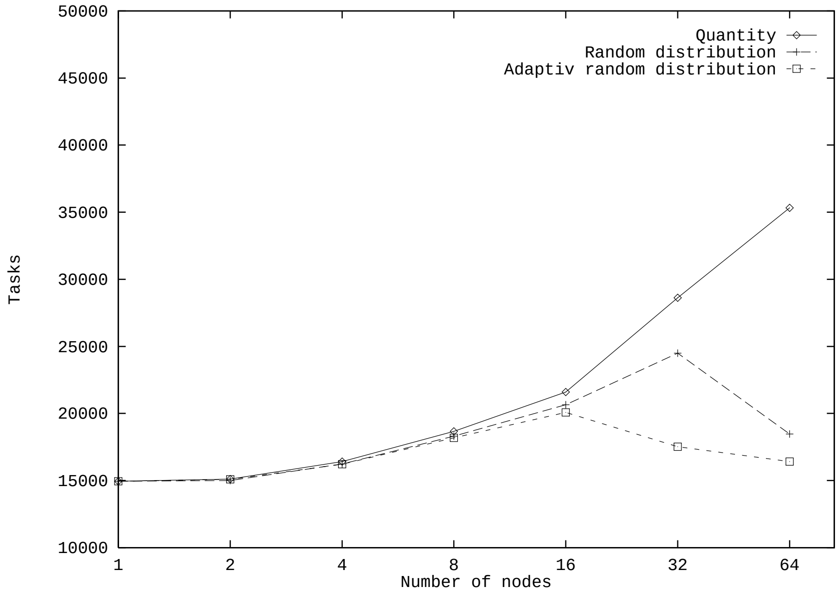

Figure 5: Comparison of the number of sub-boxes using Function 6.

implementations will execute efficiently on larger hypercubes, if the problem is sufficient large, without being altering.

It has been shown that it is appropriate to use a *hybrid* scheduler. A *receiver-initiated* quantity load balancer enables the nodes to request tasks from other nodes when idle. It is also, for *branch-and-bound algorithms* useful to include a *sender-initiated* quality load balancer which delivers promising tasks to other nodes. Two approaches have been examined. First, a scheme based on random distribution where the load balancer distributes its most promising task to randomly chosen nodes at specific intervals. The second variant acts similarly, but the tasks are distributed at intervals which vary during the execution depending on the quality of the distributed tasks. The test results show that the execution time as well as the number of tasks decrease when the scheduler becomes more sophisticated.

References

[1] O. Caprani and K. Madsen. Experiments with interval methods for non-linear systems. Technical report, Institute fur Angewandte Mathematik, Universitæt Frieburg i. Br. Copenhagen, July 1981.

[2] J. Eriksson. Parallel global optimization using interval analysis on ipsc/2 (draft). Technical report, UMINF-report, Information Processing, University of Umeå , 1990.

[3] J. Eriksson. Parallel global optimzation using interval analysis. Technical report, UMINF-report, Information Processing, University of Umeå , 1991.

[4] E. W. Felten. Best-first branch-and-bound on a hypercube. In *Conference on Hypercube Concurrent Computers and Applications*, volume 1, pages 1500–1504. ACM, 1988.

[5] E. Hansen. Global optimization using interval analysis - the multi-dimensional case. *Numerische Mathematik*, 34:247–270, 1980.

[6] R. Krawczyk. Newton-algorithmen zur bestimmung von nullstellen mit fehlerschranken. *Computing*, 4:187–201, 1969.

[7] F. C. Lin and R.M. Keller. Gradient model: A demand-driven load balancing scheme. In *IEEE Conf. on Distributed Systems*, pages 337–357, 1984.

[8] R.E. Moore. *Interval analysis*. Prentice Hall, Englewood Cliffs, 1966.

[9] S. Ranka, Y. Won, and S. Sahni. Programming a hypercube multicomputer. *IEEE Software*, pages 69–77, 1988.

[10] H. Ratchek and R.L. Voller. What can interval analysis do for global optimization. *J. of Global Optimization*, 1(2):111–130, 1991.

[11] H. Ratschek and J. Rokne. *New Computer Methods for Global Optimization.* Ellis Horwood, Chichester, 1988.

[12] J. Thoft-Christensen. Global optimering på paralleldatamat. Master's thesis, Numerical Institute in Copenhagen, 1989.

[13] G. W. Walster, E. Hansen, and S. Sengupta. Test results for a global optimization algorithm. In Boggs, Byrd, and Schnabel, editors, *Numerical optimization*, pages 272–287. SIAM J. on Scientific and Statistical Computing, 1984.