Massively Parallel Augmenting Path Algorithms for the Assignment Problem *

S. Storøy — sverre@ii.uib.no

T. Sørevik — tors@ii.uib.no

submitted May 1994,
revised March 1996

Abstract

In this paper we describe how to apply fine grain parallelism to augmenting path algorithms for the dense linear assignment problem. We prove by doing that the technique we suggest, can be efficiently implemented on commercial available, massively parallel computers. Using n processors, our method reduces the computational complexity from the sequential $O(n^3)$ to the parallel complexity of $O(n^2)$.

Exhaustive experiments are performed on a Maspar MP-2 in order to determine which of the algorithmic flavors that fits best onto this kind of architecture.

AMS Subject Classifications: 90C08, 65E10

Keywords: Parallel algorithms, matching in bipartite graphs, the assignment problem.

1 Introduction

The supercomputers of tomorrow will be parallel computers. Consequently any computational technique for large scale problems has to address the problems of how to parallelize our algorithm. Given the diversity of parallel computers, we also have to identify the most suitable hardware platform.

In this paper we investigate a strategy for fine grain parallelism of the Linear Assignment Problem (LAP). Being one of the simplest network flow problems this

*Dept. of Informatics, University of Bergen,
Thormøhlensgt. 55, 5020 Bergen, Norway.

problem has served as a useful test bed for new computational ideas, which later have been generalized to other areas of network flow. The classical example is Kuhn's [Kuh55] Hungarian method from which the general primal-dual algorithm was developed. It therefore seems as a good starting point for the investigation of massively parallel network flow algorithms.

Another reason for parallelizing the LAP is its use as the workhorse in computational intensive problems such as the traveling salesman problem [PS82] and the nonlinear assignment problem [Law76]. Efficient parallel programs for the LAP will be essential in solving these problems.

The LAP may be formulated as the following linear programming problem:

$$\begin{array}{rl} \min & \sum_{i,j} c_{ij} x_{ij} \\ \text{subject to} & \\ & \begin{array}{rcll} \sum_j x_{ij} & = & 1 & (i = 1, ..., n) \\ \sum_i x_{ij} & = & 1 & (j = 1, ..., n) \\ x_{ij} & \geq & 0 & (i, j = 1, ..., n) \end{array} \end{array}$$

The dual problem is:

$$\begin{array}{rl} \max & \sum_i u_i + \sum_j v_j \\ \text{subject to} & \\ & \begin{array}{rcll} c_{ij} - u_i - v_j & \geq & 0 & (i, j = 1, ..., n) \end{array} \end{array}$$

Alternatively, we may describe the LAP as a minimum cost perfect matching problem. Let $G = (S \cup T, C)$ be a bipartite graph, where $S = \{s_1, ..., s_n\}$ and $T = \{t_1, ..., t_n\}$ are the sets of nodes, $C = \{(s_i, t_j) : s_i \epsilon S, t_j \epsilon T\}$ is the set of arcs and c_{ij} the cost associated with arc $(s_i, t_j) \epsilon C$.

The solution of the minimum cost perfect matching of G is the solution of LAP (and vice versa). Arc (s_i, t_j) being present in the optimal matching, corresponds to the decision variable $x_{ij} = 1$ in the optimal LP-solution.

There also exists an iterative scheme called the auction algorithm due to Bertsekas [Ber81], [Ber88]. This method has attracted a lot of interest lately because it lends itself naturally to parallel implementation. In fact, most studies of parallel methods for the assignment problem consider this method.

In this paper we concentrate on the graph formulation of the problem, which can be used to device either max-flow algorithm or shortest augmenting path (SAP) algorithms. Numerous implementations of these exist and the fastest sequential methods, used in practice, all have a worst case complexity $O(n^3)$. The performance on experimental data does, however, show great differences in practical performance for the different strategies. The differences depend on the data as well as implementational details. It does however seem fair to say that good implementations of SAP-algorithms or the Hungarian method outperform the auction algorithm on sequential computers. Experiments on shared memory computers with a moderate number of processors have, however, revealed some problems in obtaining high speed-up when porting codes for these methods to shared memory multiprocessors system [Zak90], [KW91]. Zaki [Zak90] found that the better vector and concurrent capability of the auction algorithm made it superior to the SAP algorithm on an Alliant FX/8 with 8 vector processors.

Implementations for massively parallel SIMD computers also exist. Wein and Zenios [WZ90] implemented the auction algorithm on a 32K processor CM-2, while Brady. & al. [BJN+91] have implemented Munkre's algorithm [Mun57], an $O(n^4)$ version of Kunh's [Kuh55] 'Hungarian Method', on a 16K WaveTracer and on an 'in-house' constructed machine, AP2S, with approximately 10K processors. Castanon e. al [CSW89] have implemented the auction algorithm as well as Jonker and Volgenant's [JV87] augmenting path algorithm on the CM-2 and the DAP 510.

The purpose of this paper is twofold. First we show how a quite simple technique for fine grained parallelization may be applied to this sort of algorithms. We discuss which kind of hardware this technique requires in order to be implemented efficiently. The MasPar MP-1 and MP-2 satisfy these requirements and we have implemented our algorithms on this platform. Computational results show that our parallel programs run very efficient on the MasPar computers. Our approach has strong similarity with the technique used by Castanon & al [CSW89] and Brady & al. [BJN+91]. Brady & al. do, however, choose to implement an $O(n^4)$ algorithm, and as a result their parallel implementation performs very poorly.

Secondly we apply our technique to two well documented and efficient sequential codes, [JV87] and [CMT88]. The two codes behave very different depending on the input data. We report experimental results in order to determine which of the different algorithmic flavors that works best in our parallel implementation.

To provide the necessary background we outline the algorithms in section 2. In section 3 we describe how to parallelize these and discuss the hardware requirements our technique implies. Then we report our computational experiments in section 4 and comment on the results.

2 Sequential algorithms

In this section we mainly follow the description given in Carpaneto & al. [CMT88] of the Hungarian algorithm.

All algorithms discussed in this paper are based on the following two properties related to the cost matrix:

P1 Any constant may be added to or subtracted from each element in a row or a column of the cost matrix without affecting the optimal solution.

P2 If the cost matrix contains only nonnegative values, then if a solution of zero cost is feasible, this solution is optimal.

The algorithms now define the reduced cost matrix as

$$c^*_{ij} = c_{ij} - u_i - v_j, \tag{1}$$

where u_i and v_j are the dual variables, and try to reach property P2 above by modifying the dual variables if necessary. Throughout the process dual feasibility is maintained, i.e. $c^*_{ij} \geq 0$, while, in each step, the primal infeasibility is reduced, until a primal feasible solution is found.

All algorithms of this paper fit in to the following general framework:

```
algorithm LAP;
begin
    initialize; (* compute initial values of dual variables
                   and find a dual feasible assignment *)
    repeat
        select an unassigned row, i;
        findpath; (* find an augmenting path from i to unassigned
                     column j. If necessary update dual variables *)
        reassign; (* reorganize assignment to also include i and j *)
    until (all rows are assigned);
end.
```

There are numerous ways of obtaining an initial assignment as well as constructing augmenting paths. In order to highlight the problems and possibilities for parallelization, we will give a detailed description of a simple initialization routine and CMT's augmenting path routine (later called 'naiv-init' and 'CMT-aug', respectively), and explain how to parallelize these. With obvious modifications the techniques carry over to the more sophisticated variants we have implemented.

For our computational experiments, reported in sec. 4, we concentrate on two of the most ingenious and well documented sequential ones: Carpaneto & al.[CMT88] and Jonker & Volgenant [JV87].

The following two vectors of indices are used:

f : f_i = column assigned to row i; $(i = 1, ..., n)$
$\bar{f}$: $\bar{f}_j$ = row assigned to column j; $(j = 1, ..., n)$

```
procedure initialize;
f := f̄ := 0;
for j := 1 to n do (* column reduction *)
  find r and c_rj such that c_rj = min_i{c_ij};
  v_j := c_rj;
  if (f_r = 0) then
    f̄_j := r;
```

```
        f_r := j;
        u_r := 0;
    endif;
enddo;
for i := 1 to n do (* row reduction *)
    if (f_i = 0) then
        find c_ir = min_j{c_ij − v_j};
        u_i := c_ir − v_r;
        if (f̄_r = 0) then
            f̄_r := i;
            f_i := r;
        endif;
    endif;
enddo;
end(initialize);
```

The procedure *findpath* below is searching for a path of zero reduced cost starting from the unassigned row i. If no such path is found, the dual variables are updated and the search continues. Finally, when a path to an unassigned column, j, is found, the procedure stops. We use the following sets and vectors:

LR - set of labeled rows,
UC - set of unlabeled columns,
ptr - vector of row indices, ptr_j = row preceding column j in the current path
$temp$ - vector of reduced costs, $temp_j = \min\ \{c^*_{ij} : i \epsilon LR, i \neq \bar{f}_j\}$.

```
procedure findpath;
LR := {i};  UC := {1, ..., n};
forall j∈{1, ..., n} do temp_j := ∞;
repeat
        r : = next element in LR;
        forall j∈UC do
                if (c_rj − u_r − v_j < temp_j) then
                    temp_j := c_rj − u_r − v_j;
                    ptr_j := r;
                endif;
        enddo;
        if ({temp_j = 0 : j∈UC} = ∅) then
        (* update dual variables *)
                d := min(temp_j  : j ∈ UC);
                forall k∈LR do u_k := u_k + d ;
```

```
        forall j∈{1,...,n} do
          if temp_j = 0 then
            v_j := v_j − d;
          else
            temp_j := temp_j − d;
          endif;
        enddo;
    endif;
    j : = first element in {temp_l = 0 : l∈UC};
    if (f̄_j > 0) then
        LR := LR ∪ {f̄_j};
        UC := UC\{j};
    endif;
until (f̄_j = 0);
end(findpath);
```

Finally the procedure *reassign* is reorganizing the assignment such that row i and column j are being assigned too. The *ptr*-vector is used to backtrack the augmenting path which determines the appropriate update of f and $\bar{f}$.

```
procedure reassign;
while(i ≠ ptr_j) do
    l := ptr_j;
    f̄_j := l;
    k := f_l;
    f_l := j;
    j := k;
enddo;
end(reassign);
```

The complexity of the initialization procedure described here is $\Theta(n^2)$. Since the while loop of the main program and the repeat loop of *findpath* are iterated at most n times, the augmenting path part has complexity $O(n^3)$, giving an overall time complexity for the entire problem of $O(n^3)$.

3 Parallelizing the algorithms

The inherently sequential nature of the outer iteration in the algorithms, prohibits a coarse grain parallelizing. Using a matrix as the datastructure for the graph, to examine all edges associated with a vertex is equivalent to operate on a row in the matrix. Then all operations performed in each iteration become simple vector operations, like adding/subtracting a constant to a vector, elementwise comparison of vectors or finding the smallest element of a vector. All vectors have same length, but not necessarily all elements take part in the operations. These have to be masked out.

Thus, if we distribute all vectors across n processors, we may mask out the actual elements and do the appropriate operations simultaneously on all elements. This layout of data, which is similar to the one described by [CSW89], implies that each processor holds a column of the cost matrix.

The kind of hardware needed for these operations is direct and fast communication between one master processor and n slave processors, which operate in an SIMD fashion. The master broadcasts the instructions, the slaves test if they are to participate in the operation and if so, they execute it. Logically, we perform vector operations, but unlike pipelined vector instructions, the operations take place simultaneously on all elements, or a subset of elements. Consequently the execution time is independent of the vector length.

A similar masking technique, working on the bit-representation of the data, might be used for finding the maximum (or minimum) of a vector, distributed across the processors. We mask out the processors with a 0-bit in the most significant place, and ask the remaining processors to send a message to the master processor. If the master receives any message we continue with the next bit and so forth until all processors are idle or all bits are examined and the maximum is equal to zero.

The execution time of this algorithm will be independent of the vector size (n), only depending on the number of bits in the data. Thus for a given datatype this is a constant operation in terms of vectorsize, n.

With this layout of data and these simple algorithmic modifications, we do only a constant number of steps inside the repeat-loop of the *findpath* routine. This loop is repeated at most n times and the routine is called less than n times giving us a worst case, parallel complexity of $O(n^2)$.

The same technique also applies to the second loop of the initialization procedure. In the first loop, however, we are searching for the minimum over columns of the cost matrix. With all elements of a column placed at one processor, this is a job for one processor only. Fortunately, this operation is carried out on all columns, independently for all processors. Thus, it is trivial to modify the code such that all n processors simultaneously find the minimum of their column.

With these modifications the complexity of the initialization is reduced from $O(n^2)$ to $O(n)$.

Note also that the repeat loop of the main program as well as the repeat loop of *findpath* and the while loop of *reassign* are inherently sequential. Each of these loops will in the worst case be iterated n times. Thus, no straightforward parallel version of the algorithm yields a lower complexity than $O(n^2)$.

4 Parallel Implementation and Computational results

4.1 The MasPar computer

We have implemented our algorithm on MasPar's MP-1 and MP-2. These are massively parallel SIMD computers. The configurations we are using each has 16K processors.

The instruction stream is executed by a dedicated processor, the ACU (Array Control Unit), which is directly connected to each of the processors by the ACU-PE bus. The ACU-PE bus may either broadcast an instruction from the ACU to all processors in the processor array (PE-array) or the PE's simultaneously send data to the ACU. In the latter case a logical reduction (a 'global or') is performed. This is the mechanism which is used for finding the maximum element in our algorithm. The ACU also performs sequential operations.

There are two different inter-processor communication mechanisms available on the MasPar computers; the Xnet and the router. The Xnet connects the processors in a 2-d toroidal mesh and provides a very efficient nearest neighbor communication between processors of the mesh. The router is a slower, but more flexible communication mechanism between any two sets of processors. Note, that for this algorithm no inter-processor communication is needed.

On the MP-1 each processor has 16Kb memory and on the MP-2 64Kb. Using 4 Byte integer as data this implies that we are running out of memory before we are short of processors. Paging is possible, but due to the randomlike access of the data, it slows down the performance significantly [Her92]. The individual processor is a 4 bits, register based processor on the MP-1 and 32 bits on the MP-2. The different processors and the different memory sizes are the only significant difference between MP-1 and MP-2. Further information on the MasPar architecture might be found in [Bla90] and [Nic90].

There are two programming languages available, MP-Fortran and MPL. MP-Fortran is almost identical to the newly defined HPF-standard [For93]. In MP-Fortran the compiler takes care of communications and mapping of data onto the processor array. The format the Fortran-compiler uses for mapping matrices onto the processor array is unsuitable for our application. We have therefore written our programs in MPL, MasPar's C-like language with extension for programming the communication. The MPL-language also supports a built-in reduction-function for computing the minimum value in constant time [Chr90]. As documented in [Pre93] the cost of this operation is independent of the number of participating processors and is, on the MP-2, about 12 times the cost of multiplying two integers.

A crucial detail for good performance on the MasPar computers is keeping the most used variables in registers (this is done in our codes). On the MP-1 performance also improves when using short integer format. We have used 16 bits integer to identify processors and 8 bits data to mark whether or not a processor belongs to the set LR or UC.

The results reported in this paper are for the MP-2 only. On the MP-1 all routines performed poorer, but their relative performance are the same. A good approximation for execution times on the MP-1 is obtained by multiplying MP-2 times with 1.5.

4.2 The tested algorithms

For efficient sequential implementation of algorithms for sparse problems, data structures like linked lists can be an efficient tool for speeding up the computation. Using this datastructure would, however, be disastrous to the parallel technique we apply. We therefore only consider problems with dense cost matrix.

For sequential implementations there is substantial evidence to the claim that it pays off to make some extra efforts in the initialization phase in order to save work in the augmentation phase. Whether this is true for parallel implementations depends, of course, on how well these initialization methods parallelize.

To illuminate this we have chosen to implement the simple initialization of section 2, together with Jonker and Volgenant's sophisticated initialization [JV87]. While the column reduction is equal for the two, Jonker and Volgenant employ a far more elaborated and expensive code to replace the row reduction. In the second stage, called reduction transfer, additional reduction of the assigned rows from stage one is performed. The cardinality of the assignment does not change in this stage since only assigned columns are considered, but as assigned columns are made more expensive relative to the unassigned ones the shortest path in the augmentation phase tends to become shorter (an unassigned column is reached earlier), see [JV87] and [Zak90].

The third stage of JV's initialization is performing two cycles of the basic 'auction'-algorithm, due to Bertsekas [Ber81]. A motivation for this choice and evidence for its effect are given in [JV87] and [Zak90].

We have also implemented Carpaneto & al. initialization [CMT88]. This is more of a straightforward extension to the "naive" one. In the row reduction part, if they are not able to assign a row because the columns-candidates are occupied by other rows, they try to reassign any of these rows. This is essential the same as trying to find an augmenting path of length one. Our experiments showed, not surprisingly, that what we gained with this initialization we lost, by approximately the same amount, in the augmenting part.

For the augmenting part phase we have implemented CMT's routine and JV's Shortest Augmenting Path (SAP) routine. The two routines are very similar in structure. They are both described by vector operations and hence lend themselves naturally to our parallel implementation. JV's do some extra effort to find the shortest augmenting path, whether or not this efforts pays off in our setting is one of the questions we like to answer.

The interface between any of the initialization routines and the augmenting path routines are exactly the same. Thus we are free to combine any of the initialization routines with our favorite augmenting path routine. In an earlier version of our codes [SS92] we notice that when multiple assignments of column to row were possible, the order in which the columns and the rows, were traversed, played an important role. When multiple assignments of columns to a row are possible in the initialization, we always choose the leftmost column. This leads to a clustering of the unassigned columns to the rightmost part of the matrix after the initialization. In the second part, while searching for an augmenting path to unassigned columns we reversed this choice, always choosing the rightmost column in case of multiple choices. This lead to great improvement for low cost range and never any worse performance for high cost range. This 'reversed search' is implemented in the code we have tested.

4.3 The experiments

The theoretical complexity bounds found in sec. 3 do only depend on n, the problemsize. It is a well known fact, however, that the empirical behavior of different routines depends also on the distribution of the data. Here we examine the effect of the cost range R, defined as

$$R > \max_{i,j} c_{ij} \tag{2}$$

We have run a number of experiments on uniformly random generated data on the interval $[0, R)$ and done carefully timing on the separate parts of the algorithm. All timings are wall clock time, in seconds, with the system run in single user mode.

To eliminate the random effect each result reported is the average over 10 samples. We have recorded the individual results and found that the differences due to the random numbers are small. Individual results were always within 30 % of average. All programs have used the same random generated data, for given n and R. The combination of the two initialization and two augmenting path procedures give 4 possible variations. Our computational experiments are aimed at determining the most efficient one as a function of the problem size, n, and the cost range, R.

In figure 1 the reduction in complexity of our MasPar code is demonstrated. The slope of the curves in the log-log plot essentially describes the observed behavior of the routines, and it is evident from the figure that our parallel code has a reduced complexity.

In the experiments reported in figure 2 and 3 we keep the problemsize fixed to $n = 250$ while the cost range, R, varies from 10 to 10^5.

The parallel execution time of the 'naive' initialization is a linear function of n, only. This is demonstrated by the computational results reported in figure 3. The quality of the initialization, in terms of numbers of rows assigned, depends however, also on R. On average we expect a row in the reduced cost matrix to have $1 + n/R$ zeroes after column reduction. Having more zeros, the possibility of finding a zero at an unassigned row, increases with decreasing value of R. The same argument applies to the row reduction of the 'naive' initialization. Consequently we expect more rows to be assigned after the 'naive' initialization of a problem with small R

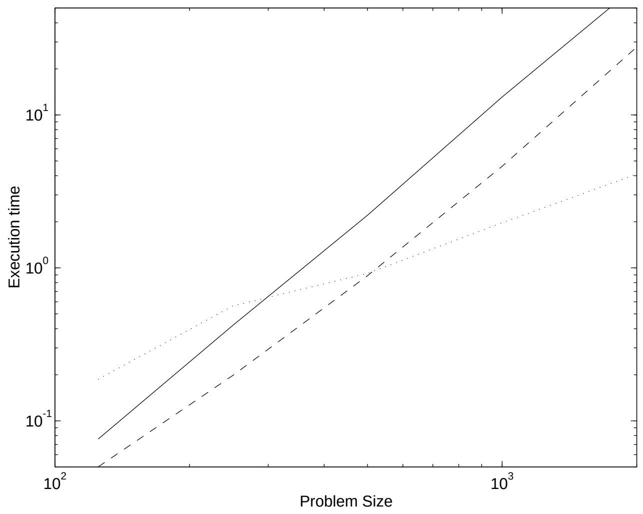

Figure 1: *Comparison of our MasPar code with the sequential code of Carpaneto et. al. run on a Sparc-2 and a Cray Y-MP. Solid line is Sparc-2, dashed line is Cray Y-MP, dotted line is MasPar MP-2.* $R = 100000$

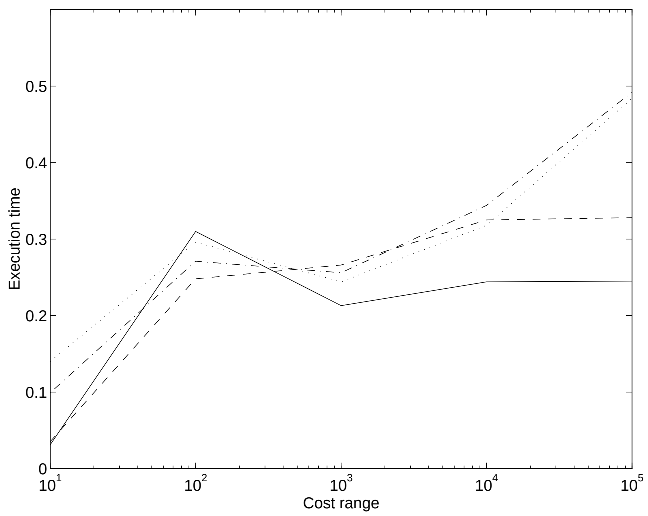

Figure 2: *Performance of the 4 different flavors of algorithm for* $n = 250$ *and varying* R. *Solid line is naiv-init/CMT-aug, dashed line is naiv-init/JV-aug, dotted line is JV-init/CMT-aug and dashdotted line is JV-init/JV-aug*

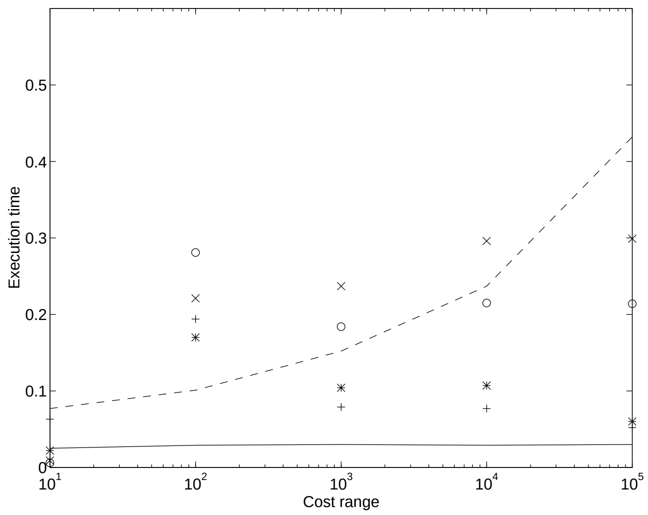

Figure 3: *This figure splits the time for initialization and augmentation. The time used for naive initialization, solid line, and JV initialization, dashed line. CMT augmentation based on naive init is shown by circles, based on JV-init with plusses. JV augmentation based on naive init is shown by x's and based on JV-init with stars. All numbers as a function of R and for $n = 250$.*

than one with equal n and larger R. A more rigorous probabilistic analysis of this is given in [Sør93].

The JV initialization exhibits a very different behavior. The differences occur in the third stage. Here it only considers the two columns with smallest reduced cost. If they are equal we quit, if possible after assigning one of them. If not, the column with smallest reduced cost is assigned to our row, if this column already has an assignment, that assignment is removed, dual variables are updated and the process repeated.

When $R << n$ the typical situation is that many entries have the same minimal reduced cost. Then the third stage of JV-init quits after one iteration, having considered only 2 columns for assignments, while the naive one considers n/R. As a consequence JV-init gives less assignments in this case. On the other extreme while $R >> n$, the iteration process changes the values of dual variables and the iteration is repeated a number of times, and as result an increasing number of rows become assigned. In this situation the iteration process may carry on for quite a while as the termination criterion is that the two smallest reduced costs should be equal or an additional assignment is made. Thus we expect the cost as well as the effect of this third stage to increase with increasing R. This is clearly demonstrated by figure 3. As the cost range increases, the augmentation is faster, regardless of augmentation technique due to the fact that less rows have to be assigned in this phase.

The overall effect of JV initialization is, however, that we loose more time in initialization than we gain in augmentation for this value of n.

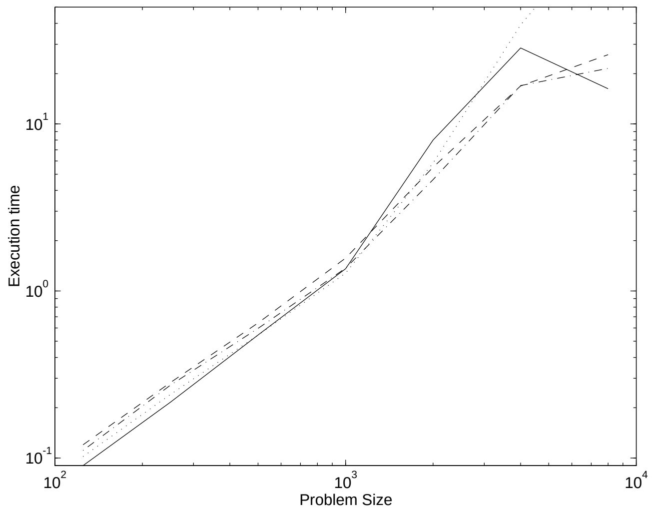

Figure 4: *Performance of the 4 different flavors of algorithm for* $R = 1000$ *and varying* n*. Solid line is naiv-init/CMT-aug, dashed line is naiv-init/JV-aug, dotted line is JV-init/CMT-aug and dashdotted line is JV-init/JV-aug*

The worst case complexity for the JV-init is $O(n^2)$. However, our experiments reveal an average behavior which seems to be surprisingly well approximated by a linear function of n, just as the naive-init. The cost does however increase with R. Thus, in particular for the high cost range, the routines based on JV initialization improve their relative performance as n increase (see figure 5).

For each unassigned node we have to create an augmenting path. The time for creating an augmenting path, we expect to be proportional to the length of the path, while the 'unit cost' of inspecting a node is independent of n as well as R. This is confirmed by our experiments.

The average length of the path we expect to depend on two conditions: i) The number of unassigned nodes/columns and ii) The number of feasible arcs (=nonzeroes in the reduced costmatrix). The first condition simply says that we must expect to work harder to assign the last rows than to assign the first. The second condition implies that low cost range might be an advantage. To what extent it is, depends on how clever our augmenting path routines choose among the feasible nodes. A thorough analysis of the average case running time for random generated data may be found in [Sør93].

For the experiments with different values of n and R we have run, JV's SAP always created a shorter augmenting path than CMT's augmenting path routine (again this is not necessarily true for individual cases, only on average). However, the 'unit cost' is approximately 40 % higher for JV's routine in our implementation, and therefore the total cost is not necessarily less. In particular we find that for $R < n$, when multiple assignments frequently occur, JV's routine more cleverly takes

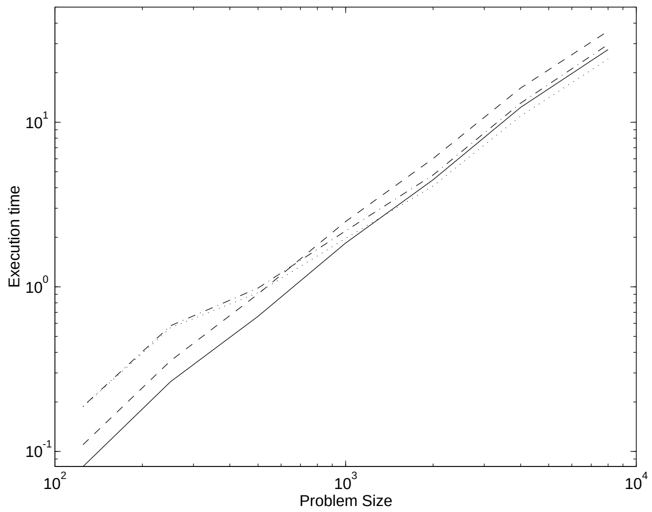

Figure 5: *Performance of the 4 different flavors of algorithm for* $R = 100000$ *and varying* n*. Solid line is naiv-init/CMT-aug, dashed line is naiv-init/JV-aug, dotted line is JV-init/CMT-aug and dashdotted line is JV-init/JV-aug*

advantage of this, and in many cases finds significantly shorter augmenting path, and is in these cases the fastest procedure. In the low cost range there is an interplay between initialization and augmenting path routine. In this range JV-sap is always, by a large margin, the most efficient to take advantage of the extra work done in JV-init. While in case of the 'naive initialization' the 'reversed search', described at the end of sec. 4.2, may make CMT-aug competitive.

The reason for the observed reduction when increasing n from 4000 to 8000 in figure 4 is due to the fact that for fixed $R = 1000$ we are now entering the low cost range regime. Thus a higher percentage of the assignments are done in the initialization and the reversed search makes the augmenting paths shorter.

5 Conclusion

In this paper we have described a technique for parallelizing augmenting path algorithms for the dense linear assignment problem. The complexity reduces from $O(n^3)$ in the sequential case to $O(n^2)$ in the parallel one, using n processors. We have implemented the parallel algorithm on a MasPar MP-2 and the theoretical results are confirmed by numerical experiments.

To speed up the computation, augmenting path algorithms usually apply an initialization procedure. We have implemented two different initialization procedures and two augmenting path routines. In order to determine the optimal choice of initialization and augmentation in this parallel setting we have run numerous experiments for all combinations of initialization and augmentation. Our results may be summarized as follows:

- For $n/R < 3$ the naive init is always to prefer
- For $n/R \geq 3$, JV-init is asymptotical best, while naive-init is best for small n. The turn-over point between the two depends on R.
- JV-SAP is to prefer for $n/R < 2$ while CMT-aug is most efficient otherwise.

Acknowledgement: The authors greatfully acknowledge constructive comments from an anonymous referee. In particular we appreciate the valuable assistance on improving our English.

References

[Ber81] D. Bertsekas. A new algorithm for the assignment problem. *Mathematical Programming*, 21:152–171, 1981.

[Ber88] D. Bertsekas. The auction algorithm: A distributed relaxation method for the assignment problem. *Annals of Operations Research*, 14:105–123, 1988.

[BJN+91] M. L. Brady, K. K. Jung, H. T. Nguyen, R. Raghavan, and R Subramonian. The assignment problem on parallel architectures. In *Proceedings of the Dimacs Implementation Challenge Workshop at Rutgers University*, 1991.

[Bla90] Tom Blank. The maspar mp-1 architecture. In *Proceedings of IEEE Compcon Spring 1990*. IEEE, February 1990.

[Chr90] Peter Christy. Software to support massively parallel computing on the maspar mp-1. In *Proceedings of IEEE Compcon Spring 1990*. IEEE, February 1990.

[CMT88] G. Carpaneto, S. Martello, and P. Toth. Algorithms and codes for the assignment problem. *Annals of Operations Research*, 13:193–223, 1988.

[CSW89] D. A. Castanon, B. Smith, and A. Wilson. Performance of parallel assignments algorithms on different multiprocessor architectures. Technical Report TP-1245, Alphatech, INC., 1989.

[For93] High Performance Fortran Forum. High performance fortran, language specification. Technical Report Version 1.0 DRAFT, Rice University, Houston Texas, 1993.

[Her92] B. G. Herland. Mpvms - maspar virtual memory system. Technical Report Report 73, Dept. of Informatics, University of Bergen, 1992.

[JV87] R. Jonker and A. Volgenant. A shortest augmenting path algorithm for dense and sparse linear assignment problems. *Computing*, 38:325–340, 1987.

[Kuh55] H. W. Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2:83–97, 1955.

[KW91] J. L. Kennington and Z. Wang. An empirical analysis of the dense assignment problem: Sequential and parallel implementations. *ORSA Journal on Computing*, 3(4):299–306, 1991.

[Law76] E. L. Lawler. *Combinatorial Optimization: Networks and Matroids*. Holt, Rinehart and Winston, New York, 1976.

[Mun57] J. Munkres. Algorithms for the assignment and transportation problem. *SIAM Journal on Computing*, 5(1):32–38, 1957.

[Nic90] John Nickolls. The design of the maspar mp-1, a cost effective massively parallel computer. In *Proceedings of IEEE Compcon Spring 1990*. IEEE, February 1990.

[Pre93] Lutz Prechelt. Measurement of maspar mp-1216a communication operations. Technical Report 01/93, Institut für Programmstrukturen und Datenorganisation, Fakultät für Informatik, Universität Karlsruhe, 1993.

[PS82] C. H. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Prentice-Hall, Englewood Cliffs, N. J., 1982.

[Sør93] Tor Sørevik. Average case complexity analysis of algorithms for the linear assignment problem. In *Proceedings from NOAS '93*. Univ. of Trondheim, Norway, 1993.

[SS92] S. Storøy and T. Sørevik. An simd, fine-grained, parallel algorithm for the dense linear assignment problem. Technical Report Report 72, Dept. of Informatics, University of Bergen, 1992.

[WZ90] Joel M. Wein and Stavros Zenios. Massively parallel auction algorithms for the assignment problem. Technical Report MIT/LCS/TM-438, Laboratory for Comp. Science, MIT, 1990.

[Zak90] H. Zaki. A comparison of two algorithms for the assignment problem. Technical Report ORL 90-002, Dept. of Mechanical and Industrial Engineering, Univ. of Illinois, Urbana-Champaign, Urbana, Il 61801, 1990.