LogP Modelling of List Algorithms

W. Amme[1], P. Braun[1], W. Löwe[2], and E. Zehendner[1]

[1] Fakultät für Mathematik und Informatik,
Friedrich-Schiller-Universität,
07740 Jena – Germany.
E-mail: amme,braunpet,nez@idec02.inf.uni-jena.de

[2] IPD, Universität Karlsruhe,
76128 Karlsruhe – Germany.
E-mail: loewe@ipd.info.uni-karlsruhe.de

Abstract—
We present techniques for distributing lists for processing on distributed and shared memory architectures. The LogP cost model is extended for evaluating the schedules for the given problems and architectures. We consider both bounded and unbounded lists. The theoretical results are confirmed by measurements on a KSR-1.

Keywords— **Parallel List Computations, LogP Model, Runtime Predictions and Measurements**

I. Introduction

Most parallel algorithms operate on arrays because they are easy to distribute and redistribute, particularly, if their size is fixed, (c.f. [9]). However, arrays are not appropriate for a wide range of problems, e.g. for algorithms on graphs. But the lack of efficient distribution techniques of the elements forces programmers to use arrays instead of lists even if the latter is a better choice from a designer's point of view. The problem is even worse for the parallelization of sequential code, as the programmer usually does not consider parallel execution.

The obvious solution, which would be the list converted into an array before distribution, is too wasteful in memory. As shown below, it is also a bad choice w.r.t. the required time. Matsumoto, Han and Tsuda [8] proposed to collect only the addresses of list elements in an array and distribute this array. This saves memory if the element's size exceeds the size of an address. However, it only works for (virtual) shared memory systems. To decide whether it is more efficient than the former solution, we have to compare costs for local and non-local copy operations. The costs depend on the size of the data.

In general, to compare the different strategies for distribution, a cost model is required which would reflect the time for searching a list compared to the time for computation on the elements. Additionally, latency, overhead, and bandwidth for communication should be considered. The LogP-machine [2] is a generic machine model reflecting the communication costs with parameters *L*atency, *o*verhead, and *g*ap (which is actually the inverse of the bandwidth). The number of processors are described by parameter P. The parameters have been determined for several parallel computers [2, 3, 6, 4]. These works confirmed all LogP-based runtime predictions.

In a virtual shared memory machine, access to non-local objects is done by data transfer via a communication network. Of course, there is a *L*atency between initiating the access and receiving the object. Initiating and receiving may be two separate operations that cause *o*verhead on the processors. Between both operations, computations are possible. A *g*ap between two succeeding memory accesses must be guaranteed due to bandwidth limitations. Hence, virtual shared memory architectures are also covered by the LogP model.

With the cost model, we may predict the quality of each distribution technique in terms of the required execution time on a specific target machine. This paper is restricted to applications where the parallel computations on the single list elements are independent of each other, analyzed e.g. by techniques of [1]. Due to this restriction, our approach may achieve better results than a general solution, confronted with the same situation. The proposed algorithm is not intended to replace general techniques, but to complete them.

The paper is organized as follows: In section 2, we define the cost model. In section 3, we introduce the distribution algorithms. Section 4 compares the approaches w.r.t. execution times within the cost model. Section 5 confirms the results by measurements. Finally, we conclude our results and show directions of further work.

II. The Cost Model

The LogP cost model reflects communication costs but ignores computation times. However, since communication times are given in terms of machine cycles, these costs are comparable with execution times on the processors. In addition to the machine dependent parameters L, o, and g, we assume two further parameters:

- $e \cdot n$ defines the (maximum) costs for searching the list of size n on the processors of our parallel machine, and

- $k \cdot n$ denote the (maximum) costs for computations on n list elements.

Note that the latter depends on both, the processor architecture and the concrete algorithm. However, all parameters are easily computable at compile time and may therefore be used for optimization.

Sending (receiving) a message in a distributed memory system or initiating a far access in a shared memory system costs time o (overhead) on the processor. A processor can not send (receive) two messages and simultaneously initiate two far accesses, respectively, within time g, but may perform other operation immediately after a send (receive) and load or pre-fetch (store) operation, respectively. In a distributed memory system, the time between the end of sending a message and the start of receiving this message is defined as latency L. In a shared memory system, L denotes the time between the end of the initiation of a far access and the time the date is actually transfered.

If the sending processor is still busy with sending the last bytes of a message while the receiving processor is already busy with receiving, the send and the receive overhead for this message overlap. This happens on many systems, especially for long messages. Modeling this effect as a negative value for the latency avoids the distinction of cases in further calculations.

The time for communication depends on the amount of transfered data. This is not covered by the original LogP-machine model but considered e.g. in [4] where L, o, g are functions of the message size instead of constants. This paper uses $L(s)$, $o(s)$, and $g(s)$, respectively, to denote the latency, overhead, and gap, respectively, for data transfers of size s. A single (indivisible) item is of size 1. $L(0)$, $o(0)$, and $g(0)$ denote the cost for data of size zero. That include cost for function calls, etc. The observation from practice is that $L(s)$, $o(s)$, and $g(s)$ may be nicely approximated by linear functions. We write $\max(o,g)(s)$ for $\max(o(s),g(s))$ which is, by assumption, also a linear function.

Example 1 *For parsytec's PowerXplorer we measured the values of $L(s)$, $o(s)$, and $g(s)$ for a wide range of sizes s. We approximated following functions: $g(s) = 117 + 1.43 \cdot s$, $o(s) = 70 + s$, and $L(s) = -0.82 \cdot s$. These approximations have been confirmed by comparing predicted and measured execution times [4].*

III. Distribution Strategies

For the parallel processing of a linear list, it is necessary to map the set of loop iterations into the set of available processors. There are well-known scheduling methods for loops, which are exclusively based on scalar variables or array operands. Unfortunately these methods cannot be applied when parallelizing algorithm using linear lists, as the iteration space often must be known at compile time. Further, when using these methods we must be able to access all necessary dates in constant time. For linear lists these two demands are obviously not fulfilled.

Essentially, when the list length is known, distribution strategies for linear lists work in two phases:

1. The list elements must be assigned to the processors.
2. A parallel execution of the list is then performed.

For shared memory architectures, it is sufficient to assign the addresses to the processors. The actual data transfer is automatically performed if accesses to the addresses occur. In a distributed memory system, the list elements themselves must be distributed.

As already mentioned, Matsumoto, Han and Tsuda [8] offer an approach for shared memory systems that—after determination of the list length—stores the address of each list element in an array. This array is scheduled vertically to the processor set and then processed in parallel. Analogously on distributed memory systems, we may store the list elements in an array instead of their addresses. We call this method *vectorization* or *vector method.*

Vectorization takes additional memory of size $c*n$, where c denotes the size of an address (for shared memory machines) and the size of a list element (for distributed memory machines), respectively. This size may be reduced to the maximum size of data transfered to a single processor. For distributed memory architectures, this is a constant part of n, and for shared memory machines, it is even reduced to P. The former reduction is easily obtained by interleaving the copy and the distribute operations. The reduction to P is achieved as follows: We collect the entry elements l_i of a list in an array l. An entry element l_i is an anchor of the list's portion that is to compute on processor i. Clearly, $|l| = P$. Determination of the entry elements can be performed by a simple list crossing. In a second phase, each processor i begins working on the list item which corresponds to l_i. The processor stops when it reaches l_{i+1} or the last element. We call this approach *list method.*

In comparison to the classification above, we distinguish *step-by-step methods* from *pipeline methods.* In the former case, distribution and computation are done step by step. Obviously, each processor should work on at most $m = \lceil \frac{n}{P} \rceil$ elements sequentially to guarantee load balancing. In the pipeline method, the processors may start working immediately after they received their portion of the list.

Since the distribution of the elements and addresses, respectively, is not completed for all processors at the same time, uniform distribution does not guarantee load balancing. This problem is discussed in the subsections and .

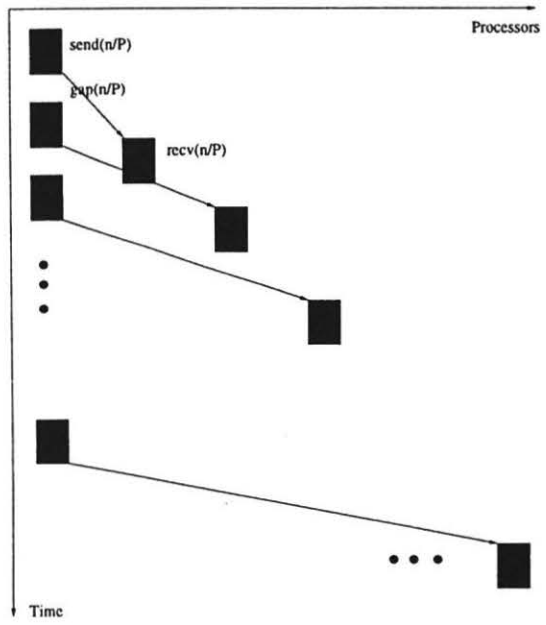

Fig. 1. Sequential distribution of array elements

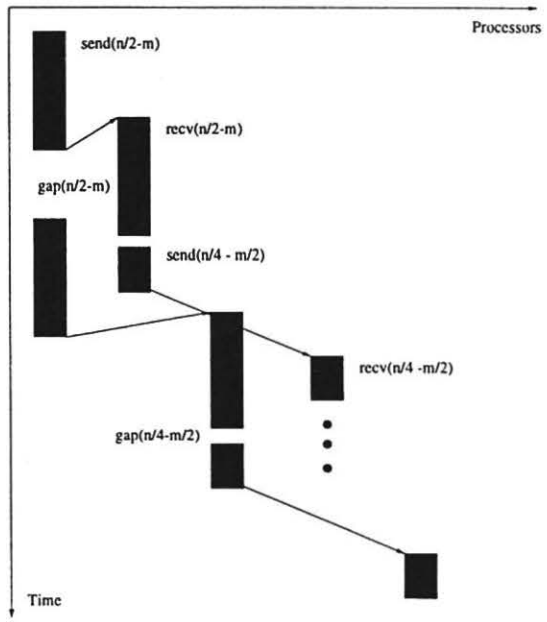

Fig. 2. Tree distribution of array elements

IV. Analyzing the Costs

This section compares the time for execution for the different distribution strategies w.r.t. the cost model from section II. The first two subsections consider distributed memory machines. The next subsection describes necessary modifications to apply the results to shared memory machines. The final subsection discusses the results and extends them to the case where n is not known at compile time.

A. Step-by-Step Method

First we analyze the step-by-step method using vectorization. Collecting n list elements in an array takes time $e \cdot n$. Thereafter, blocks of size $m = \lceil \frac{n}{P} \rceil$ must be distributed to the processors. Figure 1 sketches the distribution algorithm.

Lemma 1 *Sequential distribution of an array of size n is (upper-) bounded by*

$$t_{seq}(n) = (P-2) \cdot \max(o,g)(m) + L(m) + 2o(m).$$

Proof: $P-1$ blocks of size m must be sent sequentially. The last block leaves the source processor at time $(P-2) \cdot \max(o(m), g(m)) + o(m)$. Transmission requires time $L(m)$. The last processor requires time o for receiving the message. ◇

However, this is not always the best choice. Consider the following algorithm for distribution, a binary tree technique which may outperform the sequential distribution: The source processor possesses the complete array. If a processor P_i possesses an array equal in size to or smaller than m, it starts computation. Otherwise, P_i keeps m array elements and sends the remainder of the array to two other processors P_i^l and P_i^r, neither of which possess an array yet. P_i sends half of the array to P_i^l and the other half to P_i^r. Figure 2 sketches this distribution algorithm.

Lemma 2 *The tree distribution of an array of size n is (upper-) bounded by*

$$\begin{aligned} t_{tree}(n) \;=\; & \lfloor \log P \rfloor \cdot \\ & (\max(o,g)(\frac{n}{2}(1-\frac{1}{P})) + \\ & L(\frac{n}{2}(1-\frac{1}{P})) + \\ & 2o(\frac{n}{2}(1-\frac{1}{P}))). \end{aligned}$$

Proof: We consider the longest path in the broadcast tree. Its depth is $\lfloor \log P \rfloor$. Ignoring the m items remaining on the processors, each P_i sends half of its array to the two following processors. The i-th send operation on the longest path sends $s_i = \frac{n}{2^i}$ data items. It requires at most

$$t_i = \max(o,g)(s_i) + L(s_i) + 2o(s_i)$$

by the same reasoning as in lemma 1 (set $P = 3$ to see this). By using the linearity of $\max(o,g)(s), L(s)$, and $o(s)$, and iterating i from 1 to $\lfloor \log P \rfloor$, the proof is completed. ◇

It is not possible to guarantee that either algorithm offers better results in all situations. For small array sizes, the tree

distribution outperforms the sequential, for small numbers of processors, the opposite is true. We therefore combine both algorithms such that the sequential algorithm starts to distribute the array to $p \leq P-1$ processors sequentially. Each of the p processors obtains $\frac{n}{p}$ array elements. Then these elements are further distributed by the tree technique to maximal numbers of $\lfloor \log(\lceil \frac{P-p}{p} \rceil + 1) \rfloor$ processors.

Lemma 3 *The combined (sequential-tree-) distribution of the data is (upper-) bounded by*

$$\begin{aligned} t_{comb} &= (p-2)\cdot \max(o,g)(\frac{n}{p}) + L(\frac{n}{p}) + 2o(\frac{n}{p}) + \\ &\quad \lfloor \log(\lceil \frac{P-p}{p} \rceil + 1) \rfloor \cdot \\ &\quad (\max(o,g)(\frac{n}{2p}(1-\frac{1}{P})) + \\ &\quad L(\frac{n}{2p}(1-\frac{1}{P})) + \\ &\quad 2o(\frac{n}{2p}(1-\frac{1}{P}))). \end{aligned}$$

Proof-Sketch: Correctness follows immediately from the lemmas 1 and 2. ◇

Now we choose a value for p, such that t_{comb} is minimal. This depends, of course, on the LogP parameters and may be easily computed for concrete architectures.

Example 2 *For the LogP parameter functions from example 1, figure 3 shows the time (in μsec) for distributing $n = 1024$ and 10.000.000 array elements, respectively, depending on P. In the former case $p = 5$ is optimal, w.r.t. our algorithm, in the latter case a pure sequential distribution is most efficient.*

We compare now the time needed for distribution by vectorization with the list approach. It turns out that the list approach is always preferable because of its advantages in memory usage:

Lemma 4 *The list approach does not increase the distribution time compared to vectorization.*

Proof: The send operations must be performed sequentially. Instead of making a copy of the whole list first, and then distributing the copy, we interleave the copy and the distribute operations. Since each list element is copied only once, the cost of $e \cdot n$ for the copy operations is not exceeded. The costs for distribution of the copy remain the same. ◇

If the time $e \cdot n$ for collecting n list elements exceeds the difference between $g(n)$ and $o(n)$, the gap is always guaranteed. That means we may ignore $max(o,g)$ and use o instead.

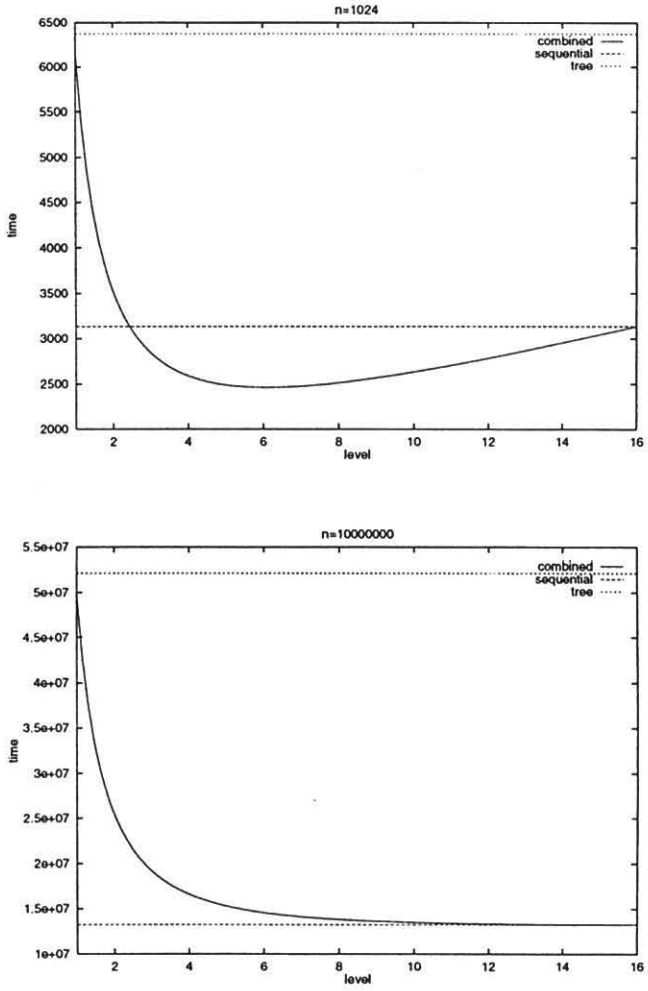

Fig. 3. Times for sequential, tree and combined distribution on a PowerXplorer

Note that for some LogP parameter functions, the combined algorithm is always faster then the pure sequential algorithm even if the size of the array approaches infinity[1]. However, these parameter functions describe only artificial architectures since both L and o must be constants, while g must increase with the size of the data transfer. Therefore, we assume that for existing parallel machines, the pure sequential distribution is the fastest for sufficiently large array sizes n. This means that for sufficiently large n, it is adequate to describe the costs of distribution by the (linear) function t_{seq} instead of t_{comb}.

The following theorem merges the results from this subsection:

Theorem 5 *Using the step-by-step approach, parallel computation of a list with n elements is (upper-) bounded by*

$$t_{step-by-step}(n) = t_{seq}(n) + e \cdot n + k \cdot (\frac{n}{P} + 1),$$

which is a linear function in n.

Proof: Because of lemma 1 and the time $e \cdot n$ for copying n list elements, distribution of the list takes time $t_{seq}(n) + e \cdot n$. Each processor works sequentially on $\lceil \frac{n}{P} \rceil$ elements, which explains the term $(\frac{n}{P} + 1) \cdot n$ as an upper boundary for computations. $t_{seq}(n)$ as well as $e \cdot n$ and $k \cdot (\frac{n}{P} + 1)$ are linear functions. Therefore, $t_{step-by-step}(n)$ is linear. ◇

B. Pipeline Methods

We use the sequential list approach to distribute the list elements according to the pipeline method. The question is: how many list elements should each processor get to guarantee that no unnecessary idle times occur? Let t_i be the time for collecting the list elements dedicated to processor p_i plus the costs for the corresponding send operation, $1 \leq i < P$. Because of the observations from the last subsection, t_i is defined by the following recurrence:

$$t_1 = o(n_1) + e \cdot n_1, \quad (1)$$
$$t_i = t_{i-1} + o(n_i) + e \cdot n_i, \quad (2)$$

where n_i is the amount of data sent to processor p_i. T_i denotes the time processor p_i needs to complete computations. Processor p_P is the source processor (the processor that possesses the original list). We set $t_P = t_{P-1}$. The source processor may start its computations if all messages have been sent. Obviously, it holds that

$$T_P = t_{P-1} + k \cdot n_P, \quad (3)$$
$$T_i = t_i + o(n_i) + L(n_i) + k \cdot n_i, \quad (4)$$

[1]To see this, compute a differentiation of t_{comb} with respect to p. For some values of the LogP parameter functions, this derivative is zero for $1 < p < P$ which is a minimum for t_{comb}. Maple does most of the job.

To avoid idle times on the processor, we solve the following linear equation system:

$$T_i = T_{i+1}, 1 \leq i \leq P, \quad (5)$$
$$n = \sum_{i=1}^{P} n_i. \quad (6)$$

Let $o(s) = o_0 + o \cdot s$ and $L(s) = L_0 + L \cdot s$. The linear equation system may be expressed symbolically as $1 \leq i \leq P-2$ by transforming the equations above.

$$\begin{aligned} T_i &= T_{i+1} \\ t_i + o(n_i) + L(n_i) + kn_i &= \\ t_{i+1} + o(n_{i+1}) + L(n_{i+1}) + kn_{i+1} & \end{aligned}$$

$$\begin{aligned} t_i + o(n_i) + L(n_i) + kn_i &= \\ t_i + 2o(n_{i+1}) + en_{i+1} + L(n_{i+1}) + kn_{i+1} & \end{aligned}$$

$$\begin{aligned} o(n_i) + L(n_i) + kn_i &= \\ 2o(n_{i+1}) + en_{i+1} + L(n_{i+1}) + kn_{i+1} & \end{aligned}$$

$$(o + L + k)n_i - (2o + L + e + k)n_{i+1} = o_0$$

By similar transformations we obtain

$$\begin{aligned} T_{P-1} &= T_P \\ (o + L + k)n_{P-1} - (o + L + k)n_P &= 0 \end{aligned}$$

We set $m_1 = o + L + k$ and $m_2 = -(2o + L + e + k)$. The general linear equation system for arbitrary LogP parameters is defined by

$$\begin{pmatrix} m_1 & m_2 & 0 & 0 & \cdots & 0 & 0 \\ 0 & m_1 & m_2 & 0 & \cdots & 0 & 0 \\ 0 & 0 & m_1 & m_2 & \cdots & 0 & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & 0 & \cdots & m_2 & 0 \\ 0 & 0 & 0 & 0 & \cdots & m_1 & m_2 \\ 1 & 1 & 1 & 1 & \cdots & 1 & 1 \end{pmatrix} \vec{n} = \begin{pmatrix} o_0 \\ o_0 \\ o_0 \\ \vdots \\ o_0 \\ 0 \\ n \end{pmatrix}$$

where the matrix is squared with P rows and columns. The i-th row, $1 \leq i < P-1$, represents the equations of the form $T_i = T_{i+1}$ (note the special case for $i = P-1$). The last row represents equation (6). The solution is real valued for $n_1 \cdots n_P$. We must round it out such that $n = \sum_{i=1}^{P} n_i$ still holds.

Theorem 6 *Using the pipeline approach, parallel computation of a list with n elements is (upper-) bounded by*

$$\begin{aligned} t_{pipe}(n) = & (o + e) * n + \\ & (P-1) * o_0 + (k - o - e) * (n_p + 1), \end{aligned}$$

which is a linear function in n.

Proof-Sketch: For the execution time $t_{pipe}(n)$ it holds that

$$t_{pipe}(n) = T_P + k\lceil n_p \rceil.$$

Easy substitutions for T_P using equations (1) to (6) complete the correctness proof. To note that $t_{pipe}(n)$ is linear, observe that n_p is a constant portion of n depending on the LogP parameters. ◇

C. Shared Memory Machines

For shared memory systems, the cost function for the data transfer differs slightly. For our purpose it is sufficient to consider load or pre-fetch operations to non-local data. Such an operation takes time o on the processor. L time units later, the data is available locally; the next load or pre-fetch must guarantee time g. However, if the transport of data from non-local to local memory may not be divided from the load operation (e.g. no pre-fetch operation exists), this differentiation is worth nothing. We have to consider far accesses as an indivisible function. In the latter case, we may integrate these costs into the costs k for the computations on the list.

In the step-by-step method, $\lceil n \cdot \frac{P-1}{P} \rceil$ elements must be considered for the determination of the entry elements. Afterwards, all processors execute $\lceil \frac{n}{P} \rceil$ elements in parallel. Altogether, this results in costs:

$$t_{step-by-step}(n) = e \cdot (n \cdot \frac{P-1}{P} + 1) + k \cdot (\frac{n}{P} + 1).$$

For the pipeline method, the quota of the list for each processor can be calculated by solving a linear equation system. All processors should stop their work at the same time. We set $m = \frac{k}{e}$ and obtain, by similar computations as in the last subsection:

$$\begin{pmatrix} -m & m+1 & 0 & 0 & \cdots & 0 \\ -m & 1 & m+1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \cdots & \vdots \\ -m & 1 & 1 & 1 & 1 & m+1 \\ 1 & 1 & 1 & 1 & 1 & 2 \end{pmatrix} \vec{n} = \begin{pmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ n \end{pmatrix}$$

Example 3 *Table I shows the proportional distribution of the list to four processors for different cost relations m.*

The costs for executing the whole list can be determined by n_1.

$$t_{pipe}(n) = (n_1 + 1) \cdot (e + k).$$

D. Discussion

The pipeline method is always faster then the step-by-step method, if $P > 1$ and communication is not 'free'. This is obvious as idle times between receiving the data and starting computations is saved, while load balancing is guaranteed.

TABLE I
PROPORTIONAL DISTRIBUTION OF A LINEAR.

m	P_1	P_2	P_3	P_4
1	50.00	25.00	12.50	12.50
2	39.13	26.09	17.39	17.39
3	34.78	26.09	19.57	19.57
4	32.47	25.97	20.78	20.78
5	31.03	25.86	21.55	21.55
10	28.07	25.52	23.20	23.20
50	25.62	25.12	24.63	24.63
100	25.31	25.06	24.81	24.81
250	25.12	25.02	24.93	24.93

We assumed that n is known in advance. If only upper bounds for n are statically known, the (upper) time bounds are still guaranteed since they are monotonely increasing with n. Now we drop this assumption and claim that N is a random variable with expectation $n = \mathbf{E}[N]$.

Theorem 7 *Let $T_{step-by-step}(n)$ and $T_{pipe}(n)$ be random variables for the time required for computation on lists with expected length n using the step-by-step and the pipeline methods, respectively. For these expectations it holds that*

$$t_{step-by-step}(n) = \mathbf{E}[T_{step-by-step}(n)], \text{and}$$
$$t_{pipe}(n) = \mathbf{E}[T_{pipe}(n)].$$

Proof: $t_{step-by-step}(n)$ and $t_{pipe}(n)$ are linear in n, see theorems 5 and 6. Because of the linearity of the expectation operator, the theorem holds. ◇

When the number of list items is unknown and its variance is too large, the list length may also be quantified by a complete list crossing at runtime. The calculation of the last subsections then occur also at runtime. This is not too expensive because the linear equation systems give a percentage of the list length n scheduled to each processor, if n is variable. We should only count the additional costs for the list crossing. Obviously, the described methods are easily extendable to any linear cost function for communication of list elements and succeeding parallel computations.

V. MEASUREMENTS

All methods have been implemented on a KSR-1 system [7] with eight processors, each with eight MByte local memory. The KSR-1 is a virtual-shared-memory-system, i.e., each processor has its own local memory, but there is only one global address space. Every memory cell can be accessed by every processor through a communication network called ALLCACHE-engine.

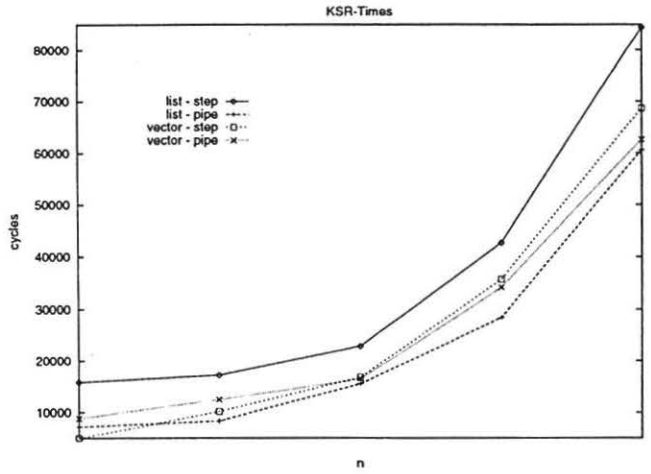

Fig. 4. Execution times for $m = 5$ on a KSR-1 with 8 processors.

Figure 4 shows the results of our measurements for $m = 5$. The performance is gaged in clock cycles for different list lengths ($n = 10, \cdots, 250$). It shows that, except for small n, the list approach with the pipeline method is faster than all other methods. This ranking remains the same for increasing m. The relative difference gets smaller for increasing computation costs k.

TABLE II

SPEED UP FOR PARALLEL LOOPS ON ARRAYS VERSUS LISTS.

P	array	list
2	1,46	1,97
3	2,34	2,79
4	2,67	3,49
5	4,03	3,76
6	5,33	4,65
7	6,40	6,26

With the list approach combined with the pipeline method, we are able to achieve a speed-up of 6.26 on the eight processors. Compared to parallel execution on an array (instead of a list), the parallel computation on a linear list is only slightly less effective now. Table II shows this behavior.

VI. Conclusion and Future Work

We compared methods for distributing lists for processing on parallel machines. We used the cost model of the LogP machine [2] to describe the communication cost on target architectures. Computation costs are given by linear functions in terms of the size of the lists. We discussed sequential as well as tree techniques for the distribution of the list elements, and derived an algorithm that combines both strategies. Uniform distribution of the list elements to the processors leads to idle times because an efficient distribution is not balanced, see subsection . These idle times may be avoided by pre-computations at compile time, see subsection and 4.3. Optimizations require either the length of the list, or an upper boundary of this length, or its expectation. We confirmed the results by measurements on a KSR-1, see section V.

The results may also be extended to:

1. distribution of arrays,
2. other distribution algorithms, and
3. arbitrary linear cost functions describing communication, synchronization and computation costs of programs of parallel architectures.

The latter leads to a framework for designers of parallel programs which gives a uniform view on distributed as well as shared memory machines. To validate this statement, we will continue to apply our methods to other parallel machines.

References

[1] W. Amme, E. Zehendner: Data dependence analysis in programs with pointers. Proc. 6th Workshop on Compilers for Parallel Computers (CPC'96), Aachen, 1996. Konferenzen des Forschungszentrums Jülich, Vol. 21, 1996, p. 371-382.

[2] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: Towards a realistic model of parallel computation. In *4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP 93)*, pages 1–12, 1993. published in: SIGPLAN Notices (28) 7.

[3] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP: A practical model of parallel computation. *Communications of the ACM*, 39(11):78–85, 1996.

[4] J. Eisenbiegler, W. Löwe, and A. Wehrenpfennig. On the optimization by redundancy using an extended logp model. In *International Conference on Advances in Parallel and Distributed Computing*. IEEE Computer Society Press, 1997.

[5] W. Löwe, W. Zimmermann, and J. Eisenbiegler, Optimizing parallel programs on machines with expensive communication. *EUROPAR' 96. Parallel Processing*, 1996.

[6] B. Di Martino and G. Ianello. Parallelization of non-simultaneous iterative methods for systems of linear equations. In *LNCS 854. Parallel Processing: CONPAR'94–VAPP VI*, pages 254–264. Springer, 1994.

[7] Kendall Square Research, KSR/Series Parallel Programming, KSR/Series Performance und KSR/Series C Programming

[8] A. Matsumoto, D. S. Han, T. Tsuda: Alias analysis of pointers in Pascal and Fortran 90: dependence analysis between pointer references. Acta Informatica, Vol. 33 (1996) 99–130.

[9] H. Zima, B. Chapman: Supercompilers for parallel and vector computers. New York: ACM Press, 1990.