# Comparison of Dynamic Load Balancing Strategies

Herbert Kuchen and Andreas Wagener
Lehrstuhl für Informatik II
RWTH Aachen*

**Abstract**

Parallel implementations of functional programming languages are often based on graph reduction. Tasks represented by task nodes in the graph are generated during run time. These tasks have to be distributed among the processors dynamically. For a prototype implementation [LK89] of such a language on a loosely coupled multiprocessor system consisting of transputers [In88], we have tested several load balancing strategies.

## 1 Introduction

In functional programs, the subexpressions of an expression can be computed in parallel (if they are needed), since there are no side effects like modifications of global variables. This implicit parallelism can be exploited by implementations on parallel computers. One generally accepted strategy (see for example [Jo84][Bu88]) to implement functional programming languages is to use graph reduction [Wa71]. [LK89] describes a graph reduction based implementation of a functional language on a loosely coupled multiprocessor system, i.e. a system, where all processors have their own local memory and communicate by exchanging messages. Each processor is physically connected to a small number of other processors (its immediate neighbors), but logically a full connection of all processors is established via routing. All processors are of the same kind.

The experiments mentioned in the following are based on this implementation on a system of transputers. The computation graph is distributed among the different local stores. It contains among others so called task nodes representing such function applications, which are worthwhile to be computed on another processor. During reduction, a task node is replaced by a graph which again may contain task nodes. In principle, every processor is able to execute each task. If a processor produces more tasks than it needs, some of them may be sent to neighbor processors for evaluation. Of course, this has to be done dynamically during run time.

Hence, the problem arises to find good strategies for the distribution of tasks in this framework. In the literature, several schemes have been proposed (see for example [HG84] [LK87] [BS81] [NX85] [Hw82] [Sm80]). Since we are mainly interested in

*Ahornstr. 55, D-5100 Aachen, West Germany

large multiprocessor systems, we only consider decentral schemes (without a global task manager) to avoid bottlenecks. There are several possibilities to classify the different load balancing strategies. We distinguish passive strategies, where idle processors ask their neighbors for work, active strategies, where the distribution of tasks is initiated by their creating processor, and mixtures of both. A good distribution algorithm should ensure that all processors have enough tasks nearly all the time, and hence allow a good speedup[†]. Therefore, they should enable a fast (re)activation of idle processors.

Section 2 mentions related work. The tested load balancing strategies are presented in section 3. Section 4 contains experimental results.

# 2 Related Work

In the literature, several load balancing strategies have been proposed. Some of them [HG84] [BS81] [LK87] are used in the context of a parallel implementation of an applicative language, while others are more general.

In [BS81], a simple passive scheme is used, where idle processors may steal a task from one of their immediate neighbors. The choice of this neighbor is based on informations on the load of the neighbors. A stolen task must not be stolen again.

A more sophisticated scheme, the *gradient model* is presented in [LK87]. Each processor has one of the states light, moderate, and heavy depending on whether it has too few tasks, a reasonable number of tasks, or too much tasks. A light processor has *pressure* 0, while the pressure of a moderate or heavy processor is computed by adding one to the minimal pressure of its neighbors. Each processor informs all its neighbors, when its pressure changes. A heavy processor may send one of its tasks to the neighbor with the lowest pressure. This strategy allows a task to move many steps in the network, including cycling. The gradient model is hard to classify in terms of active and passive strategies.

Hudak and Goldberg [HG84] suggest an active scheme, called *diffusion scheduling*, where the creating processor of a task decides to keep it or send it to an immediate neighbor. This decision is based on the load of the involved processors and, for the sophisticated version, on the distances of the task's arguments from each of these processors. The idea is to push tasks away from busy processors and to draw them toward their data. Each processor informs its neighbors about its load, when it has changed for more than 10% since the last information.

Several authors discuss different versions of the *bidding algorithm* (see for example [Hw82]), where the creating processor of a task asks its neighbors (not necessarily immediate ones) for bids to execute this task. The neighbors respond with a bid depending on their load. The processor with the best bid will receive the task. Some bidding algorithms allow a refusal of the task by the destination processor, if its load has increased too much in the meantime. So, several attempts to distribute a task may be necessary. A bidding algorithm is clearly an active strategy. Smith [Sm80] proposes a generalization of the bidding scheme, where idle processors may also request for tasks.

[†]The speedup on a multiprocessor system is measured against a single processor running the same program.

In [NX85] a passive scheme, the *drafting algorithm*, is suggested. Similarly to the gradient model, each processor has one of the three possible states light, normal, and heavy. When load changes, the neighbors are informed. A light processor sends draft requests to its heavy neighbors (not necessarily immediate ones), who each respond with a draft-age message (depending on the load). The processor with the highest draft-age is subsequently requested for a task, if it is still needed. Finally, if the requested processor is still heavy, a task will be given to the light processor.

In our framework, we have rather small tasks. Much more tasks are generated than processors are available. So, with a reasonable load balancing strategy only a small percentage of all produced tasks needs to be submitted to another processor. Hence, a long discussion between processors to find the right destination processor is not worthwhile, and most of the general algorithms are not appropriate.

# 3 The Strategies

We only consider strategies where tasks are sent to immediate neighbor processors, because we want to limit the distribution overhead and to preserve locality of computation as much as possible, which is important to reduce the communication. Experiments show that this is reasonable, though it may be interesting to transmit tasks to farer processors in the beginning stage of computation. This could allow a faster activation of the whole system, but we have not tested it yet.

## 3.1 Load Informations

Some load balancing strategies require an exchange of load informations. There are different possibilities, how a processor can inform its neighbors about its load. It can be done periodically, after each change in the number of tasks, on request, or when the number of tasks passes certain thresholds. In our experiments, the first and second strategy have produced to much overhead.

We have tested two strategies using thresholds. One causes the transmission of load messages, when the number of tasks increases from 1 to 2 and when it decreases from 1 to 0, i.e. the neighbors are informed whether work is available or not. The other strategy uses thresholds with exponential graduation, i.e. messages are sent when the number of tasks increases from $2^i - 1$ to $2^i$ (i=1,2,...) and when it decreases from $2^i$ to $2^i - 1$ (i=0,1,2,...). The idea behind this exponential graduation is that it gets less and less important to know the exact number of tasks the more of them are available. Note that we use a hysteresis to prevent the transmission of load informations after each change in the number of tasks. For simplicity, our load messages only contain the number of tasks. Of course, other informations, like the storage utilization, could also be included.

## 3.2 Passive Strategies

With passive strategies, we mean such strategies, where idle processors ask their neighbors for work. In principle, it is possible to ask when more than 0 tasks are still available.

Experiments show that this is not reasonable, since a single task may generate many subtasks. The investigated passive strategies differ mainly in the number of neighbors, which are asked for work at the same time, and in the number of requested tasks.

There are essentially two possible ways of handling requests when there is no work to distribute. The first is to store the information that the sender needs work and to give work to it when eventually some tasks are available. This method is only reasonable, when all neighbors have been asked for work. Otherwise, the idle processor does not know, when it should ask the other neighbors. The disadvantage of asking (many or) all neighbors simultanously is that they all may send tasks, while only one or a few are needed.

The second alternative is to respond with a message indicating that no work is available. When all neighbors give negative answers, it is good to wait for some time until the next request is started. Otherwise, the processors which are computing the (few existing) tasks are only delayed by their idle neighbors.

The advantage of passive strategies is that there is no (or little) overhead when all processors have work, which is the case nearly all the time for typical applications with massive parallelism like divide and conquer algorithms. Also, the main effort of load balancing is delegated to the idle processors. Hence, the busy ones are not further charged. On the other hand in the (usually short) beginning and final stage of computation, there is few work and requests only delay the busy processors.

We have tested six passive strategies. The first one, CYCREQ, works as follows. An idle processor asks its neighbors one after the other for one task. The requested neighbor either responds with a "no work" message or with a task. In the second case, no further request is started. Otherwise, the next neighbor is asked. If all neighbors have given a negative answer, the idle processor waits for some time until it again asks its first neighbor.[†] No load informations need to be exchanged. We have selected this strategy, since it is one of the simplest possible. By comparing it to more expensive strategies, it shows whether the overhead pays.

The second passive strategy, CYCREQ50, works like CYCREQ, but now the requested neighbor sends 50% of its tasks to the idle processor. CYCREQ50 allows a faster distribution of tasks among the processors, while it leads to less locality of computation.

Another stategy, MAXWREQ, works similar to CYCREQ, but now the neighbors are asked in descending order (with respect to their load). Consequently, an exchange of load informations is needed. MAXWREQ is similar to the strategy used in [BS81]. We here use exponentially graduated thresholds. MAXWREQ50 works like MAXWREQ, but 50% of the available tasks (instead of one) are sent to the asking neighbor.

The next passive strategy chosen, which we call MTMP1 (multiple tasks multiple processors), works as follows. An idle processor asks all its neighbors for a certain amount of their work. We have tested three ways of computing this amount:

[†]A delay of about 10 ms is reasonable in our application.

1) $\frac{1}{n+1}$ where n is the number of neighbors of the idle processor, which have distributable work

2) $\frac{max}{sum+max}$ where *sum* is the sum of loads of all neighbors and *max* is the maximum load of them

3) $\frac{wl(i)}{sum+wl(i)}$ where *sum* is defined like above and $wl(i)$ is the load of neighbor $i$ (with $i$ ranging over all neighbors)

The aim is always that after the initiated distribution all processors have nearly the same number of tasks and that this can be achieved with the smallest overhead possible. All methods need an exchange of load informations. These are transmitted on request, since it is important to have up to date informations. Figure 1 shows the resulting task distribution for an example situation and for each of the methods. Method 2 leads to the transmission of more tasks than method 1 in unbalanced situations (with respect to the load of the neighbors). With Method 3, neighbors, which have more tasks than the average of all neighbors, have to supply overproportionally many tasks.

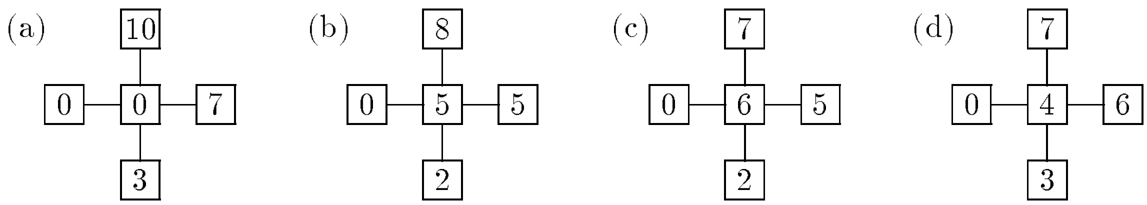


Figure 1: b), c), and d) show, how the tasks are distributed in situation a) using method 1, 2, and 3 respectively. Each square represents a processor and its label indicates the number of locally available tasks.

| | MTMP1 | | | MTMP2 | | |
|---|---|---|---|---|---|---|
| method | 1 | 2 | 3 | 1 | 2 | 3 |
| qsort(500) | 11.00 | 11.04 | 10.43 | 10.87 | 11.16 | 10.63 |
| towers(12) | 4.01 | 4.11 | 4.00 | 4.07 | 3.99 | 3.91 |
| queens(7) | 13.54 | 13.63 | 13.48 | 13.58 | 13.51 | 13.21 |
| map(5000) | 7.83 | 7.77 | 7.71 | 7.61 | 7.99 | 7.84 |
| fold(5000) | 7.25 | 7.26 | 7.21 | 7.24 | 7.42 | 7.30 |

Table 1: Run times (in seconds) of several example programs (see section 4) on a system with 12 processors for MTMP1 as well as MTMP2 and for method 1, 2, and 3.

The next strategy, MTMP2, works similar to MTMP1, but now the load informations are not transmitted on request. Instead, the simple threshold scheme (see section 3.1) is used for method 1 and the exponentially graduated thresholds are used for methods 2 and 3 So less communication is needed (compared with MTMP1) but the informations are less exact. For MTMP1 as well as MTMP2, method 3 is best (see table 1), but the differences are small. We will use MTMP1 and MTMP2 only with this method furtheron.

## 3.3 Active Strategies

Using an active strategy, tasks are distributed by their creating processor. The destination to which a task is sent depends on the load of the neighbors of this processor. Hence, an exchange of load informations is needed. Also, it may be taken into account, where the data of the task are sited. So, the access pathes to data can be shortened.

The disadvantage of active strategies is that an idle processor has to wait until a neighbor eventually decides to submit a task. Also, busy processors are additionally burdened with load balancing. On the other hand, busy processors are not delayed by idle neighbors, which is especially important in phases of low load.

We have tested two active strategies. The first, we call PERSEND. Here, it is periodically checked, whether there are neighbors which have more than $k$ tasks[*] less than the considered processor. While this is the case, tasks are sent one by one to the neighbor with the currently fewest tasks. For the exchange of load informations, we use exponentially graduated thresholds here.

The second active strategy, DIFFSCHED, is similar to (the sophisticated version of) diffusion scheduling of Hudak and Goldberg [HG84]. The idea is to send tasks away from busy processors and in the direction of their data. More concrete, let $T_{new}$ be a new task which has been created by processor $P_{src}$. $T_{new}$ is sent to processor $P_{opt}$ where $P_{opt}$ is chosen in such a way that

$$costs(T_{new}, P_{opt}) = min\{costs(T_{new}, P) | P \text{ is neighbor of } P_{src}\}$$

where
$costs(T, P) := w \cdot load(P) + refcosts(T, P)$,
$refcosts(T, P) := \sum_{i=1}^{n} dist(P, P_i)$, where $P_1, P_2, \ldots, P_n$ are the processors, which store the $n$ arguments of task $T$ in their memories,
$dist(P, P')$ is the distance from processor $P$ to $P'$ in the considered network, and
$load(P)$ denotes the number of tasks of processor $P$.

The weight $w$ is used to control the influence of load with respect to the reference costs[†]. For the exchange of load informations, we use here the exponentially graduated thresholds mentioned above.

## 3.4 Mixed Strategies

By mixed strategies, we mean strategies, where idle processors ask their neighbors for work and where tasks are additionally distributed by their creating processor. The idea is to combine the advantages of passive and active strategies. Since load informations have to be exchanged, more overhead is needed than for simple passive schemes like CYCREQ.

We have tested a strategy called ACTPAS which works as follows. Simple thresholds are used to exchange load informations. A message indicating that the load of its sender has decreased from 1 to 0 is interpreted as work request and answered by a task, if more

[*] $k = 1$ is best in our application
[†] we choose $w = 5/6$ in our application

than $k$[‡] tasks are available. Otherwise, it is stored that the requesting neighbor is idle, and if the number of locally available tasks exceeds $k$, the additional tasks are sent to idle neighbors. The communication overhead of ACTPAS is very small. Repeated requests are not needed.

## 3.5 The Task Queue

For the implementation [LK89], on which our experiments are based, it is possible to build a directed acyclic graph (dag), where the tasks are the nodes and each task points at all its subtasks (see figure 2). A task represents a function application from the point of view of the functional language.

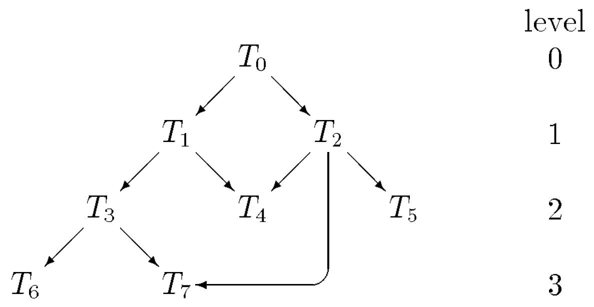


Figure 2: structure of task calls (example)

Often, tasks, which are closer to the (only) root in this dag, are more expensive than the farer away ones. The distance from the root is called the level in the following.

We try to send "big" tasks in preference to smaller ones. Otherwise, more tasks need to be transmitted leading to more communication overhead. We have tested two heuristics to tackle this, using different implementations of task queues. Each processor uses such a task queue to maintain the different task nodes, which are sited in the part of the computation graph assigned to its local store.

### 3.5.1 A Task Deque

Tasks with small levels (potentially "big" tasks) are mostly generated before the others. It is possible to use a *deque* (double ended queue) to store the tasks (see figure 3). Tasks which are generated by the own processor are inserted at one end of the deque, let us say the right one. When this processor needs a task for its own, a potentially small one is taken from this right end. Hence, the processor uses the deque in a stack-like way leading to a depth first evaluation strategy. When a neighbor needs a task, a potentially big one is taken from the left end. Tasks arriving from other processors, i.e. also potentially big tasks, are inserted at this end. Obviously, the insertion and deletion of tasks can be accomplished in constant time.

‡ $k = 2$ is best in our application

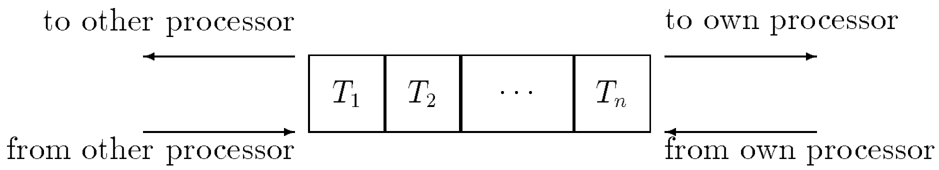


Figure 3: task deque

### 3.5.2 A Task Priority Deque

A little bit more sophisticated heuristic uses a *priority deque* [Kn73, p. 159] (see figure 4), which essentially consists of two heaps. The size of both heaps may only differ by one. The position of a task in the priority deque is determined by its level.

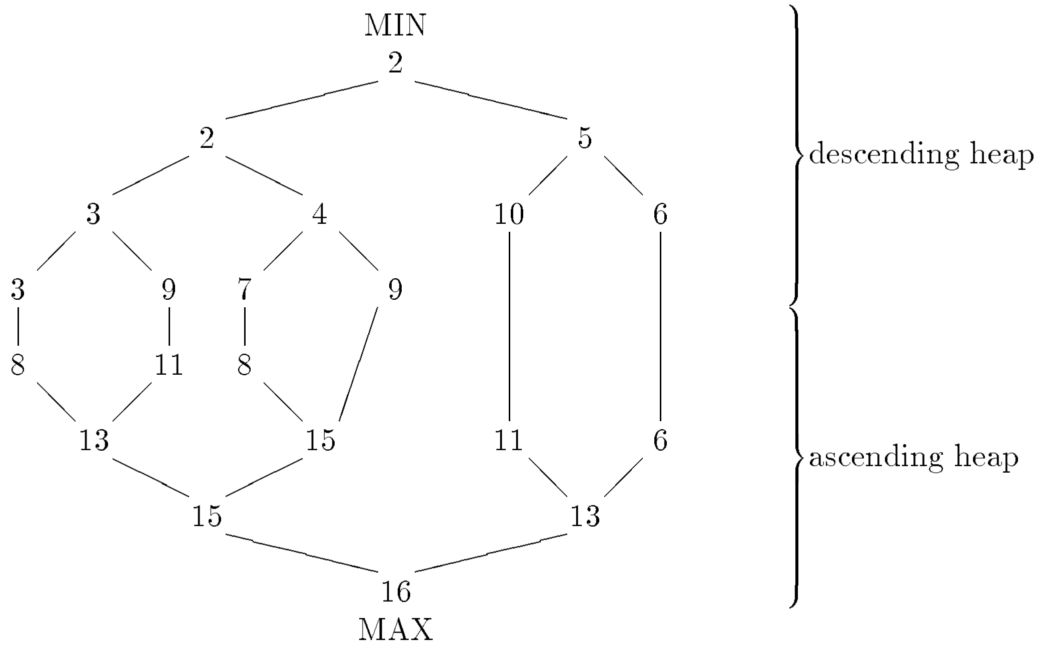


Figure 4: priority deque; each entry is represented by the level of the corresponding task

A heap is usually implemented by an array $A[1..n]$, where
$A[i] \leq A[2i]$ and $A[i] \leq A[2i+1] \quad (1 \leq i \leq \lfloor \frac{n}{2} \rfloor)$, or
$A[i] \geq A[2i]$ and $A[i] \geq A[2i+1] \quad (1 \leq i \leq \lfloor \frac{n}{2} \rfloor)$
depending on the fact, whether the heap is in ascending or descending order. It can be visualized by a (special) binary tree. To build a priority deque, an ascending heap $A$ of size $n$ and a descending heap $B$ of size $m$, with $m \leq n \leq m+1$, are connected "back by back", i.e.
$A[i] \leq B[i]$ for $i = 1, \ldots, m$ and
$A[n] \leq B[\lfloor n/2 \rfloor]$, if $n > m$.

A new element is inserted as $A[n+1]$, if $n = m$, or as $B[m+1]$ otherwise. If the new element is smaller than its "father" (according to figure 4), it is subsequently exchanged with its current father until it is greater than it or the root is reached. Otherwise, it is exchanged with its current son until it is smaller than it or the bottom is reached.

The minimal element can be deleted by moving the "last element" (i.e. A[n], if $n > m$, or B[m] otherwise) to the position of the minimal element (i.e. the root) and exchanging this element with its current smaller son until it is less or equal to this son. The maximal element can be deleted analogously.

Insertion of a new element and deletion of the minimal or maximal element can obviously be accomplished in logarithmic time (in the size of the priority deque). If this priority deque is used to store the distributable tasks of a processor, the minimal element (i.e. the task with the smallest level, which hence is a potentially big tasks) is preferably given to another processor, while the maximal element is preferably taken by the own processor.

# 4 Experimental Results

We have tested the different strategies using the implementation (described in [LK89] [Ku89] [Lo89]) of a functional language on a system of transputers (running with 20 MHz and 20 Mbit/s serial communication links). We here give a concise summary of our results. A more detailed description can be found in [Wa90]. The following example programs have been examined:

*fib:* the fibonacci-function (obvious exponential algorithm)

*qsort:* quicksort

*towers:* towers of Hanoi

*queens:* the problem to place $n$ queens, which cannot beat each other, on a $n \times n$-chess board

*map:* mapping a unary function (here the successor function) to all elements of a data structure, i.e.[†]
$map(f, [x_1, \ldots, x_n]) = [f(x_1), \ldots, f(x_n)]$

*fold:* combining the elements of a data structure by a binary associative function, i.e.
$fold(\oplus, [x_1, \ldots, x_n]) = x_1 \oplus \ldots \oplus x_n$

Most of the examples work structural recursive on one argument. During the computation of these problems, one can observe a short beginning stage, when the computation diffuses into the processor network, a stage, when all processors are busy, and a short final stage, when computation collapses.

Only quicksort has a more complicated recursion structure. Before each recursive call, the argument data structure needs to be splitted into two. The sequence of these split operations leads to a pulsating computation. After each split the number of permanently busy processors doubles until each processor has at least one substructure which it sorts by its own.

[†]We denote function application like usual in mathematics (instead of using juxtaposition like most functional languages) to improve the readability for non-functional programmers.

All examples but *fib* need data structures. Since the usual (ordinary represented) lists are not very suited for a parallel implementation, we use so called *sequences* [Ku89], which are essentially lists represented by binary trees. Sequences offer the possibility of treating the left and right subtree of a tree simultanously in a divide and conquer style, while usual lists allow only to handle the first element and the complete rest of the list in parallel.

Since the examples are well known, we omit a more detailed description. The interested reader may refer to [Ku89].

| problem | towers | | | queens | map | fold | qsort | fib |
|---|---|---|---|---|---|---|---|---|
| argument size | 8 | 10 | 12 | 7 | 5000 | 5000 | 1000 | 22 |
| PERSEND | 3.95 | 6.62 | 8.84 | 8.97 | 7.68 | 8.09 | 7.74 | - |
| DIFFSCHED | 4.17 | 4.82 | 6.81 | 7.98 | 6.56 | 6.05 | 6.29 | - |
| CYCREQ | 4.41 | 7.13 | 9.62 | *9.75* | *9.04* | 8.70 | 8.38 | *11.85* |
| CYCREQ (PD) | *4.50* | *7.24* | *9.67* | 9.67 | 8.71 | 8.79 | *8.91* | - |
| CYCREQ50 | 4.17 | 7.13 | 9.45 | 9.50 | 8.54 | 8.68 | 8.83§ | 11.75 |
| MAXWREQ | 4.17 | 6.87 | 9.42 | 9.61 | 8.46 | 8.48 | 8.66§ | - |
| MAXWREQ50 | 4.17 | 6.97 | 9.35 | 9.65 | 8.37 | 8.42 | 8.61§ | - |
| MTMP1 | 4.17 | 6.81 | 9.26 | 9.51 | 8.39 | 8.58 | 8.54§ | 11.69 |
| MTMP2 | 4.41 | 7.19 | 9.47 | 9.71 | 8.25 | 8.48 | 8.54§ | 10.68 |
| ACTPAS | 4.33 | 6.92 | 9.38 | 9.49 | 8.81 | *8.88* | 8.18 | 11.32 |
| ACTPAS (PD) | *4.50* | 6.97 | 9.45 | 8.73 | 8.64 | 8.68 | 8.35 | - |

Table 2: Speedups for several example programs on a system with 12 processors. PD indicates that a priority deque was used instead of the simple deque to maintain the available tasks.

| problem | towers | queens | map | qsort | fib |
|---|---|---|---|---|---|
| argument size | 12 | 8 | 5000 | 1000 | 28 |
| CYCREQ | 20.52 | 33.37 | 24.92 | 15.83 | *63.02* |
| CYCREQ50 | 19.95 | *37.50* | 27.14 | 15.36 | - |
| MAXWREQ50 | 22.08 | 35.89 | 29.69 | 16.07 | - |
| MTMP2 | 20.91 | 32.28 | 27.34 | 14.66 | - |
| ACTPAS | *24.42* | 33.69 | *32.20* | *17.87* | 58.66 |

Table 3: Speedups for several example programs on a system with 64 processors. PD indicates that a priority deque was used to maintain the available tasks.

Table 2 and Table 3 show the speedups on a system with 12 and 64 processors respectively. Since the measured times differ from run to run, all speedups given are computed using the average times of three runs. For the 12 processor system, we use a topology with diameter 3, where processor $i(= b_3b_2b_1b_0)_2$ is connected to processors $(i+3) \bmod 12$, $(i+9) \bmod 12$, and $(b_3b_2b_1\bar{b_0})_2$ $(0 \leq i \leq 11)$. For the 64 processor system, we use the twisted 5-cube [HK87], where each node consists of 2 connected processors (diameter 6). On the system with 64 processors, we have not tested all the strategies. Since we currently interpret the produced intermediate code in OCCAM, the absolute run times are not very fast (see table 4), but this is of minor importance. We are mainly interested in the speedups, which are possible using a certain load balancing strategy. Nevertheless, a modified compiler which produces transputer assembler code will soon be ready.

| problem | towers | | | queens | map | fold | qsort | fib |
|---|---|---|---|---|---|---|---|---|
| argument size | 8 | 10 | 12 | 7 | 5000 | 5000 | 1000 | 22 |
| run time (sec) | 1.83 | 7.47 | 30.07 | 106.80 | 53.42 | 50.56 | 176.77 | 43.14 |

Table 4: Run times of several example programs on one processor

*fib* needs very few communication. When a task is given to a processor, only the result has to be returned. Accesses to external data are not needed. Here, the simplest strategy, CYCREQ, is best. Any overhead does not pay. Speedups close to the number of processors are possible.

Active strategies are always worst. Especially, DIFFSCHED is to expensive and does not lead to a good load distribution. The other strategies show only small differences. Therefore, the best choice is of less importance than one might expect at a glance. CYCREQ is the best strategy for 12 processors, while ACTPAS is best for 64 processors. Hence, for small and medium size multiprocessor systems simple passive schemes are sufficient, while larger systems require a strategy with an active component to shorten the beginning stage of computation.

A comparison of the columns *towers(8)*, *towers(10)*, and *towers(12)* of table 2 shows that there is few influence of the argument size to the relation of the load balancing strategies. Of course, the possible speedups depend very much on the argument size. If a problem is too small for the considered multiprocessor system, only bad speedups can be expected. For problems using data structures, speedups up to 10 can be reached with 12 processors and up to 40 with 64 processors. Note that *towers(12)* and *map(5000)* are rather small tasks for the 64 processor system. The disappointing speedups for *qsort(1000)* depend on the recursion structure of quicksort mentioned above.

The priority deque only leads to an improvement for quicksort. Because of the more complicated recursion structure, it seems to be more important to distribute the right tasks. In most cases, the simple deque leads to a rather good distribution of work and to better speedups, since it is less expensive. In general, simple strategies are better than more complicated ones.

**Acknowledgements**
The authors would like to thank all people, who gave us helpful comments and suggestions, especially Rita Loogen.

# References

[BS81] F.W. Burton, M.R. Sleep: Executing functional programs on a virtual tree of processors, Conf. on Functional Programming Languages and Computer Architectures, 1981, 187-194

§with priority deque

[Bu88] G. Burn: A Shared Memory Parallel G-Machine Based on the Evaluation Transformer Model of Computation, Workshop Impl. of Lazy Functional Languages, Progr. Methodology Group, Report 53, Göteborg, 1988, 301-330

[HG84] P. Hudak, B. Goldberg: Experiments in Diffused Combinator Reduction, Proc. of the ACM Symp. on Lisp and Funct. Progr., 1984

[HK87] P.A.J. Hilbers, M.R.J. Koopman, J.L.A. van de Snepscheut: The Twisted Cube, PARLE Conference, 1987, in: LNCS 258/259, Springer

[Hw82] K. Hwang et al.: A UNIX-based Local Computer with Load Balancing, Computer, vol. 15(6), 50-56, 1982

[In88] INMOS Limited: Transputer Reference Manual, Prentice Hall, 1988

[Jo84] Th. Johnsson: Efficient Compilation of Lazy Evaluation, SIGPLAN Notices 13(6), 1984, 58-69

[Kn73] D.E. Knuth: The Art of Computer Programming, Vol.3, Sorting and Searching, Addison Wesley, 1973

[Ku89] H. Kuchen: Parallele Implementierung einer funktionalen Programmiersprache auf einem OCCAM-Transputer-System unter besonderer Berücksichtigung applikativer Datenstrukturen, Thesis, RWTH Aachen, 1989, (in german)

[LK87] F.C.H. Lin, R.M. Keller: The Gradient Model Load Balancing Method, IEEE Tr. on SE, SE-13, No. 1, 1987, 32-38

[LK89] R. Loogen, H. Kuchen, K. Indermark, W. Damm: Distributed Implementation of Programmed Graph Reduction, Proc. PARLE 1989, LNCS 365, Springer, 136-157

[Lo89] R. Loogen: Parallele Implementierung funktionaler Programmiersprachen, Thesis, RWTH Aachen, 1989, modified version published as: IFB 232, Springer Verlag, 1990, (in german)

[NX85] L.M. Ni, Chong-Wei Xu, T.B. Genderau: A Distributed Drafting Algorithm for Load Balancing, IEEE Tr. on SE, SE-11, No. 1, 1985, 1153-1161

[Sm80] R. Smith: The contract net protocol: High-Level communication and control in a distributed problem solver, IEEE Tr. Comput. C-29, 1104-1113, 1980

[Wa71] C.P. Wadsworth: Semantics and Pracmatics of the Lambda Calculus, Thesis, Oxford, 1971

[Wa90] A. Wagener: Lastverteilungsstrategien und Netzwerktopologien für eine Realisierung der parallelen abstrakten Maschine PAM, Master thesis, RWTH Aachen, 1990, (in german)