Experiments with Parallel Algorithms for Combinatorial Problems

G.A.P. Kindervater

Centre for Mathematics and Computer Science, Amsterdam

H.W.J.M. Trienekens

Erasmus University, Rotterdam

ABSTRACT

In the last decade many models for parallel computation have been proposed and many parallel algorithms have been developed. However, few of these models have been realized and most of these algorithms are supposed to run on idealized, unrealistic parallel machines.

The parallel machines constructed so far all use a simple model of parallel computation. Therefore, not every existing parallel machine is equally well suited for each type of algorithm. The adaptation of a certain algorithm to a specific parallel architecture may severely increase the complexity of the algorithm or severely obscure its essence.

Little is known about the performance of some standard combinatorial algorithms on existing parallel machines. In this paper we present computational results concerning the solution of knapsack, shortest paths and change-making problems by branch and bound, dynamic programming, and divide and conquer algorithms on the ICL-DAP (an SIMD computer), the Manchester dataflow machine and the CDC-CYBER-205 (a pipeline computer).

1980 Mathematics Subject Classification: 90C27, 68Q10, 68R05.

Key Words & Phrases: parallel computer, SIMD, MIMD, pipelining, dataflow, branch and bound, dynamic programming, divide and conquer, knapsack, shortest paths, change-making.

Note: This paper appeared in European Journal of Operational Research, Vol. 33, pp 65-81, 1988.

1. Introduction

In the last decade computer scientists have proposed many parallel computers, based on various models of computation and architectures. However, so far only a few of these parallel computers have been built, due to today's technical limitations. All existing parallel computers use the easier models of computation and the easier architectures. Therefore these machines are less powerful than the theoretical models.

In the mean time operations researchers have developed many parallel algorithms for combinatorial problems. Nearly all these algorithms are based on models of parallel computation which are too complicated to be used in today's parallel computers. As a result, few of these algorithms have actually been implemented today simply by lack of the parallel computers needed.

Report 8550/A
Erasmus University Rotterdam, Econometric Institute
P.O. Box 1738, 3000 DR Rotterdam, The Netherlands

It is possible to execute these algorithms on some of today's parallel machines either by emulating the desired model of computation on such a machine or by reformulating the algorithm. However, such an emulation or reformulation increases the overall complexity of the algorithm or obscures the essence of the algorithm. Sometimes this effect is so severe that a particular parallel computer appears to be completely unsuited for executing certain types of algorithms.

Our purpose has been to gain insight in the behavior of some standard techniques for combinatorial problems, such as *branch and bound*, *dynamic programming* and *divide and conquer*, on some existing parallel computers. We considered three well known types of combinatorial problems and three parallel computers. The problems were *knapsack*, *shortest paths* and *change-making*; the machines were the *ICL-DAP* (an SIMD processor array), the *Manchester dataflow machine* (an experimental MIMD dataflow computer) and the *CDC-CYBER-205* (a pipeline machine that might be classified as an SIMD machine).

It turns out that the SIMD machines are very efficient in executing synchronized algorithms that contain regular computations and regular data transfers. In these types of algorithms all the processors always execute the same instruction on data residing at the same place in their local memories and all the data travel the same way. As soon as the computations or the data transfers become irregular or asynchronous, the SIMD machines become much less efficient. So, the SIMD machines are very good for dynamic programming, but bad for branch and bound or divide and conquer.

The concept of dataflow appears to be very promising for executing parallel algorithms, especially algorithms with irregular computations and data transfers like branch and bound or divide and conquer. The performance of the Manchester dataflow machine is, however, limited by its experimental character. It has amongst others a small memory capacity and overall throughput. But we hope that this performance will be improved by ongoing research.

2. Relevant Aspects of the Parallel Computers Used

In this section we will give a brief overview of the basics of parallel computation, followed by a short description of the three parallel computers we used. We will emphasize those features that are relevant for the analysis of the performance of the combinatorial algorithms we studied.

The main models of computation are *control driven*, *data driven* and *demand driven* [Treleaven, Brownbridge & Hopkins 1982]. In control driven computers the user has to specify the exact order in which the computations must be performed and also which operations can be performed in parallel. In the data driven model an operation can be performed as soon as all its operands are available and in the demand driven model an operation can be initiated as soon as its outcome is needed. All sequential computers use the control driven method. At present, most of the existing parallel computers also use this method. In recent years a few data driven computers, called dataflow machines, have been built [Watson 1984], but these machines are still in their infancy. Today, there are no working computers using the demand driven model of computation; yet several are being developed, for example the ALICE machine at Imperial College, London [Darlington & Reeve 1981].

Within each of these models there are further differentiations, for example in the way the various processors (processing elements) communicate. Another is the independence of the processing elements [Flynn 1966]. In *MIMD* (multiple instruction stream, multiple data stream) computers the different processors are allowed to perform different types of instructions on possibly different data at the same time. In *SIMD* (single instruction stream, multiple data stream) computers all processors are restricted to perform the same type of operation at a time.

There are two kinds of parallelism a computer can obtain while executing a program. The first one is a *coarse grained* one. This kind of parallelism occurs when a program contains certain statements that can be executed in parallel. For example the body of a for loop (which can be executed in parallel for each value of the loop index as long as these iterations are independent of each other) or a sequential list of statements which are independent of each other (and therefore can be executed concurrently). The second kind of parallelism is a *fine grained* one. This parallelism occurs if there is parallelism within a statement. For example the assignment of a new value to an element of a multi-dimensional array (the indices of the array element can be computed in parallel) or the use of multiple processors for a multiplication or

division.

2.1. The ICL-DAP

The ICL Distributed Array Processor is a commercially available SIMD computer with 4096 processors. These processors are located at the nodes of a 64 by 64 mesh. Each processor is connected to its four neighbors, with wrap around connections at the boundaries, and has its own local memory; cf. Figure 1. Software makes it possible to look at the 4096 processing elements as if they were located in a one dimensional array, where each processor is connected to only two neighbors. The processors are capable of simultaneously performing the same instruction on local data, with the restriction that the data have to reside at exactly the same place of the respective local memories. It is possible to *mask* a processor, which has the effect that the result of the instruction executed is not stored. Masking is effectuated by local data. It enables the use of conditional operations.

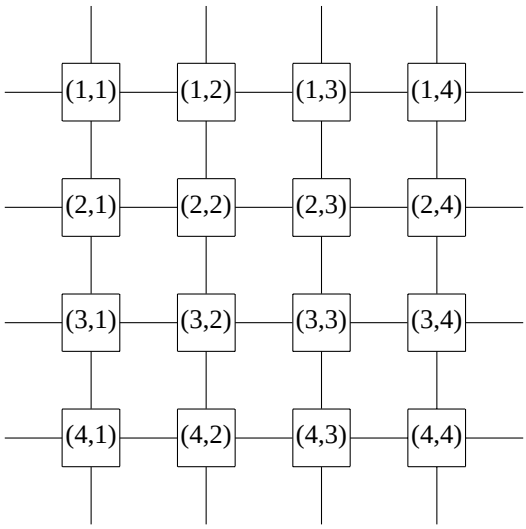


Figure 1. A 4×4 DAP.

Programs are executed on the ICL-DAP through a host computer. The host translates the program into DAP machine code and stores the machine code program and its data in the DAP. After that, control is given to the DAP, which performs the program on the data. After the DAP has finished, control is returned to the host and the host extracts the results from the DAP [ICL 1981].

The ICL-DAP can be programmed in the high-level language DAP-FORTRAN [ICL 1979]. This is an extension of standard FORTRAN with vector and matrix instructions, which can be used to process the elements of a vector or a matrix in parallel. The DAP or the FORTRAN compiler do not detect any parallelism in a program on their own accord. The programmer has to detect the parallelism himself by analyzing the algorithm. By invoking the vector and matrix instructions of DAP-FORTRAN he can state explicitly which operations must be performed in parallel. The parallelism thus obtained is a coarse grained one at the algorithmic level.

The vector and matrix instructions perform their parallel operations on vectors of dimension 64 or 4096 or on matrices of dimension 64 by 64 respectively. In performing operations on vectors of dimension 64, 64 processing elements cooperate in handling one vector element. If a particular problem is too big to fit in such a vector or matrix, the programmer has to divide the problem into subproblems fitting in these vectors or matrices and the solutions of these subproblems must be combined sequentially. This corresponds to simulating a DAP of bigger dimensions on a DAP of dimension 64 by 64. So, the DAP is best suited for problems of dimension at most 64 or 4096.

Although the DAP is capable of executing programs written in standard FORTRAN, no instructions of these programs are executed in parallel.

The performance of a program is measured by counting the number of instructions executed by the DAP. To get an estimation of the CPU time, the number of instructions is multiplied by the average time needed for an instruction. This way of timing neglects the differences between execution times of the various instructions. There is no way to measure exactly the CPU time used by the DAP (especially since the DAP can be used shared with another user).

2.2. The Manchester Dataflow Machine

Dataflow is a technique for representing computations in terms of directed graphs. The nodes of the graph are instructions to be performed and the arcs are data routes. The data transmitted over the data routes are represented as *tokens*. A node accepts the tokens from its incoming arcs, performs an operation on them and sends the results away on its outgoing arcs. Whether or not two nodes can be executed concurrently depends on whether or not one of the two nodes needs the output of the other as input. Arcs not starting at a node receive the input data and arcs not ending at a node produce the output.

A node is *enabled* (can start its execution) as soon as the required tokens have arrived on the incoming arcs. The execution of a node may not be immediate, but will happen eventually. Also the time needed to execute instructions or to transport tokens from one node to another may vary. It is assumed, however, that all these times are finite. The computation is completely asynchronous. Therefore, it can happen that tokens have to wait for others on incident input arcs. A second consequence is that a dataflow graph in general allows for different execution sequences.

Figure 2 shows a dataflow graph calculating x^2-xy using primitive boxes DUP (which duplicates its input), $\uparrow 2$ (which produces the square of its incoming value), $\times$ (which multiplies its inputs with each other) and – (which subtracts the right input from the left input).

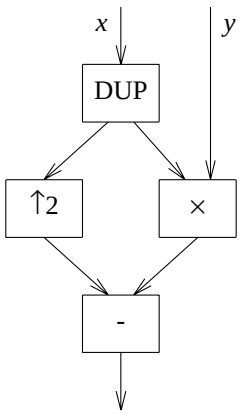


Figure 2. A dataflow graph.

A possible execution sequence is shown in Figure 3; stars (∗) represent the generated tokens moving through the graph.

To exploit the parallelism contained in the dataflow model of computation, an unconventional hardware organization is required. For building a general purpose dataflow machine a data structure of some sort is needed for representing the dataflow graph of a particular problem. On the Manchester dataflow machine this data structure consists of labeled nodes containing the instruction to be performed and the destination(s) of the result(s).

The Manchester dataflow machine is an experimental computer, which consists of a ring of elements each performing a special task (see Figure 4). A token consists of a value and a destination node. The *token queue* buffers the incoming tokens and sends them, one at a time, to the matching unit. The *matching unit* is an associative memory, which groups tokens with the same destination node into packages and presents

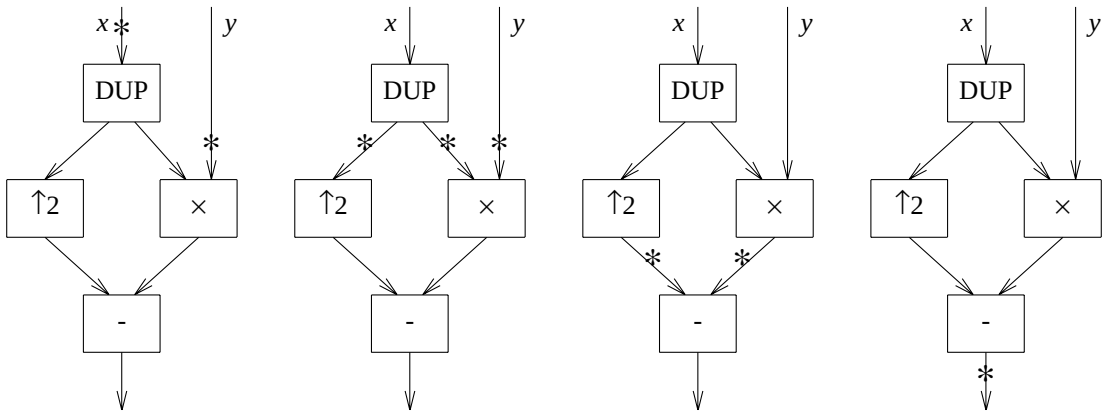


Figure 3. An execution sequence.

them to the node store. In the matching unit, tokens are stored until their partners have arrived. For efficiency reasons the machine only allows packages of one or two tokens. The *node store* contains the dataflow graph to be executed. Each node consists of the instruction to be performed and the destination(s) of the result(s). The node store adds this information and sends the whole as an executable package to the processing unit. In the *processing unit* the package is sent via a distribution network to an idle processing element. After processing, the results arrive via a arbitration network at the switch. At the *switch* input (output) tokens are inserted into (removed from) the ring; non-output tokens are sent along to the token queue.

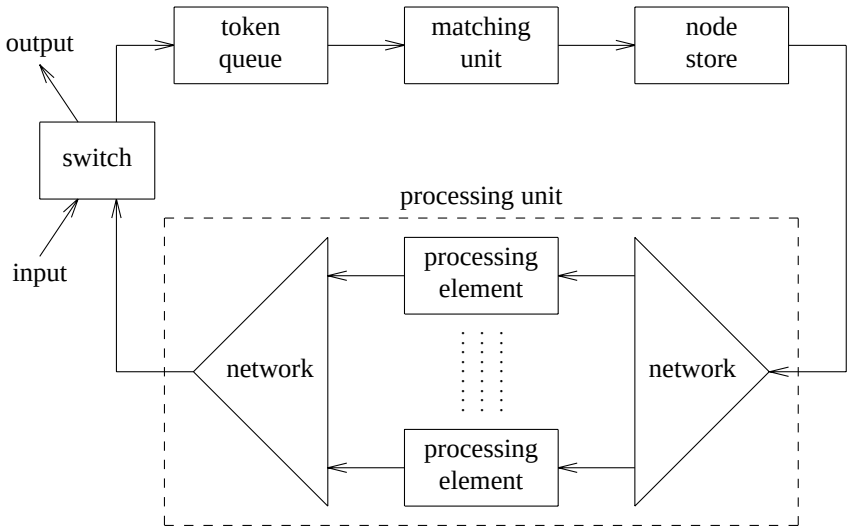


Figure 4. The Manchester dataflow machine.

The processing unit makes use of fine grained MIMD-type parallelism (the processing elements process different executable packages simultaneously). It is clear that the degree of parallelism depends on the number of processing elements. On a higher level, the units in the ring continuously perform operations on the flow of packages, which gives a parallelism as in an assembly line.

The critical part of the system is the matching unit. All units can be tailored to meet its maximum throughput capacity easily, e.g., the speed of the processing unit can be adapted by adding (or removing) processing elements. A way to overcome this bottleneck is to construct several rings and connect them through the switch, which then becomes a full interconnection network.

The Manchester dataflow machine can be programmed in the high-level language SISAL (Streams and Iteration in a Single Assignment Language) [McGraw et al. 1984]. SISAL has no concept of sequential execution and no direct control statements such as GOTO. To avoid the ambiguities that might arise from reassigning values to variables, the language allows each variable to be assigned only once in a program. In loops a construct is provided to reassign variables. Further, SISAL has strict type and scope rules and prohibits all forms of side effects (this because side effects introduce data dependencies which are very hard to catch). More about single assignment languages can be found in [Ackerman 1982]. The nature of a single assignment language makes it, in comparison with FORTRAN or PASCAL, easy to compile a program into a dataflow graph.

Due to the computation model used, the parallelism in a program is detected by the dataflow machine itself. The only thing a programmer can do is trying to specify his program in such a way that the dataflow graph constructed is as broad as possible.

The Manchester dataflow machine is operated in the same way as the ICL-DAP. Program development and compilation is done on a host computer. The host first stores the generated dataflow graph in the node store and then inserts the data via the switch in the ring. The data activate the dataflow machine. Output tokens leave the ring via the switch and are collected on the host.

It is very difficult to measure the performance of a program on the Manchester dataflow machine. The only measurement that can be made is the execution time until the arrival of the first output token at the host. This measurement is not very informative because after this arrival the computations may continue. This drawback can be overcome by reorganizing a program in such a way that it produces a single output token at the end of its execution.

A better insight into the performance of a program can be gained by emulating the dataflow machine on a sequential computer. Therefore the Manchester Dataflow Group developed an emulator. To keep this emulator manageable, some simplifying assumptions about the system architecture had to be made. The principal assumptions are the following:

(*i*) The time needed to execute an instruction is equal for all instructions (execution therefore proceeds in discrete steps of equal time).

(*ii*) An unlimited number of processing elements can be used during any one time step and the throughput capacity of the ring is infinite.

(*iii*) Output from an instruction can be transmitted to a successor instruction within the execution time period.
These assumptions are unrealistic. But they are helpful in making an approximate characterization of a program.

The two fundamental time measurements are S_1, the total number of instructions executed (which would be the number of time steps if only one processing element was available) and S_∞, the number of time steps with an unlimited number of processing elements (which is the critical path length of the underlying dataflow graph). The ratio $\pi = S_1/S_\infty$ gives a measure of the average parallelism in a program. A more detailed trace of the behavior of a program can be obtained if desired.

A detailed description of the Manchester dataflow machine can be found in [Gurd, Kirkham & Watson 1985]

2.3. The CDC-CYBER-205

The CDC-CYBER-205 is a commercially available computer able to perform the same operation on all elements of a vector of variable length in a pipelined way. In order to do this, the functional units are segmented. Each segment does a small part of the operation to be performed and sends the results to its neighboring segment. In this way a pipeline is created; cf. Figure 5. The segmentation makes it possible to deliver, after a certain start-up time which is independent of the size of the vector, a result of such a vector operation at each clock cycle. When executing vector instructions it is possible to specify whether or not a generated result must be stored. This enables the use of conditional operations.

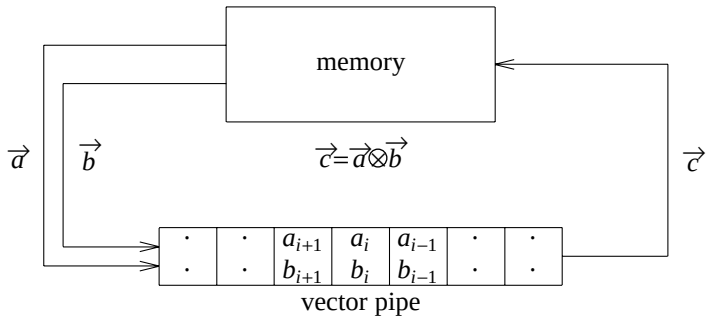


Figure 5. The CYBER-205.

Due to its capability of performing vector operations the CYBER-205 is very similar to an SIMD computer, although strictly spoken the results are generated in a sequential way.

The CYBER-205 can be programmed in the high-level language FORTRAN-200 [CDC 1983]. This extended standard FORTRAN contains vector instructions, which process vector elements in a pipelined manner. The FORTRAN-200 compiler is able to detect some parallelism in the program by trying to vectorize **do**-loops, but far from every **do**-loop can be vectorized in this way. By using the vector instructions the programmer can specify which operations must be pipelined. However, this means that he has to analyze his algorithm and detect the parallelism himself. The parallelism thus created is a coarse grained one on the algorithmic level.

The CYBER-205 is capable of executing a program written in standard FORTRAN, but unless the compiler is told to try to vectorize this program and manages to vectorize at least part of it, no part of the program is executed in a pipelined manner.

The performance of a program on the CYBER-205 is measured by the CPU time needed to execute the program.

3. The Performance of Some Combinatorial Algorithms

3.1. Change-Making

Change-making is the following combinatorial problem: given a coinage system, determine the number of different combinations of coins with which a certain amount can be paid for without change.

Let n be the number of coins in the coinage system, let v_i $(i=1,...,n)$ be the value of coin i and let $P(z,i)$ $(z\geq 0,\ i=1,...,n)$ be the number of different combinations amount z can be paid for when only coins with value $v_1,...,v_i$ may be used.

Let Z denote the amount to be paid. We then have to determine $P(Z,n)$. The following recursive equation holds:

$$P(z,i)=\sum_{k=0}^{\lfloor z/v_i\rfloor} P(z-kv_i,i-1) \quad (z\geq 0,\ i=2,...,n).$$

The initial conditions are:

$$P(z,1)=\begin{cases}1 & \text{if } z=0 \bmod v_1,\\ 0 & \text{otherwise.}\end{cases}$$

The change-making problem can also be seen as a network problem. Let $G=(V,A)$ be a directed graph. The set of vertices V consists of nodes $v(z,i)$ $(z=0,...,Z,\ i=0,...,n)$. There is an arc from node $v(z_1,i)$ to $v(z_2,i+1)$ if and only if $z_2=z_1+kv_{i+1}$ for some nonnegative integer k $(i=0,...,n-1)$. The change-making

problem is equivalent to the problem of determining the number of different paths in G from $v(0,0)$ to $v(Z,n)$. A path α is different from a path β if α contains an arc not in β or β contains an arc not in α.

The change-making problem can be solved by divide and conquer and by dynamic programming. Both techniques use the above recursion, but they use them in reverse directions.

Divide and conquer solves the problem by splitting it into easier problems, solving these easier problems and combining their solutions to the solution of the original problem. The easier problems are of the same form as the original one and are solved in the same way.

Dynamic programming combines the solutions of smaller problems to the solution of a bigger problem. Starting with trivial problems, eventually the solution of the problem to be solved is constructed.

If the change-making problem is viewed as a graph problem, dynamic programming is a form of implicit enumeration of all paths in the graph whereas divide and conquer reduces to explicit enumeration of all paths. In the divide and conquer method, the graph is a tree and the enumeration of all paths is in essence a tree traversal.

3.1.1. Implementing Divide and Conquer

Divide and conquer boils down to a direct evaluation of $P(Z,n)$ through the recursion given above. For an SIMD machine it is necessary that all processors perform the same instructions. However, the number of subproblems in which a particular (sub)problem is split depends entirely on the data of that instance. But there exists an upper bound on this number and by adding dummy subproblems one can arrange that each subproblem is split in the same number as the others, i.e., a number equal to this upper bound. In this way, the processors can always execute the same instructions at a time.

On an SIMD machine, the obvious implementation is that each processor takes care of one subproblem. However, the number of subproblems created is exponential. Therefore, only very small size problems can be solved in parallel. It is also possible to use different processors for solving the change-making problem for different amounts - where the coinage system remains the same - at the same time. Each processor then solves a problem sequentially and the time needed to do this equals the time needed to solve the biggest problem. In our investigations we only wanted to solve one instance of the change-making problem at the same time. For this case the SIMD machines are not so well suited and therefore, we implemented the divide and conquer approach only on the Manchester dataflow machine.

Recursion is a natural technique for programming divide and conquer. This technique results in a straightforward and elegant implementation on the Manchester dataflow machine. Due to the fine grain parallelism of this machine, computations are performed asynchronously and in parallel wherever possible. The computations however have to be synchronized for combining the solutions of the subproblems. The exact order in which the computations are performed is nondeterministic. The synchronization before combining the solutions of the subproblems ensures that this order is a feasible one.

3.1.2. Implementing Dynamic Programming

The dynamic programming algorithm can be stated as follows:

for $z \leftarrow 0$ **to** Z **do** $P(z,1) \leftarrow$ **if** $z = 0 \mod v_1$ **then** 1 **else** 0;
for $i \leftarrow 2$ **to** n **do**
  for $z \leftarrow 0$ **to** Z **do**
  {
    $P(z,i) \leftarrow 0$;
    for $k \leftarrow 0$ **to** $\lfloor z/v_i \rfloor$ **do** $P(z,i) \leftarrow P(z,i) + P(z - kv_i, i-1)$
  }.

Note that if the change-making problem is solved for a certain amount Z, all problems for smaller amounts are solved as well.

The above algorithm can be implemented in a direct way on the Manchester dataflow machine. The computations are performed in some asynchronous feasible order. The parallelism is bounded by the synchronizations due to the fact that in each iteration values of the previous iteration are needed.

To be able to implement the dynamic programming algorithm on an SIMD machine, one has to analyze the parallelism in the program and state the detected parallelism explicitly using the tools the language provides. The parallelism in the dynamic programming algorithm resides in the **for** *z* loops. But to make this parallelism explicit, we must rewrite part of the algorithm.

A problem is the last **for** *z* loop. Whereas one would like to compute *P* for all *z* in parallel, this is impossible on an SIMD machine because the work to be done differs with varying *z* (the number of iterations of the **for** *k* loop depends on the value of *z*). By adding dummy iterations one can achieve that each loop contains the same number of iterations, which is equal to the number of iterations needed for the maximal *z*. A dummy iteration could consist of adding 0. The addition is treated as a sequential process and is not parallelized. Therefore the **for** *k* loop should be interchanged with the **for** *z* loop. The modified program looks like:

for $z \leftarrow 0$ **to** Z **do** $P(z,1) \leftarrow$ **if** $z = 0 \mod v_1$ **then** 1 **else** 0;
for $i \leftarrow 2$ **to** n **do**
{
    for $z \leftarrow 0$ **to** Z **do** $P(z,i) \leftarrow 0$;
    for $k \leftarrow 0$ **to** $\lfloor Z/v_i \rfloor$ **do**
        for $z \leftarrow 0$ **to** Z **do** $P(z,i) \leftarrow P(z,i) + P(z - kv_i, i-1)$
}.

During execution it must be ensured that all references to *P* with negative first index are equal to 0.

The rewritten algorithm can be implemented straightforwardly on the DAP. In doing this we have to view the DAP as a one-dimensional array of processors. Processor *z* computes the values $P(z,i)$ $(i = 1,...,n)$. Each **for** *z* loop is executed in parallel. To be able to compute the sum of the possible combinations, a processor needs the *P* value of the previous iterations of its kv_i-th neighbors $(k = 1,...,\lfloor Z/v_i \rfloor)$. This can be accomplished in parallel by using a DAP-FORTRAN shift routine. Such a routine has the nice property that is shifts in zeros for nonexisting values, making it very easy to add the dummy iterations.

Due to the fact that the DAP has only 4096 processors, the amount *Z* to be paid is limited to 4095.

For the CYBER-205 basically the same procedure can be applied, but instead of being processed in parallel, the **for** *z* loops are now processed in a pipelined (and strictly speaking sequential) manner. The difference with the DAP is that on the CYBER the dummy iterations need not be added because it is possible to input only part of a vector into the pipeline.

3.1.3. Improving Divide and Conquer and Dynamic Programming

Divide and conquer as well as dynamic programming have their pros and cons for solving the change-making problem. Divide and conquer is very easy to program, but the subproblems generated are not mutually exclusive. So, it may happen that solutions to certain subproblems are recomputed. The computation could be sped up if these recomputations could be prevented. Dynamic programming solves all problems $P(z,i)$ $(z = 0,..,Z,\ i = 1,..,n)$ regardless whether or not the solution of a particular problem is needed to construct the solution of problem $P(Z,n)$. The computation could be sped up if there is a way to eliminate subproblems not needed in constructing the solution to $P(Z,n)$.

It is possible to combine the good sides of both methods. The idea is to use divide and conquer to construct the set of subproblems needed and thereafter dynamic programming to solve the problem using only this set of subproblems [Polya, Tarjan & Woods 1983]. This can be realized by adding a mechanism to the divide and conquer approach, which upon request for the solution of a particular subproblem takes the following steps:

- If the subproblem has already been solved, it returns the solution of this subproblem,

- If the subproblem is being solved at the moment, it queues the request for the solution of the subproblem and returns the solution as soon as it is available,

- If the subproblem has not been considered before, it solves this subproblem and stores the solution.

As in the case of the original divide and conquer algorithm we implemented the improved algorithm only on the Manchester dataflow machine. The mechanism could not be written in SISAL due to the fact that its behavior is nonfunctional: given a certain input, the outcome is not completely determined by this input, but also by certain 'environmental' factors. The mechanism was therefore written in TASS, an assembler language. After that, the mechanism was linked to the SISAL program.

3.1.4. Computational Results

The ordering of the coins has consequences for the number of operations to be performed by the divide and conquer algorithm. As mentioned before, the divide and conquer approach is a tree traversal, in which each leaf of the tree must be visited exactly once. So, the work to be done is proportional to the number of edges in the tree. An optimal tree has as few edges as possible. For such a tree, no node has more children than each of its children has. This is realized by splitting each subproblem using the remaining coin with the highest value. Therefore the coins should be ordered by increasing value.

The ordering of the coins has no consequences for the dynamic programming algorithm as long as the coin with the smallest value is used for the initializations. The first coin can be dealt with in $O(1)$ time, whereas the others need $O(Z/v_i)$ iterations for the combination of previous results ($i = 2,...,n$).

In all computational results shown, the ordering of the coins is optimal with respect to the method of solution used. The coinage system used is part of the Dutch system, made up of coins or bank notes of 1, 5, 10, 25, 100, 250, 500 and 1000 cents.

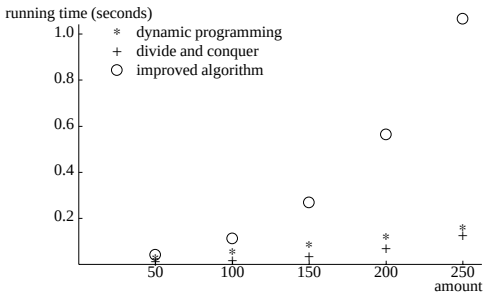


Figure 6. Execution times on the Manchester dataflow machine with 20 processors.

Figure 6 shows some results of the dynamic programming, divide and conquer, and improved algorithms run on the Manchester dataflow machine. Due to a limited memory capacity of the hardware, only small size problems could be solved. The behavior of the programs on the Manchester dataflow machine can be explained from simulations on a sequential computer. These results are shown in Figures 7, 8 and 9. Due to memory restrictions it was impossible to simulate bigger problems.

Figure 7 shows S_1 (the total number of instructions executed) versus the amount to be paid for the various programs. As expected, for small problems the divide and conquer program executes less instructions than the other two programs. But this reverses when the problem size increases. By increasing problem size, the improved algorithm executes less steps than divide and conquer but more steps than dynamic programming. The first is easily explained by the elimination of duplications. The second can only be explained if determining the state of a subproblem is more expensive than computing everything, needed or not.

Figure 8 shows S_∞ (the total number of time steps needed if there was an unlimited number of processing elements) versus the amount to be paid. The S_∞ of divide and conquer and of dynamic programming behave in the same way and differ by a constant. Both programs compute the solution by combining

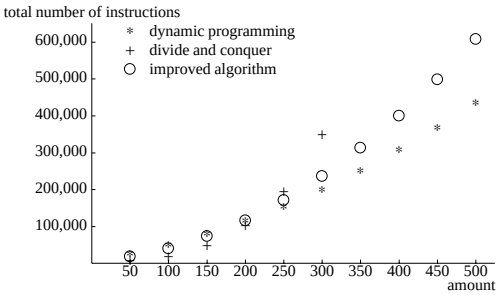


Figure 7. Total number of instructions on the dataflow machine.

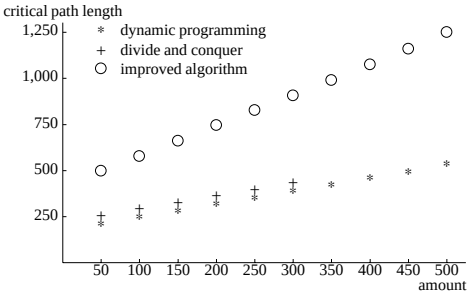


Figure 8. Critical path length on the dataflow machine.

the solutions of subproblems. Since both use the same recursive formula, their S_∞'s have the same behavior. The difference is due to the work involved in the recursion. Because the recursion has always the same depth, the difference is a constant. The S_∞ of the improved program is larger. Determining the state of a subproblem appears to be a time consuming affair. Besides that, requests for the same subproblem have to be handled sequentially.

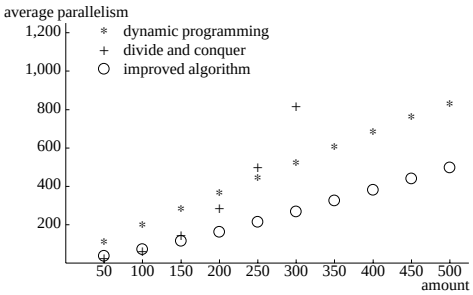


Figure 9. Average parallelism on the dataflow machine.

Figure 9 shows the average parallelism π versus the amount to be paid. Divide and conquer shows an explosion in parallelism with increasing problem size. This is because the subproblems generated are not mutually exclusive. If problem size increases, computing power is lost in solving an ever-increasing number of the same subproblems in parallel. As expected, the average parallelism of the improved program is less than the average parallelism of dynamic programming. This is due to the sequential part of the mechanism which determines the state of a subproblem and to the fact that the solution of a problem must temporary halt if one of its subproblems is being solved at the moment.

We conclude that, in the test environment under consideration, it is not worthwhile to be clever. It is much cheaper to compute everything.

Figure 10 shows our results on the execution of dynamic programming on the DAP and CYBER-205. For the size of problems we considered, the execution time on the DAP is linear. This execution time depends only on the number of subproblems to be combined. Taking the combinations can be performed in

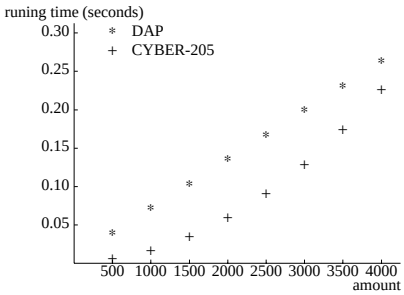


Figure 10. Dynamic programming on the DAP and CYBER-205.

parallel and thus in constant time. The execution time on the DAP behaves in the same way as the critical path length of dynamic programming on the dataflow machine (Figure 8), because in the dataflow simulator we assume an unlimited number of processing elements for taking the combinations. As can be seen, the execution time on the CYBER-205 increases more than linear. This curve corresponds to the total number of instructions performed by the dataflow implementation (Figure 7).

3.2. Shortest Paths

Given a complete directed graph with vertex set $\{1,...,n\}$ and a length c_{ij} for each arc (i,j), one wishes to find the shortest path lengths for all pairs of vertices. We solved this problem using the algorithms due to Dijkstra and Floyd & Warshall.

3.2.1. **Dijkstra** [Dijkstra 1959]

Dijkstra's algorithm solves the one-to-all shortest paths problem in the case of nonnegative arc lengths. The nonnegativity of the lengths ensures that the total length of a path, when it is extended, cannot decrease. Therefore, it is possible to determine in the lth iteration of the algorithm the vertex lth closest to the origin. Denoting the origin by i^*, we have the algorithm:

$N \leftarrow \{1,...,n\} \setminus \{i^*\}$;
for all $j \in N$ **do** $d_j \leftarrow c_{i^*j}$; $d_{i^*} \leftarrow 0$;
for $l \leftarrow 2$ **to** n **do**
{
  $j^* \leftarrow \min\{j \mid (d_j = \min\{d_k \mid k \in N\})\ (j \in N)\}$;
  $N \leftarrow N \setminus \{j^*\}$;
  for all $j \in N$ **do** $d_j \leftarrow \min\{d_j, d_{j^*} + c_{j^*j}\}$
}.

In order to find all shortest paths, all vertices have to be considered as origin in turn. This can obviously be done in parallel.

On the DAP this algorithm is implemented using vector instructions, where 64 processors take care of one vertex. If each processor would do the computations for one vertex, it would be possible to solve problems of size up to 4096. The memory capacity of a single processor is, however, limited to a few hundred numbers. Therefore, only relatively small size problems fit into the DAP and a great part of the processors would be idle. Vector instructions then give a far better performance. As a consequence, we considered problems of size up to 64 only. The **for** l loops are treated sequentially and within a step the operations are performed in parallel. DAP-FORTRAN provides an (assembler) function which can compute the minimum of a vector using parallelism. The processors have to communicate with each other for finding the vertex with the next shortest distance from the origin. The number of this vertex and its corresponding distance have to be broadcasted to all other processors. The '**for all** $j \in N$' instructions are executed for all $j \in \{1,...,n\}$ in parallel; with the use of a mask, which keeps track of the set N, only the relevant updates are performed. Since the computations are done in parallel and idle processors cannot do any useful work meanwhile, this is not a waste of computing power.

The CYBER-205 implementation is straightforward. The initialization and the instructions within the iterative loop can be pipelined. The language provides an (assembler) routine able to compute the minimum of a vector using the pipeline and the conditional instructions are performed using masks.

We implemented Dijkstra's algorithm in two different ways on the Manchester dataflow machine. The first implementation closely resembles the SIMD implementation: a mask indicates the vertices to which the shortest distance still has to be computed, and depending on the value of the mask, the results obtained are stored. In the second implementation, a list of vertices belonging to the set N is maintained. The operations are only performed on elements of this list; only values that are needed are computed. The first way is very easy to implement but has the disadvantage that computing power is wasted on vertices to which the shortest path is already known; the second way is harder to implement but does not waste computing power. Since the Manchester dataflow machine is an MIMD computer, processors that do unnecessary work could perform other available tasks, thus achieving a better overall performance. This is in contrast to an SIMD machine, where the overall performance is not influenced if some of the processors are silenced by a mask. Computations are performed asynchronously and in parallel wherever possible. In each iteration, however, the computations are synchronized on the point where the minimum value has to be computed. Updating the distances cannot be started unless the next shortest distance is known.

Both DAP and CYBER-205 FORTRAN provide an instruction for finding the index of an array element with minimum value. The SISAL language has a serious drawback in this respect: first the minimum value must be obtained and then the corresponding index can be found.

3.2.2. Floyd-Warshall [Floyd 1962; Warshall 1962]

The algorithm due to Floyd and Warshall computes the shortest path lengths for all pairs of vertices simultaneously. The arc lengths do not have to be nonnegative and the occurrence of negative length cycles is detected. At the lth iteration, the shortest paths for all pairs of vertices are computed with intermediate vertices from the set $\{1,...,l\}$. The algorithm is as follows:

for $j \leftarrow 1$ **to** n **do for** $i \leftarrow 1$ **to** n **do** $d_{ij} \leftarrow c_{ij}$;
for $l \leftarrow 1$ **to** n **do**
  for $j \leftarrow 1$ **to** n **do for** $i \leftarrow 1$ **to** n **do** $d_{ij} \leftarrow \min\{d_{ij}, d_{il}+d_{lj}\}$.

On the DAP processor (i,j) computes the length of a shortest path from vertex i to vertex j. At the lth iteration, processor (i,j) needs the current shortest distances computed by processor (i,l) and processor (l,j). This is achieved by broadcasting the lth column of the distance matrix rowwise and the lth row columnwise. This implementation restricts the problem size to 64. Bigger problems can be solved in this way by assigning more pairs of vertices to one processor.

On the CYBER-205, the initializing loops and the last **for** i loop of the algorithm are pipelined.

The Manchester dataflow machine will perform the algorithm in some arbitrary feasible order. Therefore, it might happen that values of different iterations are computed at the same time.

3.2.3. Computational Results

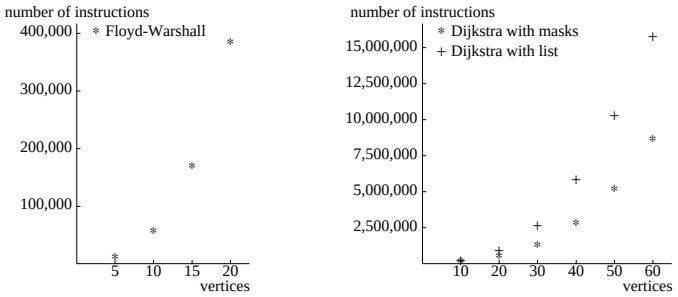
On all machines we solved problems of size n up to 60, with distances drawn uniformly from [1,1000]. For each size, we generated 3 instances. The entries in the Figures represent mean values. Dijkstra's algorithm is applied with all vertices as origin to make the results comparable to those of the Floyd-Warshall algorithm. On the DAP and CYBER-205 this has to be done sequentially, but on the Manchester dataflow machine simultaneous computation is possible.

On the DAP, Floyd-Warshall shows a linear behavior and Dijkstra a quadratic one due to the fact that the basic routine has to be applied n times in sequence. At problem sizes which are a multiple of 64, a jump in the computing times will occur, after which the linear and quadratic behavior will continue. At those discontinuities, vectors and matrices outgrow their maximum size 64 and have to be split at the expense of longer computing times. On the CYBER-205, both algorithms have a cubic behavior, as on any sequential computer, but the solution times for these small problems are about 10 times shorter than on a

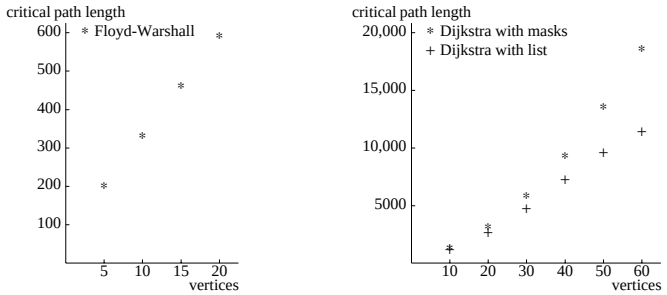
number vertices	Floyd-Warshall			Dijkstra		
	DAP	CYBER 205	CYBER 170-175	DAP	CYBER 205	CYBER 170-750
10	0.0025	0.001	0.002	0.021	0.001	0.002
20	0.0049	0.003	0.018	0.059	0.004	0.019
30	0.0073	0.007	0.057	0.124	0.010	0.058
40	0.0097	0.013	0.114	0.201	0.020	0.147
50	0.0121	0.022	0.215	0.311	0.034	0.271
60	0.0145	0.035	0.363	0.444	0.052	0.478

Figure 11. Running times (in seconds) on the DAP, CYBER-205 and CYBER-170-750.

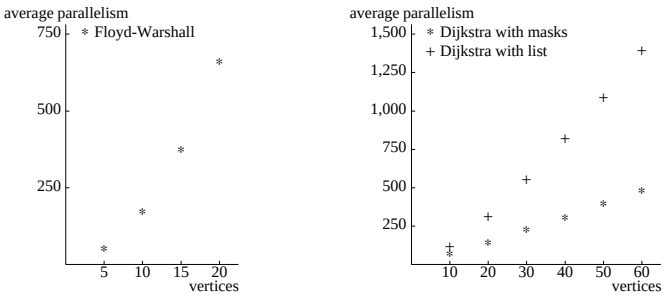
CYBER-170-750. Dijkstra's algorithm has a worse performance than Floyd-Warshall's. See Figure 11.



(a) Total number of instructions.



(b) Critical path length.



(c) Average parallelism.

Figure 12. Performance on the dataflow machine.

The simulator of the Manchester dataflow machine gives a linear behavior of S_∞ for the Floyd-Warshall algorithm. Due to the limited capacity of the matching store, the biggest problem we could handle was of size 20. Both versions of Dijkstra's algorithm have a nonlinear critical path length. This is because

at each iteration a minimum has to be computed which takes $O(\log n)$ time in parallel. S_∞ is larger for the version using masks than for the one doing no useless work. For the total number of instructions performed and the overall parallelism it is the other way around. On a machine with a limited number of processors the former version will perform better, and on a powerful machine (or the simulator) the latter is to be preferred. Cf. Figure 12.

3.3. The Knapsack Problem

Given n items, each with a profit c_j and a nonnegative weight a_j ($j=1,...,n$), and given a knapsack with capacity b, one wishes to find a subset of the items of maximum total profit and of total weight no more than b. This can be formulated as an integer linear programming model of the following form:

maximize $\sum_{j=1}^{n} c_j x_j$

subject to

$$\sum_{j=1}^{n} a_j x_j \le b,$$

$$x_j \in \{0,1\} \ (j=1,...,n).$$

The problem is *NP*-hard [Garey & Johnson 1979]. We consider two types of implicit enumeration: dynamic programming and branch and bound.

3.3.1. Dynamic Programming

We introduce the notation $C(j,z) = \max_{S \subseteq \{1,...,j\}} \{\Sigma_{k \in S} c_k \mid \Sigma_{k \in S} a_k \le z\}$. Using the optimality principle of dynamic programming, one attains the maximum profit $C(j,z)$ either by excluding item j and taking the profit $C(j-1,z)$ or by including item j and adding c_j to the profit $C(j-1,z-a_j)$. By recursively applying this idea, we get the following algorithm [Bellman 1957]:

for $z \leftarrow 0$ **to** b **do** $C(0,z) \leftarrow 0$;
for $j \leftarrow 1$ **to** n **do**
{
  for $z \leftarrow 0$ **to** $a_j - 1$ **do** $C(j,z) \leftarrow C(j-1,z)$;
  for $z \leftarrow a_j$ **to** b **do** $C(j,z) \leftarrow \max\{C(j-1,z), C(j-1,z-a_j)+c_j\}$
}.

On the DAP, the obvious implementation is to compute the values $C(j,z)$ for $z=0,..,b$ in parallel and for $j=1,...,n$ in sequence, where processor z computes the values $C(1,z)$, $C(2,z)$,...,$C(n,z)$. Here, the DAP is considered as an one-dimensional array of processors. In iteration j, a processor needs its own C-value, that of its a_jth neighbor, and c_j. Using a DAP-FORTRAN shift routine, this is accomplished for all processors in parallel. Because the shift routines fill in zeros for non-existing values, all states z can be dealt with in the same way. In this way, we get an $O(n)$ algorithm and a speedup of $O(b)$, provided b is no greater than 4095.

For the CYBER-205 basically the same procedure can be applied, although the parallel instructions are performed sequentially and a data shift is unnecessary. In the jth iteration, not all values $C(j,z)$ have to be evaluated explicitly. For all z with $\Sigma_{k \in \{1,...,j\}} a_k \le z \le b$, all considered items fit together in the knapsack and hence $C(j,z) = \Sigma_{k \in \{1,...,j\}} c_k$. In terms of the algorithm: in each iteration it is sufficient to compute the C-values up to the sum of the weights of the items considered. On a truly parallel computer (with enough processors), this observation would make no difference, but depending on the problem at hand it can lead to substantial savings on the sequential CYBER-205.

A SISAL version of Bellman's algorithm has been run on the Manchester dataflow machine. Since the computation is completely asynchronous, it might be possible that values of different iterations are evaluated at the same time, but a speedup of $O(b)$ remains best achievable.

3.3.2. Branch and Bound

Branch and bound methods generate search trees in which each node has to deal with a subset of the feasible solution set. In the case that the objective function has to be maximized, at each node an upper bound on the optimal value of that node is computed. If at a node the upper bound is no greater than the best overall solution found so far, this node cannot produce a better solution and can therefore be discarded from further examination. Otherwise, a node is split in such a way that smaller subsets of feasible solutions can be considered separately while no feasible solutions are eliminated.

For the rest of this section we assume that the items have been ordered according to nonincreasing c/a.

For the knapsack problem we can derive an upper bound by relaxing the integrality constraints $x_j \in \{0,1\}$ to $0 \le x_j \le 1$ ($j=1,...,n$). This linear-programming-relaxation can be solved efficiently by a greedy algorithm and in the solution at most one variable will be fractional. Setting this variable to zero provides a feasible $\{0,1\}$-solution, which can be used for bounding the search tree. A node will be split by fixing variables to 0 or 1. Suppose, a node has the first k variables fixed (denoted by $\tilde{x}_1,...,\tilde{x}_k$), then we generate the subproblems $\{\tilde{x}_1,...,\tilde{x}_k,1,free,...,free\}$, $\{\tilde{x}_1,...,\tilde{x}_k,0,1,free,...,free\}$, $\{\tilde{x}_1,...,\tilde{x}_k,0,0,1,free,...,free\}$,..., $\{\tilde{x}_1,...,\tilde{x}_k,0,0,0,0,...,0\}$.

Since the evaluation of a node is hardly parallelizable and the DAP is an SIMD-type computer, the parallelism has to be exploited at the level of parallel evaluation of various nodes. By assigning each node to a different processor, at most 4096 nodes can be handled at the same time. In cases of branch and bound where the work to be done within a node very much depends on that node, the SIMD-restriction of the DAP becomes a severe problem. Since the LP-relaxation of the knapsack problem can be solved in a regular way by a greedy algorithm, all nodes can be dealt with concurrently. However, all processors have to perform the same operation on data residing in the same place of their local memories. Therefore, specific information on a particular node cannot be taken into account satisfactorily. For example, fixed variables at one node may be free variables at another and the only way an SIMD machine can take care of this is by letting all processors look at all variables. Each time the nodes are split, the work has to be redistributed over the processors. If at any time more than 4096 nodes exist, a priority queue is needed and each time the 4096 'best' nodes are evaluated. In our situation, we chose for a lexicographical enumeration scheme, i.e., a parallel depth first process. The priority queue is maintained by all processors concurrently, but involves a lot of work.

On the CYBER-205, the same implementation will work. Here, the newly generated nodes have to be composed to a vector in order to use the pipeline and a priority queue is necessary if the vector length exceeds 65535.

On the Manchester dataflow machine, we would like to have a completely asynchronous implementation of the algorithm. The MIMD-type parallelism allows for efficient implementation of the computation of upper and lower bounds. To kill subproblems that cannot yield the optimal solution, at each time the best feasible solution found so far has to be known by all subproblems under consideration. Since in SISAL, because of the single assignment rule, no global updatable variables exist, the only way to accomplish this within the language is by synchronizing the subproblem examinations after the computation of the lower bounds. But, synchronization means waste of computing power as processes have to wait for each other. Therefore we used the same assembler routine as in the improved divide and conquer algorithm for simulating a global memory that contains the best overall feasible solution.

3.3.3. Computational Results

For the DAP and CYBER-205, we generated three types of problems. In type 1 the profits and weights are drawn uniformly from [1,64]. To get types 2 and 3, we added 512 and 1024 to both the profits and the weights. For all three types we considered an instance with $n =$ 100, 200 and 300; for dynamic

programming *b* equals 4095, which is the largest problem size we can solve on the DAP without partitioning the program, and for branch and bound *b* equals 4200. From type 1 to 3 the knapsack problems are harder to solve by means of branch and bound methods. This comes from the empirical fact that, in general, they are more difficult if the number of items that fit into the knapsack is smaller and the profit/weight-values are varying less.

n	type	DAP	CYBER-205	CYBER-170-750
100	1	0.019	0.011	0.257
100	2	0.019	0.022	0.420
100	3	0.019	0.019	0.359
200	1	0.038	0.036	0.832
200	2	0.038	0.045	0.828
200	3	0.038	0.039	0.704
300	1	0.058	0.062	1.373
300	2	0.058	0.067	1.238
300	3	0.058	0.059	1.047

Figure 13. Running times (in seconds) of dynamic programming on the DAP, CYBER-205 and CYBER-170-750 ($b = 4095$).

Dynamic programming gives more or less expected results on the DAP. The estimated CPU time grows linear with *n*, but there is no distinction for the different types. Since the distance which data have to travel increases with increasing type numbers, one expects an increasing computing time. The only information which can be retrieved from the DAP, however, is the number of instructions performed and that number is the same for all types of problems. The CYBER-205 computing times display the sequential nature of this machine. The running times are 20 times better than on the CYBER-170-750. Cf. Figure 13.

n	type	DAP	CYBER-205	CYBER-170-750
100	1	0.2	0.01	0.01
100	2	3.0	0.07	0.01
100	3	5.0	1.78	0.25
200	1	5.0	0.12	0.03
200	2	18.0	3.51	0.10
200	3	38.0	35.54	2.16
300	1	11.0	0.36	0.06
300	2	-	-	-
300	3	-	-	-

Figure 14. Running times (in seconds) of branch and bound on the DAP, CYBER-205 and CYBER-170-750 ($b = 4200$).

Branch and bound turns out to be inefficient on both the DAP and CYBER-205 (see Figure 14). The search trees for the type 1 problems are narrow. This implies for the DAP that only a small part of the processors is doing useful work and for the CYBER-205 that the vector lengths are small. For the type 2 and 3 problems, the search trees are very broad. This ensures an economic use of the DAP processors and the CYBER-205 pipeline. But here the amount of work to redistribute the subproblems over the processors on the DAP and to rearrange the subproblems into a vector on the CYBER-205 is enormous. This part of the program completely dominates the computation of lower and upper bounds. For these reasons the traditional CYBER-170-750 performs better than the DAP and CYBER-205.

On the Manchester dataflow machine, we only could run some very small problem instances. The profits and weights are drawn from [1,100]. We generated problems with n = 10, 20, 30 and 40 and b = 100, 200 and 300.

n	$b=100$	$b=200$	$b=300$
10	418	431	437
20	756	765	784
30	1091	1109	1122
40	1443	1466	1479

(a) Critical path length.

n	$b=100$	$b=200$	$b=300$
10	30	70	106
20	37	85	128
30	39	89	135
40	41	89	133

(b) Average parallelism.

Figure 15. Dynamic programming on the dataflow machine.

Dynamic programming shows an S_∞ linear in the problem size n and a parallelism growing with b. With growing b more elements fit into the knapsack. This explains an increasing S_∞ for constant n. Cf. Figure 15.

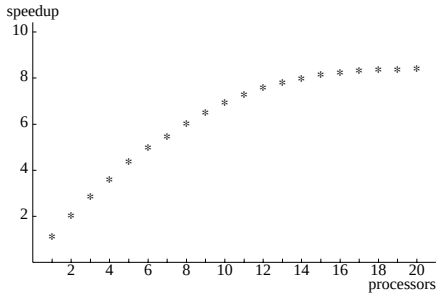


Figure 16. Typical speedup curve for dynamic programming on the dataflow machine; $n=40$ and $b=300$.

For the problem instances considered, the hardware results are comparable: for less than 10 processors the speedup (the running time of the algorithm using one processor divided by the running time of the algorithm using p processors) increases almost linear, after that hardly any gain is made (Figure 16).

n	$b=100$	$b=200$	$b=300$
10	892	1226	750
20	1219	2300	1394
30	1287	2735	1767
40	4518	3407	5468

(a) Critical path length.

n	$b=100$	$b=200$	$b=300$
10	9	8	9
20	14	21	12
30	19	21	15
40	48	24	77

(b) Average parallelism.

Figure 17. Branch and bound on the dataflow machine.

Branch and bound results look promising. The S_∞ and π correspond to the depth and the width of the search tree; see Figure 17. Because communication is cheap and the parallelism is fine grained, no time is lost in the assignment of tasks to processors. Therefore, it can be expected that problem instances for which broad search trees are needed can be solved efficiently on this sort of machines.

4. Conclusions

The ICL-DAP and CDC-CYBER-205 are very well suited for performing regular and relatively simple computations in a fast way, due to their SIMD-type parallelism. They turn out to be inefficient if the behavior of the algorithm is irregular or cannot be predicted in advance. The problems solvable on the Manchester dataflow machine are still very small. However, the computational results give a strong indication that this machine seems to capture all sorts of parallelism. In Manchester, research is going on and one of the aims is an improvement of the matching store which is the bottleneck in the present configuration. Only after that, it will be clear how a dataflow computer will behave on more realistic problems.

Acknowledgements
We would like to thank the DAP Support Unit of Queen Mary College, London, and the Manchester Dataflow Group for their assistance and for making their equipment available to us, and J.K. Lenstra for his help during the preparation of this paper.

References

W.B. Ackerman (1982). Data flow languages. *IEEE Computer 15(2)*, 15-25.

R.E. Bellman (1957). *Dynamic Programming*, Princeton University Press, Princeton, NJ.

CDC (1983). *FORTRAN 200 Version 1*, Reference Manual 60480200, CDC Sunnyvale, CA.

J. Darlington, M. Reeve (1981). ALICE - a multi-processor reduction machine for the parallel evaluation of applicative languages. *ACM Proc. 1981 Conf. Funct. Progr. Lang. and Comput. Architecture*, 65-75.

E.W. Dijkstra (1959). A note on two problems in connexion with graphs. *Numer. Math. 1*, 269-271.

R.W. Floyd (1962). Algorithm 97: shortest path. *Comm. ACM 5*, 345.

M.J. Flynn (1966). Very high-speed computing systems. *Proc. IEEE 54*, 1901-1909.

M.R. Garey, D.S. Johnson (1979). *Computers and Intractability: a Guide to the Theory of NP-Completeness*, Freeman, San Francisco.

J.R. Gurd, C.C. Kirkham, I. Watson (1985). The Manchester prototype dataflow computer. *Comm. ACM 28*, 34-52.

ICL (1979). *DAP: FORTRAN Language*, Technical Publication 6918, ICL, London.

ICL (1981). *DAP: Developing DAP Programs*, Technical Publication 6920, ICL, London.

J. McGraw, S. Skedzielewski, S. Allan, D. Grit, R. Oldehoeft, J. Glauert, C. Kirkham, B. Noyce (1984). *SISAL: Streams and Iteration in a Single Assignment Language*, Language Reference Manual Version 1.2, Lawrence Livermore National Laboratory, Livermore, CA.

G. Polya, R.E. Tarjan, D.R. Woods (1983). *Notes on Introductory Combinatorics*, Birkhauser, Boston.

P.C. Treleaven, D.R. Brownbridge, R.P Hopkins (1982). Data-driven and demand-driven computer architecture. *Comput. Surveys 14*, 93-143.

S. Warshall (1962). A theorem on boolean matrices. *J. Assoc. Comput. Mach. 9*, 11-12.

I. Watson (1984). The dataflow approach - architecture and performance. F.B. Chambers, D.A. Duce, G.P. Jones (eds.). *Distributed Computing*, Academic Press, London, Ch. 2.