Automatic Generation of Versatile Benchmarks for Parallel Production System Architecture

J. N. Amaral J. Ghosh

TR TR-PDS-1996-011 July 1996

Parallel & Distributed Systems group

Department of Electrical & Computer Engineering

University of Texas at Austin

Austin, Texas 78712

Automatic Generation of Versatile Benchmarks for Parallel Production System Architectures

José Nelson Amaral *

Joydeep Ghosh†

Electrical and Computer Engineering Department
The University of Texas at Austin,
Austin, TX 78712

August 6, 1996

Abstract

The shortage of adequate benchmarking facilities is a major problem in the proper evaluation of production system machine organizations. This paper presents a new benchmark problem that allows independent variations in the size of the database, the number of productions, the ratio between local and global data, and the variance in the size of local data clusters. This benchmark, available via the Internet through anonymous ftp, is based on the traditional Traveling Salesperson Problem and is simple yet versatile. It can be used to evaluate parallel production system machines, including those for which the criteria for correct concurrent execution is based on the serializability principle. The advantages of our benchmark are illustrated through its application in the performance evaluation of a recently proposed concurrent production system architecture. Our experiments indicate that the performance of this architecture scales with the size of the database, but is degraded by a database with high variance in the size of local data clusters.

*Supported by a grant from Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) and by Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS) - Brazil.

†Supported in part by NSF grant ECS-9307632, and ARO contracts DAAH 04-94-G0417 and DAAH04-95-10494.

1 Introduction

Sound benchmarking is a cornerstone of current computer system evaluation practice. Several computer companies employ large groups that are engaged full-time in the formulation, evaluation/update and application of benchmarks. Their efforts are clearly evident, for example, in the various versions of SPECmarks and transaction processing benchmarks. However, in spite of a high volume of production systems developed for commercial, industrial, military defense, and educational applications in the past fifteen years [4, 12, 16], there is a glaring shortage of comprehensive benchmarks to test new architectures and system organizations. Production System programs written in the corporate world usually contain the expertise of the business for which they are developed and are seldom made publicly available as benchmarking for academic research. Moreover, most of these systems have a fixed number of productions and fixed database size, which make them inappropriate for researchers, who need the ability to independently change the size of the production set and database to study the architectural impacts.

It is thus not surprising that a majority of the architectures and innovative organizations proposed since the 80's were evaluated with a small number of "toy" problems, or with non-versatile benchmarks. At best, they would allow the researcher to modify the size of the database but not its locality properties, or to modify the number of productions without modifying the ratio between global and local productions. A notable exception is the work of Gupta [11], who examined several real-life Production Systems such as R1 and XSEL, used by DEC to find good configurations for computer systems and to act as a sales assistant for VAX computer systems, respectively. However, these production systems *were written for a serial model of execution* that goes through the match-select-act cycle before the next rule to be fired is chosen. Such programs are inadequate when the execution model changes, say, to allow parallel rule firing according to the serializability criterion of correctness [21].

In fact, none of the existing benchmarks were written for a parallel control and execution model, a testimony to the serial control of OPS5 and of popular expert system shells. This is one reason why speedups obtained by researchers who tried to implement such programs on parallel machines, were limited[4]. The situation is somewhat analogous to adapting programs, originally written for serial machines, to run on shared-memory or distributed memory parallel machines, but much worse since effective parallelizing compilers are not available for production systems.

While developing a novel architecture for concurrent execution of production systems [3, 5, 6] we personally encountered the problem of the lack of adequate benchmark programs to test a new organization. After a long, tedious and largely fruitless search for non-trivial public-domain benchmarks for parallel production systems, we came to the conclusion that it may be worthwhile to create a new benchmark that is useful for a broad range of production system architectures. This paper introduces such a benchmark.

In section 2 we list some of benchmarks commonly used to evaluate new organizations for Production System machines. Section 3 presents a new versatile benchmark facility that allows the automatic generation of benchmark programs with varied data set and production set characteristics. Section 4 illustrates the use of benchmarks created by this facility in the evaluation of a concurrent production system machine.

2 Existing Benchmarks

This section summarizes some benchmarks that have been previously used for performance measurement of new organizations and architectures for Production Systems. These benchmarks can be divided into "toy problems" and "real-life programs". In some of the "toy problems", it is possible to expand the size of the database, say by increasing the number of participants in a tournament, the number of guests in a dinner party or the number of rooms in a hotel. However, in general, these benchmarks have a fixed number of productions and fixed amount of locality in the database. Most of these programs are not sophisticated enough to actually evaluate characteristics of new organizations designed to speed up Production Systems. In the category of "real-life programs", we find sizable production systems that might be useful to evaluate new designs. The problem with these benchmarks is that they are not versatile enough to allow changes in the characteristics of the production set or the database, and take some effort and domain knowledge to understand. Also, many of them are not in the public domain.

In Tables 1 and 2, the number of productions and the average number of antecedents and consequents per production indicates the size of the benchmark program. The number of working memory element (WME) types indicates how decoupled the database might be. Some authors make available the initial number of WMEs in the working memory, while others report the average working memory size throughout the program execution. Under "# WMEs" we report the number stated in the reference. This number gives an idea of the size of the database manipulated by each benchmark. A "—" indicates an entry for which information was not provided in the corresponding referenced work.

Bench.	# Prod	Ant./prod	Cons./prod	# WME types	# WMEs	Ref.
`life`	40	6.1	1.3	5	104	[3, 14]
`hotel`	80	4.1	2.0	62	484	[3]
`patents`	86	5.2	1.2	4	136	[3]
`waltz2`	10	2.7	8.0	7	60	[3]
`R1`	1932	5.6	2.9	31	—	[10]
`XSEL`	1443	3.8	2.4	36	62	[10]
`PTRANS`	1016	3.1	3.6	81	—	[10]
`HAUNT`	834	2.4	2.5	23	60	[10]
`DAA`	131	3.9	2.9	20	708	[10]
`SOAR`	103	5.8	1.8	12	353	[10]
`mab`	26	7.3	—	26	67	[7]
`alexia`	15	8.4	—	35	3734	[7]
`chart`	95	4.1	—	78	97	[7]
`amd`	196	8.2	—	197	1610	[7]
`abacab`	107	8.0	—	171	1028	[7]

Table 1: Static measures for some existing benchmarks.

The benchmarks included in Tables 1 and 2 are found in the works of Acharya *et al.* [2, 1], Amaral [3, 4], Bouaud [7], Brant *et al.* [9], Dixit and Moldovan [8], Gupta [10], Kuo *et al.* [14], Kuo and Moldovan [15], Miranker and Lofaso [17, 18], Neiman [19], Oflazer [20], Schmolze [21], and Ishida [13]. We now briefly state what each benchmark program does.

In Tables 1 and 2, `MAB`, `M&B`, `mab` are different implementations of the classic "monkey and ba-

Bench.	# Prod	Ant./prod	# WMEs	Pub.
`MAB`	13	2.6	11	[17, 18]
`Mud`	884	2.4	241	[17]
`Waltz`	33	3.9	42	[17]
`Mesgen`	155	2.9	34	[17, 18]
`Mapper`	237	3.3	1153	[17, 18]
`Jig25`	6	—	50	[18]
`Tourney`	17	—	123	[18]
`Robot`	75	—	410	[18]
`Rubik`	70	—	287	[18]
`weaver`	637	—	152	[2, 9, 18]
`waltz`	33	—	42	[9, 18]
`manners`	8	—	—	[9]
`ARP`	118	—	—	[9]
`Cafeteria`	94	—	—	[15]
`Tournam.`	26	—	—	[15]
`Toru-Waltz`	48	—	—	[15, 14]
`Hotel`	723	—	—	[15]
`Snap`	574	—	—	[15]
`CKT_Des`	107	—	400	[13]

Table 2: Static measures for some existing benchmarks.

nanas" problem, `chart` is a syntax chart parser, `amd` is a semantic analyzer for natural language, `abacab` is a blackboard controller. `R1` configures VAX computer systems, `XSEL` acts as a sales assistant for VAX computer systems, `PTRANS` is a program for factory management, `HAUNT` is an adventure game program, `DAA` is a program for VLSI design, and `SOAR` is an experimental problem solving architecture implemented as a production system. `Tournament` schedules bridge tournaments, `waltz`, `waltz2`, and `Toru-Waltz` are different implementations of the line labeling problem, `Cafeteria` sets up a cafeteria, `hotel` models a hotel operation, `patents` is a solution for the "Confusion of Patents Problem", and `life` is Conway's game of life. `Mud` analyzes the casting from oil wells, `Mapper` assists a tourist to navigate Manhattan's public transportation system, `Mesgen` takes Dow Jones figures and converts them into text describing the course of a trading day. `Robot` plans the movements for a robot arm, `Jig25` is a simple jigsaw puzzle solver, `Tourney` schedules players for a bridge tournament, `Weaver` is a VLSI box router, and `Rubik` solves Rubik's cube. `CKT_des` is a circuit design expert system developed at NTT Laboratories.

The limitations of the benchmarks encountered in the literature indicates that the research community would greatly benefit from the development of versatile benchmarking facilities that allow not only for the expansion of the knowledge base by replication of data, but also for changes in the number of productions, and in the locality properties of the knowledge base. It is desirable that such benchmark facilities use a standard production system language to facilitate its use by different research groups.

In section 3 we present a new facility to produce benchmark programs for production systems. These programs solve a modification of the well-known Traveling Salesperson Problem (TSP) that we call the *Contemporaneous TSP* (CTSP). This benchmark allows the researcher to modify the size of the database, the number of productions, the database size, the amount of locality in the database, and the ratio between productions that access local and global data. All these modifications are implemented by simply modifying a "map" that specifies the location of the cities.

3 A Contemporaneous TSP

In this modified version of the TSP, cities are grouped into states or "countries". The tour has to be constructed such that the salesperson enters each country only once. The location and borders of the countries must allow the construction of a tour observing this restriction. The problem is formally stated as follows:

> An instance of CTSP is represented by $(K, C, c, \mu_c, \sigma_c, O, d)$. $K = \{C_1, C_2, ..., C_n\}$ is a "continent" formed by "countries". Each country $C_i = \{c_{i,1}, c_{i,2}, ..., c_{i,m(i)}\}$ contains $m(i)$ "cities". The number of cities per country $m(i)$ is normally distributed with average μ_c and standard deviation σ_c. The ordering $O =< C_{\pi(1)}, C_{\pi(2)}, ..., C_{\pi(n)} >$ specifies the order in which the countries shall be visited. The function $d(c_{i,k}, c_{j,l}) \in Z^+$ specifies the distance between any two cities in the continent. The problem consists of finding an ordering of cities $< c_{i,\tau(1)}, c_{i,\tau(2)}, ..., c_{i,\tau(m(i))} >$ within each country C_i that minimizes the cost of the global tour:

$$\sum_{i=1}^{n} \sum_{j=1}^{m(i)-1} d(c_{i,\tau(j)}, c_{i,\tau(j+1)}) \quad + \quad \sum_{i=1}^{n-1} d(c_{\pi(i),\tau(m(i))}, c_{\pi(i+1),\tau(1)}) \ + \\ d(c_{(\pi(n),\tau(m(n))}, c_{\pi(1),\tau(1)}). \qquad (1)$$

The first term of equation 1 sums the lenght of the tour within each country. The second term sums the distance between the last city visited in country i and the first city visited in country $i+1$. The last term of equation 1 is the distance between the last city visited in country $C_{\pi(n)}$ and the first city visited in country $C_{\pi(1)}$.

This formulation of TSP is called "contemporaneous" because it reflects some aspects of modern day life. In the current global economy, travelpersons are likely to have greater needs than the traditional salesperson driving from town to town. Consider a music star in a worldwide tour carrying along a huge crew and sophisticated equipment: the singer will visit many different locations in each continent; the cost of flying back and forth between continents is much higher than movements within a continent and depends on the locations of departure and arrival. Other situations involving sophisticated traveling requirements include the planning of airline routes and national political campaigns in large countries such as USA, Brazil and India. Applications in which data locality allows the creation of clusters include: insurance database management, banking industry, a national health care information network, and a national criminal offense information network[1].

[1]In the 1994 "Brady Bill", the USA Congress mandated the construction of such a network for background verification for the purchase of fire weapons.

3.1 A Production System Solution for CTSP

The formulation presented above for the CTSP is generic enough to allow its application in many fields: there is no restriction in what the words continent, country, city, and distance might represent. To facilitate the construction of a Production System solution that is useful for testing new PS architectures, we used a simpler version of CTSP with the following restrictions:

- The problem is symmetric, i.e., $d(c_{k,i}, c_{l,j}) = d(c_{l,j}, c_{k,i})$ for any i, j, k, and l.
- A continent is a two-dimensional Euclidian space.
- A country is a contiguous, rectangular shape within this space.
- The number of cities in each country follows a normal distribution with average μ_c and standard deviation σ_c.
- The city locations are uniformly distributed within each country.
- There is a common boundary between two countries that are consecutive in the ordering O.

This simplified version can be solved in different ways. Thus one needs to further decide on specific solutions (set of rules and control mechanism), and then apply to specific instantiations (city and country locations). We have studied one solution, `tsp`, and a variant, `tsp2`. `tsp` consists of a set of productions for each country and a set of productions for each country boundary. The control mechanism is based on the *serializability principle*: any fireable production can be executed concurrently with other productions that are being fired, so long as the end result is the same as at least one sequential execution of the instantiation set [21]. Thus it is tailor-made for parallel production systems.

The data set is constructed in such a way that the distances among cities located within each country are stored in WMEs with different types. Given a country C_i, the country that precedes C_i in the order O is denoted by $P(C_i)$; the country that succeeds C_i in the order O is denoted $S(C_i)$. It is not necessary to store in the data base the distance between every two cities in the continent. For a city $c_{i,j}$ in a country C_i, the only relevant distances are the distance to the cities within C_i, to the cities in $P(C_i)$, and to the cities in $S(C_i)$. The following list illustrates WMEs typically used in our solution to CTSP:

```
(GERMANY_city ^name GERMANY_01 ^status not_in_trip)
(FRANCE_city ^name FRANCE_10 ^status in_trip)
(GERMANY_dist ^from GERMANY_04 ^to GERMANY_07 ^value 135)
(FRANCE_GERMANY_dist ^from FRANCE_14 ^to GERMANY_03 ^value 357)
(GERMANY_POLAND_dist ^from GERMANY_01 ^to POLAND_05 ^value 55)
```

Our solution has seventeen *local* productions per country and twelve productions per country boundary. This organization allows the researcher to vary the number of productions by creating continents with different number of countries. The size of the data base is determined by the number of countries and the average number of cities per country. The variance in the amount of data processed by each cluster of production is directly related to σ_c^2, the variance in the number of cities per country.

The heuristic used in the PS solution of the problem involves the computation of two extra locations for each country C_i: the geometric center of the borders with $P(C_i)$ and with $S(C_i)$. Because we impose the restriction that countries have rectangular shapes in a two-dimensional Euclidian space, the border between two subsequent countries in the tour is always a segment of a straight line. The *border center* $b(C_i, C_j)$ between countries C_i and C_j is the center of the line segment that forms the boundary. The heuristic used to construct the internal tour in a country C_i is described below:

- The first city $c_{i,\tau(1)}$ in the internal tour of a country C_i is the city with minimum distance $d(b(C_i, P(C_i)), c_{i,\tau(1)})$.
- While the internal tour of country C_i is not complete, select a city $c_{i,l} \in C_i$ such that $d(c_{i,k}, c_{i,l}) - d(c_{i,l}, b(C_i, S(C_i)))$ is minimum, where $c_{i,k}$ is the latest city inserted in the tour.
- Whenever the internal tours of two adjacent countries C_i and C_j are completed, the last city visited in C_i is connected to the first city visited in C_j.
- Whenever there is a segment of tour formed by four cities $(c_{m,\tau(i)}, c_{m,\tau(j)}, c_{m,\tau(k)}, c_{m,\tau(l)})$ such that $d(c_{m,\tau(i)}, c_{m,\tau(j)}) + d(c_{m,\tau(k)}, c_{m,\tau(l)}) > d(c_{m,\tau(i)}, c_{m,\tau(k)}) + d(c_{m,\tau(j)}, c_{m,\tau(l)})$, change this segment of tour to $(c_{m,\tau(i)}, c_{m,\tau(k)}, c_{m,\tau(j)}, c_{m,\tau(l)})$.

The rationale of this heuristic is to add to the internal tour the cities that are close to the latest city included in the tour and far from the border in which the internal tour ends. There is a limited local optimization of the constructed tour. We developed a C program that allows researchers to specify continent maps and to experiment with different numbers of countries, μ_c, and σ_c.

Table 3 summarizes the static characteristics of `tsp`, as applied to problems with n countries, with an average of μ_c cities per country. In `tsp`, a single set of productions performs the optimization at all country borders. In the second solution, identified as `tsp2` in Table 3, a specialized set of productions is used in the optimization of each country border.

Measure	`tsp`	`tsp2`
# of productions	$20\,n + 1$	$30\,n + 1$
# WME types	$8\,n + 8$	$15\,n + 1$
# WMEs in initial database	$n\,(2\,\mu_c^2 + 2\,\mu_c + 3)$	$n\,(2\,\mu_c^2 + 2\,\mu_c + 3)$

Table 3: Static measures for the CTSP benchmark as functions of C and μ_c

This simplified CTSP offers many advantages for production system benchmarking: the number of productions in the program can be varied by changing the number of countries; the ratio of global to local data is controlled by the average number of cities in each country; the variations in the size of local data clusters is specified by σ_c; and the specification of the continent "map" is very simple making it easy for a researcher to generate new instantiations of the benchmark. The CTSP benchmarking facility is available through anonymous ftp to: `pine.ece.utexas.edu` in `/a/pine/home/pine/ftp/pub/parprosys`. Some parts of this facility, dealing with a sample map generation, templates for creating WMEs and the rule templates for `tsp2` are provided in Appendices A, B and C respectively.

3.2 Static Measures for Benchmarks Generated by CTSP

In section 2 we presented static measures for benchmarks used in studies described in the literature. An analysis of Tables 1 and 2 indicates that important parameters, such as number of productions and size of the initial data base, are not distributed over a continuous range from which a researcher can pick several evaluation points. CTSP alleviates this problem by allowing the easy generation of benchmark programs wherein one can arbitrate independently the number of productions and the size of the initial data base. As an illustration, some examples of design points in the space characterized by the number of WMEs in the initial data base and the number of productions in the program generated by `tsp`, are shown in Figure 1. This graph demonstrates that with CTSP, a researcher can generate benchmark programs in different regions of this design space.

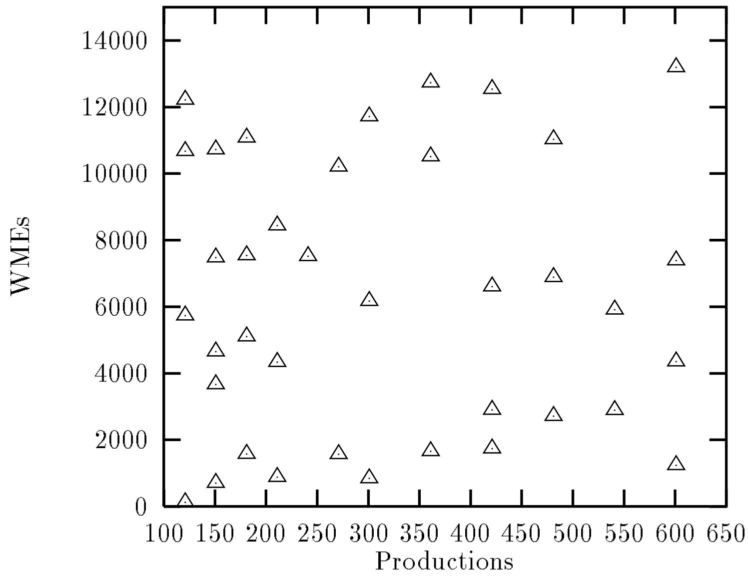

Figure 1: Various design points provided by `tsp` due to changes in the average number of cities in each country (μ_c). The standard deviation in the number of cities per country has been held constant ($\sigma_c = 3$).

Another interesting feature in a tool that automatically generates benchmarks with a database that can be divided among data clusters, is the ability to play with the variation in the size of these data clusters. This ability allows the investigation of the impact of data locality in a new architectural or system solution. Table 4 shows the average number of WMEs per cluster and the standard deviation for the number of WMEs per cluster in some benchmarks generated by CTSP. We also show in Table 4 the average and the standard deviation for the number of cities per state in the benchmarks generated.

Benchmark	# Clusters	# WMEs/Cluster		# Cities/State	
		Average	Std Deviation	Average	Std Deviation
mapa_5_0_8	5	83.0	2.2	8.0	0.0
mapa_5_1_22	5	531.8	92.0	22.0	2.0
mapa_5_2_33	5	1119.0	174.4	32.4	2.7
mapa_10_0_25	10	678.0	3.2	25.0	0.0
mapa_10_1_15	10	249.3	89.0	14.7	2.8
mapa_10_2_5	10	45.6	65.5	5.4	5.1
mapa_12_3_8	12	78.7	161.0	7.3	9.3
mapa_12_4_20	12	444.8	475.4	19.8	10.3
mapa_12_1_23	12	566.2	70.6	22.8	1.5
mapa_14_2_5	14	34.7	61.2	4.6	4.8
mapa_14_0_15	14	258.0	3.7	15.0	0.0
mapa_14_1_20	14	437.9	143.8	19.9	3.4
mapa_16_2_8	16	81.2	124.9	7.7	7.7
mapa_20_3_5	20	46.2	107.6	5.3	8.7
mapa_20_4_17	20	337.2	409.6	17.1	10.8
mapa_32_2_20	32	454.2	349.6	20.2	8.0
mapa_64_1_15	64	258.5	181.3	15.0	5.7

Table 4: Examples of benchmarks generated by CTSP.

4 Performance Evaluation Using CTSP

CTSP was used to measure the performance of a new parallel architecture described in [5, 6][2]. This architecture is formed by a number of identical processors connected through a bus. At compile time each production is uniquely assigned to a processor according to a partitioning algorithm that takes into consideration inter-production dependencies and workload balance. Each processor stores locally all Working Memory Elements (WME) that are tested by its production antecedents. This architecture uses a *partially informed* selection to choose the next production to be fired, allowing the superposition of the matching and acting phases of a production system. The results produced were proven to be correct under the serializability criterion. The architecture relies on the use of associative memories as lookaside tables to guarantee correct operation without global synchronization. This architecture also uses multiple functional units (called β-units) in the matching engine *within* each processor to execute the Rete algorithm.

All experiments presented in this section use an instantiation of CTSP with seven "states"[3]. The mean and the variance for the number of cities per state is set based on the needs of each experiment. The seven-state map is shown in Figure 2 (also see Appendix A). We perform two set of experiments. First, we maintain the standard deviation σ_c constant and change the average number of cities in each state in order to study the effect of problem size on performance. Then we maintain the average number of cities in each state constant and change the standard deviation. This allows us to study the effect of load imbalance among processors.

Figure 3 plots results for the first experiment in which the mean of the number of cities per state

[2]We have not described the architecture here to avoid overlap with [6]. The interested reviewer is requested to ftp this article or download it from our home-page.

[3]In the description of CTSP on section 3 a group of cities form a "country" and a group of countries form a "continent". In the experiments described here, cities form states and states form countries.

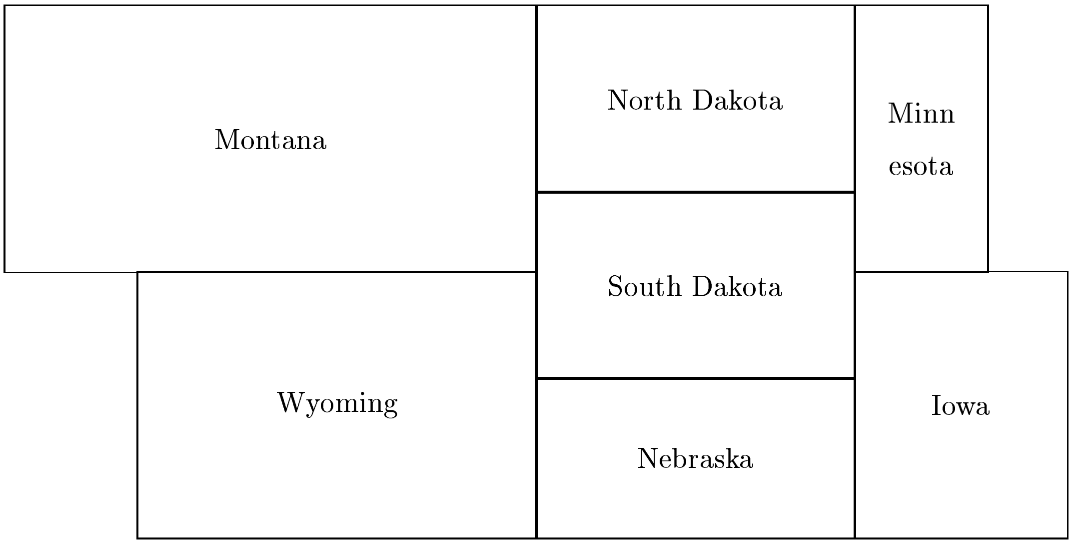

Figure 2: Abstract "Country" map (with apologies to Minnesotans).

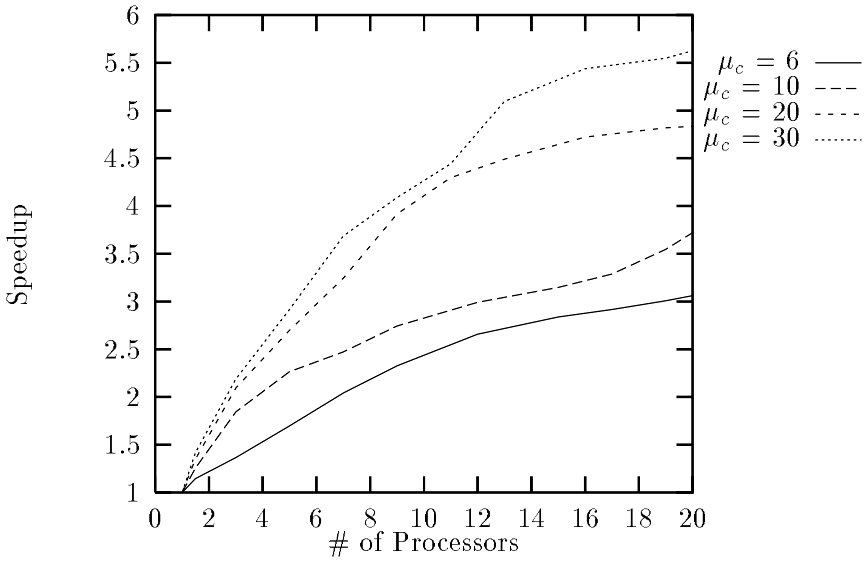

Figure 3: Variations in speedup due to changes in the average number of cities per country (μ_c). Variance in the number of cities is constant ($\sigma_c = 3$).

is changed while its variance remains constant. It shows the speed improvement[4] as the number of processors is increased in a machine with a 10 β-unit Rete Network. The base of comparison for these curves is a machine with a single processor and 10 β-units in the Rete Network. The effect of problem size on speedup is evident.

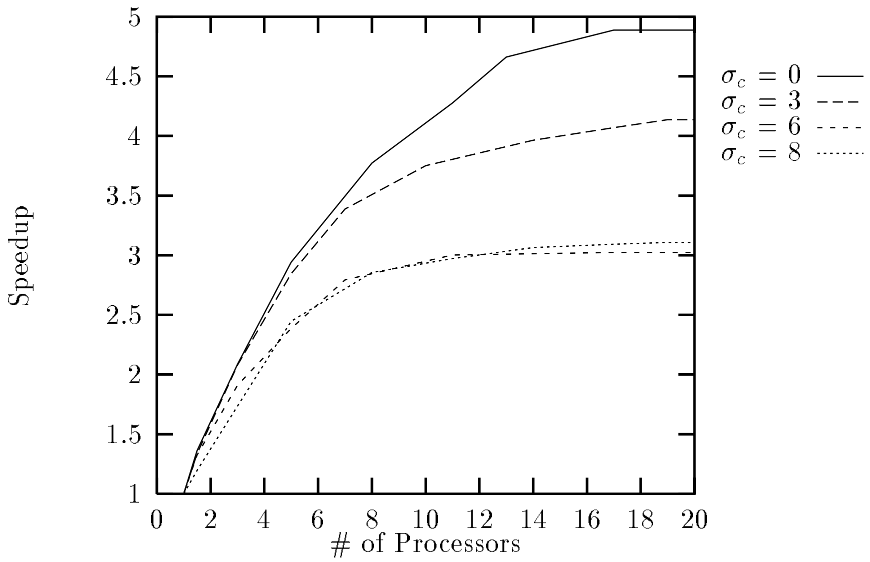

Figure 4: Effect of load imbalance on the speedup of parallel architectures.

Figures 4 quantifies the effect of imbalance among the different local data clusters. In this graph, the mean for the number of cities per state remains constant while the standard deviation is varied. Figure 4 plots the curves for a 10 β-units per processor architecture. The base of comparison for the curves in Figure 4 is a single processor architecture with 10 β-units.

Figure 5 studies a similar effect when each processor has only one β unit, for systems with 8 and 16 processors respectively. The amount of speedup obtained decreases when there is a higher variance in the number of cities in each state. This is an expected effect because higher variance in the size of local clusters causes more imbalance in processor workload.

5 Conclusion

This paper aims to redress the lack of adequate benchmarks to evaluate parallel production system architectures. We presented a new benchmark facility that allows the user to independently modify the size of the database, number of productions, ratio between the amount of local and global data, and the size of local data clusters. Benchmarks generated by this facility have the versatility of "toy" problems yet can be expanded to the data and production set size of "real world" problems. These benchmarks also produce correct results when serializability is used as the criterion for correctness.

[4]The speed improvement measured in the graphs of Figures 3, 4, and 5 is only due to increase in number of processors. Substantial speedup due to the use of multiple functional units in the matching engine is not shown in this graph since we are always comparing systems with the same number of functional units per processor.

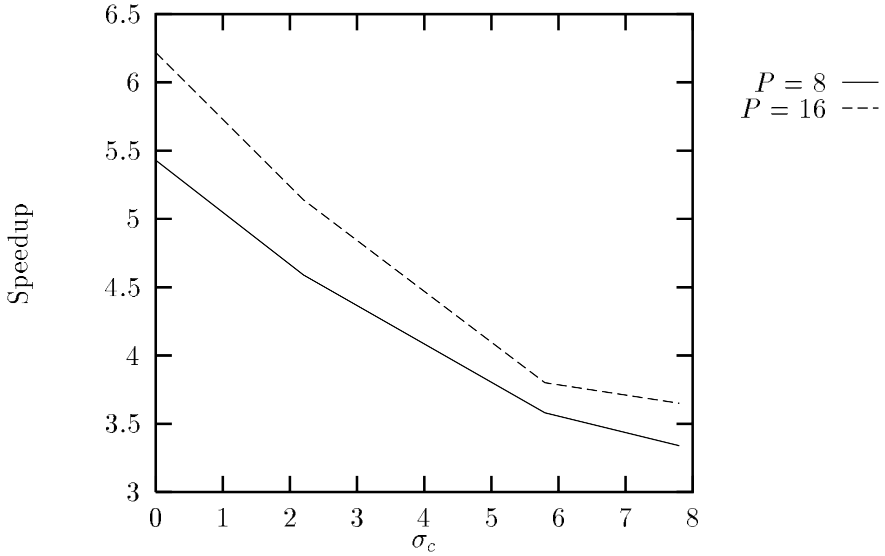

Figure 5: Effect of the variance in the number of cities per country, (σ_c^2) on speedup. Both 8 processor and 16 processor configurations have a single β-unit per processor, and are compared to a uniprocesor with one β-unit.

We illustrated the use of the benchmark facility in studying the performance of an architecture for concurrent production systems. The results indicate that for a given standard deviation in the size of local data clusters, the amount of speedup obtained by the architecture scales well with the size of the database. The improvement in performance for databases with low variance among local data cluster sizes, is also quantified.

References

[1] A. Acharya and M. Tambe. Collection-oriented match: Scaling up the data in production systems. Tech. Rep. CMU-CS-92-218, Carnegie-Mellon University, Pittsburgh, December 1992.

[2] A. Acharya, M. Tambe, and A. Gupta. Implementation of production systems on message-passing computers. In *IEEE Trans. on Parallel and Distributed Systems*, volume 3, pages 477–487, July 1992.

[3] J. N. Amaral. *A Parallel Architecture for Serializable Production Systems.* PhD thesis, The University of Texas at Austin, Austin, TX, December 1994. Electrical and Computer Engineering.

[4] J. N. Amaral and J. Ghosh. Speeding up production systems: From concurrent matching to parallel rule firing. In L. N. Kanal, V. Kumar, H. Kitani, and C. Suttner, editors, *Parallel Processing for AI*, chapter 7, pages 139–160. Elsevier Science Publishers B.V., 1994.

[5] J. N. Amaral and J. Ghosh. Performance measurements of a concurrent production system architecture without global synchronization. In *Proc. 9th International Parallel Processing Symposium*, pages 790–797, Santa Barbara, CA, April 1995.

[6] J. N. Amaral and J. Ghosh. A concurrent architecture for serializable production systems. *IEEE Transactions on Parallel and Distributed Processing*, 1996. Accepted for publication. Can be downloaded by anonymous ftp (`pegasus.ece.utexas.edu`, file `/pub/papers/concurps_itpdp.ps.Z`), or obtained from www.lans.utexas.edu under Selected Publications - Journal Papers.

[7] J. Bouaud. Tree: the heuristic driven join strategy of a rete-like matcher. In *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pages 496–502, August 1993.

[8] V. V. Dixit and D. L. Moldovan. The allocation problem in parallel production systems. *Journal of Parallel and Distributed Computing*, 8:20–29, 1990.

[9] D. A. Brant et al. Effects of database size on rule system performance: Five case studies. In *17th Conference on Very Large Databases*, September 1991.

[10] A. Gupta. Implementing OPS5 production systems on DADO. In *Proceedings of International Conference on Parallel Processing*, pages 83–91, 1984.

[11] A. Gupta. *Parallelism in Production Systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, March 1986.

[12] K. Hwang, J. Ghosh, and R. Chowkwanyun. Computer architectures for artificial intelligence processing. *IEEE Computer*, pages 19–29, Jan. 1987.

[13] T. Ishida. An optimization algorithm for production systems. *IEEE Transaction on Knowledge and Data Engineering*, 6(4):549–558, August 1994.

[14] C.-M. Kuo, D. P. Miranker, and J. C. Browne. On the performance of the CREL system. *Journal of Parallel and Distributed Computing*, 13:424–441, December 1991.

[15] S. Kuo and D. Moldovan. Implementation of multiple rule firing production systems on hypercube. *Journal of Parallel and Distributed Computing*, 13:383–394, December 1991.

[16] S. Kuo and D. Moldovan. The state of the art in parallel production systems. *Journal of Parallel and Distributed Computing*, 15:1–26, June 1992.

[17] D. P. Miranker. Treat: A better match algorithm for AI production systems. In *Proceedings of National Conference on Artificial Intelligence*, pages 42–47, July 1989.

[18] D.P. Miranker and B. J. Lofaso. The Organization and Performance of a TREAT Based Production System Compiler. *IEEE Trans. on Knowledge and Data Engineering*, pages 3–10, March 1991.

[19] D. E. Neiman. *Design and Control of Parallel Rule-Firing Production Systems.* PhD thesis, University of Massachusets, Amherst, MA, September 1992.

[20] K. Oflazer. Partitioning in parallel processing of production systems. In *Proceedings of International Conference on Parallel Processing*, pages 92–100, 1984.

[21] J. G. Schmolze. Guaranteeing serializable results in synchronous parallel production systems. *Journal of Parallel and Distributed Computing*, 13:348–365, December 1991.

A Map File

The following is the input file to generate the program `mount` using the benchmark generating tool.

```
states: 10
average_cities_per_state: 15
standard_deviation: 3
MONTANA {(0,2500);(2000,3500)}
WYOMING {(500,1500);(2000,2500)}
COLORADO {(700,500);(2200,1500)}
NORTH_DAKOTA {(2000,2800);(3200,3500)}
SOUTH_DAKOTA {(2000,2100);(3200,2800)}
NEBRASKA {(2000,1500);(3200,2100)}
KANSAS {(2200,500);(3500,1500)}
MINNESOTA {(3200,2500);(3700,3500)}
IOWA {(3200,1500);(4000,2500)}
MISSOURI {(3500,0);(4100,1500)}
MONTANA NORTH_DAKOTA
NORTH_DAKOTA MINNESOTA
MINNESOTA IOWA
IOWA MISSOURI
MISSOURI KANSAS
KANSAS COLORADO
COLORADO WYOMING
WYOMING NEBRASKA
NEBRASKA SOUTH_DAKOTA
SOUTH_DAKOTA MONTANA
end
```

B WME templates

Following is a list of templates to create WMEs for each state generated by the tool. Also you will find the templates used for pairs of states. In the code below the characters `@`, `#`, and `%` are used as wild cards. They are automatically replaced by the name of the states in the map provided by the user.

```
; ******************************************
; WMEs used for a specific state
;
(literalize  @_city status name fitness)
(literalize  @_distance from to value)
```

```
(literalize  @_distance_to_border city
             state value)
(literalize  @_trip status from to distance)
(literalize  @_next_city name fitness)
(literalize  @_optimization status city_1
             city_2 fitness)
(literalize  @_border_city name)
(literalize  @_all_borders_dist_computed
             value)

; ******************************************
; WMEs used for pair of states
;

(literalize  @_#_border status)
(literalize  @_#_crossing from_status
             from_city to_status to_city
             distance)
(literalize  @_border_distance from to value)
(literalize  @_border_trip status position
             from to distance)
(literalize  @_#_distance from to value)
(literalize  @_#_optim_from from_city to_city
             fitness)
(literalize  @_#_optim_to   from_city to_city
             fitness)
(literalize  @_city status name fitness)
(literalize  @_distance from to value)
(literalize  @_distance_to_border city state
             value)
(literalize  @_trip status from to distance)
(literalize  @_next_city name fitness)
(literalize  @_optimization status city_1
             city_2 fitness)
(literalize  @_border_city name)
(literalize  @_all_borders_dist_computed
             value)
```

C Rule Templates

Next we present the actual rule templates used by the benchmark generating tool to create the programs. In these rules also the wild cards get replaced by state names.

```
; ******************************************
; We are done when all the inner trip
; writings are done
```

```
(p termination
 (inner_trips_writing ^status done)
 -(inner_trips_writing ^status undone)
 -(inner_trips_writing ^status finito)
 -->
 (write end)
 (modify 1 ^status finito))

; *******************************************

(p @_create_first_city_candidates
  (@_all_borders_dist_computed ^value true)
  (@_city ^name <c1> ^status not_in_trip)
  (%_@_border)
  (@_distance_to_border ^city <c1>
                        ^state % ^value <df>)
  (@_#_border)
  (@_distance_to_border ^city <c1>
                        ^state # ^value <dt>)
  -(@_city ^status in_trip)
  -(@_city ^status first_city)
  -->
  (modify 2 ^status first_city_candidate
          ^fitness (compute (<df> - <dt>))))

; *******************************************

(p @_choose_first_city
 (@_city ^name <c1>
   ^status first_city_candidate ^fitness <f>)
 -(@_city ^status first_city_candidate
                              ^fitness < <f>)
 -(@_city ^status not_in_trip)
 -(@_city ^status first_city)
 -->
 (modify 1 ^status first_city))

; *******************************************

(p @_remove_first_city_candidates
 (@_city ^name <c1> ^status first_city)
 (@_city ^name <c2>
        ^status first_city_candidate)
 -(@_city ^status in_trip)
 -->
 (modify 2 ^status not_in_trip))
```

```
; ******************************************

(p @_put_first_city_in_trip
 (@_city ^name <c> ^status first_city)
 -(@_city ^status first_city_candidate)
 -->
 (modify 1 ^status in_trip))

; ******************************************
;

(p @_create_border_trips_1
 (@_trip ^from <c1> ^to <c2> ^distance <d12>)
 (@_trip ^from <c2> ^to <c3> ^distance <d23>)
 (@_distance ^from <c1> ^to <c3>
                       ^value <d13>)
 -(@_city ^status not_in_trip)
 -(@_city ^status first_city)
 -(@_city ^status just_chosen)
 -(@_city ^status first_city_candidate)
 -(@_city ^status next_city_candidate)
 -(@_optimization)
 -(@_trip ^to <c1>)
 -->
 (make @_border_trip ^status created
      ^position begin ^from <c1> ^to <c2>
                           ^distance <d12>)
 (make @_border_trip ^status created
       ^position begin ^from <c2> ^to <c3>
                           ^distance <d23>)
 (make @_border_distance ^from <c1>
                     ^to <c2> ^value <d12>)
 (make @_border_distance ^from <c2>
                     ^to <c3> ^value <d23>)
 (make @_border_distance ^from <c1>
                     ^to <c3> ^value <d13>)
 (make @_border_distance ^from <c2>
                     ^to <c1> ^value <d12>)
 (make @_border_distance ^from <c3>
                     ^to <c2> ^value <d23>)
 (make @_border_distance ^from <c3>
                     ^to <c1> ^value <d13>)
 (make @_border_city ^name <c1>)
 (make @_border_city ^name <c2>)
 (make @_border_city ^name <c3>))

; ******************************************
```

```
(p @_create_border_trips_2
 (@_trip ^from <c1> ^to <c2> ^distance <d12>)
 (@_trip ^from <c2> ^to <c3> ^distance <d23>)
 (@_distance ^from <c1> ^to <c3>
                       ^value <d13>)
 -(@_city ^status not_in_trip)
 -(@_city ^status first_city)
 -(@_city ^status just_chosen)
 -(@_city ^status first_city_candidate)
 -(@_city ^status next_city_candidate)
 -(@_optimization)
 -(@_trip ^from <c3>)
 -->
 (make @_border_trip ^status created
       ^position end ^from <c1> ^to <c2>
                          ^distance <d12>)
 (make @_border_trip ^status created
       ^position end ^from <c2> ^to <c3>
                          ^distance <d23>)
 (make @_border_distance ^from <c1>
                       ^to <c2> ^value <d12>)
 (make @_border_distance ^from <c2>
                       ^to <c3> ^value <d23>)
 (make @_border_distance ^from <c1>
                       ^to <c3> ^value <d13>)
 (make @_border_distance ^from <c2>
                       ^to <c1> ^value <d12>)
 (make @_border_distance ^from <c3>
                       ^to <c2> ^value <d23>)
 (make @_border_distance ^from <c3>
                       ^to <c1> ^value <d13>)
 (make @_border_city ^name <c1>)
 (make @_border_city ^name <c2>)
 (make @_border_city ^name <c3>))
```

The criterion to select the next city in the tour involves selecting the closest city and also the one that is farthest from the final border. It is desirable to choose the cities closer to the final border at the end of the tour. The heuristic compares the difference between the distances of a newly selected city to the last city included in the tour and the final border in the current state.

```
(p @_make_next_city_candidates
  (@_city ^name <c2> ^status in_trip)
  (@_city ^name <c3> ^status not_in_trip)
  (@_distance ^from <c2> ^to <c3>
                        ^value <d23>)
  (@_#_border)
  (@_distance_to_border ^city <c3>
```

```
                   ^state # ^value <d_c3_s>)
  -(@_city ^status first_city_candidate)
  -(@_city ^status  just_chosen)
  -(@_trip ^from <c2>)
  -->
  (modify 2 ^status next_city_candidate
      ^fitness (compute (<d23> - <d_c3_s>))))

; *******************************************

(p @_choose_next_city
  (@_city ^name <c2> ^status in_trip)
  (@_city ^name <c3>
    ^status next_city_candidate ^fitness <f>)
  (@_distance ^from <c2> ^to <c3>
                                ^value <d23>)
  -(@_city ^status next_city_candidate
                              ^fitness < <f>)
  -(@_trip ^from <c2>)
  -(@_city ^status not_in_trip)
  -(@_city ^status first_city_candidate)
  -(@_city ^status just_chosen)
  -->
  (make @_trip ^status not_printed
        ^from <c2> ^to <c3> ^distance <d23>)
  (modify 2 ^status just_chosen))

; *******************************************

(p @_discharge_remaining_candidates
  (@_city ^name <c1> ^status just_chosen)
  (@_city ^name <c2>
   ^status next_city_candidate ^fitness <f>)
  -->
  (modify 2 ^status not_in_trip))

; *******************************************

(p @_change_status_chosen_city
  (@_city ^name <c> ^status just_chosen)
  -(@_city ^status next_city_candidate)
  -->
  (modify 1 ^status in_trip))
```

The optimization of the tour is local. It involves three trips and consists of changing the order in which those trips are taken.

```
(p @_make_optimization_candidate
  (@_trip ^status not_printed ^from <c1>
                   ^to <c2> ^distance <d12>)
  (@_trip ^status not_printed ^from <c2>
                   ^to <c3> ^distance <d23>)
  (@_trip ^status not_printed ^from <c3>
                   ^to <c4> ^distance <d34>)
  (@_distance ^from <c1> ^to <c3>
                        ^value <d13>)
  (@_distance ^from <c3> ^to <c2>
                        ^value <d32>)
  (@_distance ^from <c2> ^to <c4>
                        ^value <d24>)
  -(@_optimization ^status selected)
  -(@_optimization ^status implemented)
  -(@_boundary_city <c2>)
  -(@_boundary_city <c3>)
  -->
  (make @_optimization ^status generated
                   ^city_1 <c2> ^city_2 <c3>
^fitness (compute ( (<d12> + (<d23> + <d34>))
              - (<d13> + (<d32> + <d24>))))))

; *******************************************

(p @_select_optimization
  (@_optimization ^status generated
                 ^city_1 <c2> ^city_2 <c3>
                 ^fitness {<f> > 0})
  -(@_optimization ^status selected)
  -(@_optimization ^fitness > <f>)
  -->
  (modify 1 ^status selected))

(p @_remove_optimization_1
  (@_optimization ^status selected)
  (@_optimization ^status generated
    ^city_1 <c1> ^city_2 <c2> ^fitness <f>)
  -->
  (remove 2))

; *******************************************

(p @_remove_optimization_2
  (@_optimization ^city_1 <c1> ^city_2 <c2>
                 ^fitness {<f> <= 0})
  -->
  (remove 1))
```

```
; ******************************************

(p @_implement_optimization
  (@_optimization ^status selected
                 ^city_1 <c2> ^city_2 <c3>)
  (@_trip ^from <c1> ^to <c2>
                    ^distance <d12>)
  (@_trip ^from <c2> ^to <c3>
                    ^distance <d23>)
  (@_trip ^from <c3> ^to <c4>
                    ^distance <d34>)
  (@_distance ^from <c1> ^to <c3>
                        ^value <d13>)
  (@_distance ^from <c2> ^to <c4>
                        ^value <d24>)
  -(@_optimization ^status generated)
  -->
  (modify 1 ^status implemented)
  (remove 2)
  (remove 3)
  (remove 4)
  (make @_trip ^status not_printed ^from <c1>
              ^to <c3> ^distance <d13>)
  (make @_trip ^status not_printed ^from <c3>
              ^to <c2> ^distance <d23>)
  (make @_trip ^status not_printed ^from <c2>
              ^to <c4> ^distance <d24>))

; ******************************************

(p @_remove_implemented_optimization
  (@_optimization ^status implemented
                 ^city_1 <c2> ^city_2 <c3>)
  -(@_optimization ^status generated)
  -->
  (remove 1))
```

The rules that optimize the border are independent of the rules that generate the internal tour within the state. At times there might be some conflicts between the internal trips of one state and the border trips. Whenever this happens, the border trips are maintained.

```
(p @_correct_conflicting_trips_1
  (@_trip ^from <c1> ^to <c2>)
  (@_trip ^from <c2> ^to <c3>)
  (@_border_trip ^status modified ^from <c1>
                ^to <c3> ^distance <d13>)
```

```
  (@_border_trip ^status modified ^from <c3>
                 ^to <c2> ^distance <d32>)
  -->
  (remove 1)
  (remove 2)
  (modify 3 ^status corrected)
  (modify 4 ^status corrected)
  (make @_trip ^status not_printed ^from <c1>
               ^to <c3> ^distance <d13>)
  (make @_trip ^status not_printed ^from <c3>
               ^to <c2> ^distance <d32>))

; *******************************************

(p @_correct_conflicting_trips_2
  (@_trip ^from <ca> ^to <cb>)
  (@_trip ^from <cb> ^to <cc>)
  (@_border_trip ^status modified ^from <cb>
                 ^to <ca> ^distance <d13>)
  (@_border_trip ^status modified ^from <ca>
                 ^to <cc> ^distance <d32>)
  -->
  (remove 1)
  (remove 2)
  (modify 3 ^status corrected)
  (modify 4 ^status corrected)
  (make @_trip ^status not_printed ^from <cb>
               ^to <ca> ^distance <d13>)
  (make @_trip ^status not_printed ^from <ca>
               ^to <cc> ^distance <d32>))

; *******************************************

(p @_final_print
  (@_#_border ^status printed)
  (%_@_border ^status printed)
  (@_trip ^status not_printed ^from <c1>
                              ^to <c2>)
  -(@_border_trip ^status modified)
  -->
  (write R13: <c1> <c2>)
  (modify 3 ^status printed))

; *******************************************

(p @_printing_done
  (inner_trips_writing ^status undone
                       ^state @)
```

```
  (@_#_border ^status printed)
  (%_@_border ^status printed)
  -(@_trip ^status not_printed)
  -->
  (modify 1 ^status done))

; ******************************************
; Make border crossing

(p @_#_make_crossing
  (@_#_border ^status undone)
  (#_border_trip ^position begin ^from <ca>
                 ^to <cb> ^distance <dab>)
  (@_border_trip ^position end ^from <c2>
                 ^to <c3> ^distance <d23>)
  (@_#_distance ^from <c3> ^to <ca>
                          ^value <d3a>)
  -(#_border_trip ^to <ca>)
  -(@_border_trip ^from <c3>)
  -->
  (modify 1 ^status done)
  (make @_#_crossing ^from_status generated
         ^to_status generated ^from_city <c3>
         ^to_city   <ca> ^distance <d3a>))

; ******************************************
;

(p @_#_make_optim_candidate_from
  (@_#_crossing ^from_status generated
               ^from_city  <c3> ^to_city <ca>
               ^distance <d3a>)
  (@_border_trip ^from <c2> ^to <c3>)
  (@_border_trip ^from <c1> ^to <c2>
                           ^distance <d12>)
  (@_border_distance ^from <c1> ^to <c3>
                   ^value <d13>)
  (@_#_distance ^from <c2> ^to <ca>
                          ^value <d2a>)
  -->
  (modify 1 ^from_status optimized)
  (make @_#_optim_from ^from_city <c3>
        ^to_city <ca> ^fitness
    (compute (<d12>+(<d3a>-(<d13>+<d2a>))))))

; ******************************************
;
```

```
(p @_#_make_optim_candidate_to
  (@_#_crossing ^to_status generated
                ^from_city <c3> ^to_city <ca>
                ^distance <d3a>)
  (#_border_trip ^from <ca> ^to <cb>)
  (#_border_trip ^from <cb> ^to <cc>
                 ^distance <dbc>)
  (#_border_distance ^from <ca> ^to <cc>
                     ^value <dac>)
  (@_#_distance ^from <c3> ^to <cb>
                ^value <d3b>)
  -->
  (modify 1 ^to_status optimized)
  (make @_#_optim_to ^from_city <c3>
                     ^to_city   <ca> ^fitness
    (compute (<dbc>+(<d3a>-(<d3b>+<dac>))))))

; ******************************************
;

(p @_#_eliminate_negative_optim_from
  (@_#_optim_from ^fitness {<f> < 0})
  -->
  (remove 1))

(p @_#_eliminate_negative_optim_to
  (@_#_optim_to ^fitness {<f> < 0})
  -->
  (remove 1))

; ******************************************
;

(p @_#_optimize_connection_from
  (@_#_crossing ^from_city <c3>
                ^to_city <ca>)
  (@_#_optim_from ^from_city <c3>
            ^to_city <ca> ^fitness {<f> > 0})
  (@_border_trip ^from <c2>
                 ^to <c3> ^distance <d23>)
  (@_border_trip ^from <c1> ^to <c2>)
  (@_border_distance ^from <c1> ^to <c3>
                     ^value <d13>)
  (@_#_distance ^from <c2> ^to <ca>
                ^value <d2a>)
  -(@_#_optim_to ^from_city <c3>
                ^to_city <ca> ^fitness > <f>)
  -->
```

```
  (remove 1)
  (remove 2)
  (remove 3)
  (remove 4)
  (make @_border_trip ^status modified
         ^from <c1> ^to <c3> ^distance <d13>)
  (make @_border_trip ^status modified
         ^from <c3> ^to <c2> ^distance <d23>)
  (make @_#_crossing ^from_status optimized
        ^to_status generated ^from_city <c2>
        ^to_city   <ca> ^distance <d2a>))

; ******************************************
;

(p @_#_remove_invalid_optim_from
  (@_#_optim_from ^from_city <c3>
                  ^to_city <ca>)
  -(@_#_crossing ^from_city <c3>
                 ^to_city   <ca>)
  -->
  (remove 1))

(p @_#_remove_invalid_optim_to
  (@_#_optim_to ^from_city <c3>
                ^to_city <ca>)
  -(@_#_crossing ^from_city <c3>
                 ^to_city <ca>)
  -->
  (remove 1))

; ******************************************

(p @_#_optimize_border_to
  (@_#_crossing ^from_city <c3>
                ^to_city <ca>)
  (@_#_optim_to ^from_city <c3>
            ^to_city <ca> ^fitness {<f> > 0})
  (#_border_trip ^from <ca> ^to <cb>
                           ^distance <dab>)
  (#_border_trip ^from <cb> ^to <cc>)
  (#_border_distance ^from <ca> ^to <cc>
                               ^value <dac>)
  (@_#_distance ^from <c3> ^to <cb>
                          ^value <d3b>)
  -(@_#_optim_from ^from_city <c3>
                ^to_city <ca> ^fitness > <f>)
  -->
```

```
  (remove 1)
  (remove 2)
  (remove 3)
  (remove 4)
  (make #_border_trip ^status modified
        ^from <cb> ^to <ca> ^distance <dab>)
  (make #_border_trip ^status modified
        ^from <ca>  ^to <cc> ^distance <dac>)
  (make @_#_crossing ^from_status generated
        ^to_status optimized ^from_city <c3>
        ^to_city <cb> ^distance <d3b>))

; ******************************************

(p @_#_final_print
 (@_#_border ^status done)
 (@_#_crossing ^from_status optimized
    ^to_status optimized ^from_city <c1>
    ^to_city <c2>)
 -(@_#_crossing ^from_status generated)
 -(@_#_crossing ^to_status generated)
 -(@_#_optim_from)
 -(@_#_optim_to)
 -->
 (write <c1> <c2>)
 (modify 1 ^status printed))
```