PROCEEDINGS

Southern African Computer Symposium 1991

6th de Rekenaarsimposium van Suider Afrika 1991

DE OVERBERGER HOTEL, CALEDON

2 - 3 JULY 1991

SPONSORED BY

ISM

FRD

GENMIN

EDITED BY

M H Linck

DEPARTMENT OF COMPUTER SCIENCE • UNIVERSITY OF CAPE TOWN

PROCEEDINGS / KONGRESOPSOMMINGS

6th
SOUTHERN AFRICAN COMPUTER SYMPOSIUM

6de
SUIDELIKE-AFRIKAANSE REKENAARSIMPOSIUM

De Overberger Hotel, Caledon

2 - 3 JULY 1991

SPONSORED by

ISM
FRD
GENMIN

EDITED by

M H LINCK

Department of Computer Science

University of Cape Town

TABLE OF CONTENTS

FOREWORD

The 6th Computer Symposium, organised under the auspices of SAICS, carries on the tradition of providing an opportunity for the South African scientific computing community to present research material to their peers.

It was heartening that 31 papers were offered for consideration. As before all these papers were refereed. Thereafter a selection committee chose 21 for presentation at the Symposium.

Several new dimensions are present in the 1991 symposium:

* The Symposium has been arranged for the day immediately after the SACLA conference.

* It is being run over only 1 day in contrast to the 2-3 days of previous symposia.

* I believe that it is first time that a Symposium has been held outside of the Transvaal.

* Over 85 people will be attending. Nearly all will have attended both events.

* A Sponsorship package for both SACLA and the Research Symposium was obtained. (This led to reduced hotel costs compared to previous symposia)

A major expense is the production of the Proceedings of the Symposium. To ensure financial soundness authors have had to pay the page charge of R20 per page.

A thought for the future would be consideration of a poster session at the Symposium. This could provide an alternative approach to presenting ideas or work.

I would sincerely hope that the twinning of SACLA and the Research Symposium is considered successful enough for this combination survive. As to whether a Research Symposium should be run each year after SACLA, or only every second year, is a matter of need and taste.

A challenge for the future is to encourage an even greater number of MSc & PhD students to attend the Symposium. Unlike this year, I would recommend that they be accommodated at the same cost as everyone else. Only if it is financially necessary should the sponsored number of students be limited.

I would like to thank the other members of the organising committee and my colleagues at UCT for all the help that they have given me. A special word of thanks goes to Prof. Pieter Kritzinger who has provided me with invaluable help and ideas throughout the organisation of this 6th Research Symposium.

M H Linck
Symposium Chairman

SYMPOSIUM CHAIRMAN

M H Linck, University of Cape Town

ORGANISING COMMITTEE

D Kourie, Pretoria University.

P S Kritzinger, University of Cape Town.

M H Linck, University of Capè Town.

SPONSORS

ISM

GENMIN

FRD

LIST OF REFEREES FOR 6th RESEARCH SYMPOSIUM

NAME	INSTITUTION
Barnard, E	Pretoria
Becker, Ronnie	UCT
Berman S	UCT
Bishop, Judy	Wits
Berman, Sonia	UCT
Brink, Chris	UCT
Bodde, Ryn	Networks Systems
Bornman, Chris	UNISA
Bruwer, Piet	UOFS
Cherenack, Paul	UCT
Cook Donald	UCT
de Jaeger, Gerhard	UCT
de Villiers, Pieter	Stellenbosch
Ehlers, Elize	RAU
Eloff, Jan	RAU
Finnie, Gavin	Natal
Gaynor, N	AECI
Hutchinson, Andrew	UCT
Jourdan, D	Pretoria
Kourie Derrick	Pretoria
Kritzinger, Pieter	UCT
Krzesinski, Tony	Stellenbosch
Laing, Doug	ISM
Labuschagne, Willem	UNISA
Levy, Dave	Natal

MacGregor, Ken	UCT
Machanick, Philip	Wits
Mattison Keith	UCT
Messerschmidt, Hans	UOFS
Mutch, Laurie	Shell
Neishlos, N	Wits
Oosthuizen, Deon	Pretoria
Peters Joseph	Simon Fraser
Ram, V	Natal, Pmb.
Postma, Stef	Natal, Pmb
Rennhackkamp, Martin	Stellenbosch
Shochot, John	Wits
Silverberg, Roger	Council for Mineral Technology
Smit, Riël	UCT
Smith, Dereck	UCT
Terry, Pat	Rhodes
van den Heever, Roelf	UP
van Zijl, Lynette	Stellenbosch
Venter, Herman	Fort Hare
Victor, Herna	Stellenbosch
von Solms, Basie	RAU
Wagenaar, M	UPE
Wentworth, Peter	Rhodes
Wheeler, Graham	UCT
Wood, Peter	UCT

6TH RESEARCH SYMPOSIUM - 1991

FINAL PROGRAM

TUESDAY 2nd July 1991

10h00 - 13h00 **Registration**

13h00 - 13h50 **PUB LUNCH**

14h00 - 15h30 SESSION 1A

Venue: Hassner

Chairman: Prof Basie von Solms

14h00 - 14h30
"A value can belong to many types. "
B H Venter, University of Fort Hare

14h30 - 15h00
"A Transputer Based Embedded Controller Development System"
M R Webster, R G Harley, D C Levy & D R Woodward, University of Natal

15h00 - 15h30
"Improving a Control and Sequencing Language"
G Smit and C Fair, University of Cape Town

SESSION 1B

Venue: Hassner C

Chairman:Prof Roelf v d Heever

14h00 - 14h30
"Design of an Object Orientated Framework for Optimistic Parallel Simulation on Shared-Memory Computers" P Machanick, University of Witwatersrand

14h30 - 15h00
"Using Statecharts to Design and Specify the GMA Direct-Manipulation User Interface" L van Zijl & D Mitton, University of Stellenbosch

15h00 - 15h30
"Product Form Solutions for Multiserver Centres with Heirarchical Classes of Customers" A Krzesinski, University of Stellenbosch and R Schassberger, Technische Universität Braunschweig

15h30 - 16h00 **TEA**

16h00 - 17h30 SESSION 2A

Venue: Hassner

Chairman: Prof Derrick Kourie

16h00 - 16h30
"A Reusable Kernel for the Development of Control Software" W Fouché and P de Villiers, University of Stellenbosch

16h30 - 17h00
"An Implementation of Linda Tuple Space under the Helios Operating System" P G Clayton, E P Wentworth, G C Wells and F de-Heer-Menlah, Rhodes University

17h00 - 17h30
"The Design and Analysis of Distributed Virtual Memory Consistency Protocols in an Object Orientated Operating System" K MacGregor, University of Cape Town & R Campbell, University of Illinois at Urbana-Champaign

19h30 **PRE-DINNER DRINKS**

20h00 **GALA CAPE DINNER**
(Men: Jackets & ties)

WEDNESDAY 3rd July 1991

7h00 - 8h15 BREAKFAST

8h15 - 9h45 SESSION 3A

Venue: Hassner

Chairman: Assoc Prof P Wood

8h15 - 8h45
"Concurrency Control Mechanisms for Multidatabase Systems" A Deacon, University of Stellenbosch

8h45 - 9h15
"Extending Local Recovery Techniques for Distributed Databases" H L Victor & M H Rennhackkamp, University of Stellenbosch

9h15 - 9h45
"Analysing Routing Strategies in Sporadic Networks" S Melville, University of Natal

SESSION 3B

Venue: Hassner C

Chairman: Prof G Finnie

8h15 - 8h45
The Design of a Speech Synthesis System for Afrikaans" M J Wagener, University of Port Elizabeth

8h45 - 9h15
"Expert Systems for Management Control: A Multiexpert Architecture" V Ram, University of Natal

9h15 - 9h45
"Integrating Simularity-Based and Explanation-Based Learning" G D Oosthuizen and C Avenant, University of Pretoria

9h45 - 10h15 TEA

10h15 - 11h00 SESSION 4

Venue: Hassner

Chairman: Prof P S Kritzinger
Invited paper: E Coffman

11h00 - 11h10 BREAK

11h10 - 12h40 SESSION 5A

Venue: Hassner

Chairman: Prof C Bornman

11h10 - 11h40
"Efficient Evaluation of Regular Path Programs"
P Wood, University of Cape Town

11h40 - 12h10
"Object Orientation in Relational Databases"
M Rennhackkamp, University of Stellenbosch

12h10 - 12h40
"Building a secure database using self-protecting objects" M Olivier and S H von Solms, Rand Afrikaans University

SESSION 5B

Venue: Hassner C

Chairman: Prof A Krzesinski

11h10 - 11h40
"Modelling the Algebra of Weakest Preconditions"
C Brink & I Rewitsky, University of Cape Town

11h40 - 12h10
"A Model Checker for Transition Systems"
P de Villiers, University of Stellenbosch

12h10 - 12h40
"A New Algorithm for Finding an Upper Bound of the Genus of a Graph"
D I Carson and O R Oellermann, University of Natal

12h45-12h55 GENERAL MEETING of RESEARCH SYMPOSIUM ATTENDEES

Venue: Hassner

Chairman: Dr M H Linck

13h00 - 14h00 LUNCH

FINIS 6th COMPUTER SYMPOSIUM

PAPERS

of the

6TH RESEARCH SYMPOSIUM

An Implementation of Linda[1] Tuple Space under the Helios[2] Operating System.

P.G. Clayton*, E.P. Wentworth, G.C. Wells and F.K. de-Heer-Menlah
Department of Computer Science, Rhodes University, Grahamstown, 6140 RSA
Internet: cspc@alpha.ru.ac.za

Abstract

We discuss the implementation of Rhoda, our Linda-like Tuple Space server which runs under the Helios operating system. The approach analyses and partitions tuple space at compile time in order to reduce the run time overhead of tuple matching. The interaction between the concurrent processes and the tuple partitions is used as the basis for distributing the partitions and processes in the network. The paper presents some empirical results and discusses the suitability of the Helios nucleus for supporting the approach.

Keywords: *distributed systems, parallel processing, transputer, Linda, Helios.*

1. Introduction

The Linda programming paradigm is a simple and elegant approach to parallel processing, based on the concept of *generative communication* [GEL 85]. This is a form of communication in which an active message (or tuple) may be converted through process creation and evaluation into a passive value. Linda is not a language per se, it is a small set of control and coordination operations which can be imbedded into a programming language (typically one of the well known imperative programming languages) to introduce or enhance parallel capabilities.

At the center of the Linda programming model is a shared, associative memory called *tuple space* (*TS*). Objects called *tuples* are output to - and input from - *TS* by components of the application program. At the abstract programming level, *TS* is global to all components of a parallel program, even though they might be executing on individual processors which have no physical memory in common. Parallel components of an application program (processes or tasks) never communicate directly with each other, only with *TS*. Consequently, *TS* acts as a decoupling agent. This reduces program complexity by allowing parallel programs to be decoupled both spatially and temporally.

A *tuple* is a sequence of typed (actual or formal) fields, rather similar in concept to a parameter list. In addition to passive data values, the contents of a tuple may be a reference to active executing or executable code. Tuples are selected from *TS* by associative matching.

To communicate with tuple space, Linda provides a set of six primitive operations:

out(t) output tuple *t* to the *TS*

---

[1]Linda is a trademark of Scientific Computing Associates, Inc., New Haven.

[2]Helios is a trademark of Perihelion Software Ltd., Somerset, England.

eval(t) evaluate tuple *t* (This operation is similar to *out* in that it outputs tuple *t* to *TS*, but *t* may be an active tuple whose result is yet to be evaluated.)

in(s) input a tuple *t* from *TS* which matches the template *s* (If no matching tuple is available, the requesting process is suspended until one becomes available. If more than one matching tuple exists in *TS*, an arbitrary matching tuple is returned. The tuple is removed from *TS*.)

rd(s) read a tuple *t* from *TS* which matches the template *s* (*rd* is conceptually very similar to *in*. It returns a copy of a tuple without removing it from *TS*.)

inp(s)
and

rdp(s) similar in function to *in* and *rd*, these operations are predicates which attempt to match a tuple *t* to the template *s*, and return a failure value immediately if no match is found. If the operation succeeds, both a tuple and a success value are returned.

These operators communicate only with *TS*, and none of the high-level system services which distributed operating systems usually superimpose upon their transport layers are provided. A Linda system may make use of the existing low-level transport layer provided by a distributed operating system, or may require a specialized transport layer to be written. In the former case, application programs should be unconcerned about the particular target architecture, and about whether they will run under an operating system or as standalone programs.

A number of informative articles on the use of the Linda approach to parallelism have already appeared in print, some of which are listed among this paper's references [CAR 89a] [CAR 89b] [GEL 88] [AHU 86]. We do not concern ourselves in this paper with presenting a suite of tutorial examples, or with persuading readers of the merits of this programming approach; we concentrate on implementation issues, assuming a rudimentary knowledge of the abstract programming environment presented by the Linda primitives, and a conviction of its value to parallel processing.

This paper represents a status report on an implementation effort underway at Rhodes University to build an efficient, distributed *TS*-manager for transputer-based parallel processing systems in the Helios operating environment. To distinguish the experimental effort at Rhodes University from existing commercially available implementations of Linda, our system is known as Rhoda. For the purpose of this paper, the terms Linda and Rhoda are used interchangeably, although Rhoda is generally used to refer specifically to the Rhodes implementation.

2. An overview of the Rhoda Implementation

A side effect of the high level of decoupling between parallel components of a Linda program is that efficiency becomes more of a concern of the implementation and less of a concern of the application programmer. This places pressure on the developers of a Linda implementation to provide an efficient transport layer which will allow *TS* to be simultaneously visible to all components of the application program. A range of strategies can be used to implement a global *TS* in a parallel processing environment in which processors do not have a shared physical memory. At one extreme, *TS* could be stored at a dedicated central node which is accessed via a transparent message routing system. Even if run-time hashing is used to improve search

performance in this approach, delays caused by message routing can degrade the performance of the system, and a single centralized *TS*-manager can become a bottleneck which impedes massive parallelism. At the opposite end of the implementation spectrum, *TS* could be replicated in each processing node, and local *TS*-managers could transparently propagate changes through the network. A major encumbrance to this approach is the provision of a locking mechanism which ensures that program components wishing to remove tuples from *TS* are given exclusive delete access.

The Rhoda implementation under Helios uses a centralized *TS* model, but partitions *TS* with the view to reducing the run time matching overheads of Linda operations, and so that distributed *TS*-managers can be used to control a small (possibly localized) group of related tuples. A partitioned *TS* is in contrast to the Linda programming assumption of a single shared *TS*. This section provides a brief overview of the Rhoda compilation path, which adds additional housekeeping information to source programs to enable them to work with the partitioned model described in the remainder of the paper.

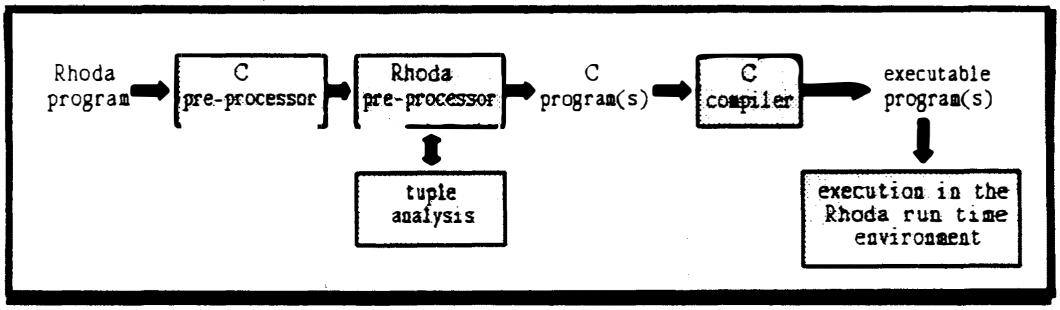

Figure 1 - Structure of the Rhoda compilation path.

Figure 1 depicts the compilation phases present in the Rhoda compiler. C is currently used as the host language for Rhoda. Apart from the normal C pre-processor, Rhoda makes use of a second pre-processor to compile and pass a list of all tuple operations, and the program components which issue them, to a tuple analysis module. This module analyzes *TS* interaction with components of the application program, to divide tuples into groups based on their structure, and to suggest an appropriate placement strategy for tuple groups and application program components in the processor network. The grouping of tuples is an integral feature of the Rhoda implementation, and is described in more detail below. By grouping tuples at compile time, a substantial matching overhead is avoided at run time. Distinct tuple groups also facilitate the distribution of *TS* in the distributed memory environment. The initial placement strategy of the Rhoda system divides a task force (application program components and *TS*-managers) into appropriate process clusters for placement on the processor network, in positions which will incur a relatively low inter-cluster communication cost. This aspect of *TS* analysis is described in more detail by de-Heer-Menlah [DHM 90].

The tuple groupings determined by the analysis module are used by the Rhoda pre-processor to translate ideal Linda syntax into concrete C syntax which opens, closes, and addresses file-like tuple partitions. A Rhoda program usually contains a number of components (for example, a master process and a worker process), which must all be present during tuple analysis. The output

of the pre-processor stage is a series of C programs, one for each unique parallel component of the original source.

3. Partitioning tuple space

The syntax for tuple fields makes provision for actual fields in the form of constant values or run-time expressions, and for formal fields denoted by program variables which are preceded by the "?" character. The Linda input primitives provide tuple templates against which tuples placed in *TS* by output primitives are compared. It is common practice for Linda programmers to use a constant valued field to ensure a correct tuple matching. For example, the initial field of a tuple is frequently a string literal. The matching process is potentially a computationally expensive operation, and is an area in which efficient implementation is a crucial issue.

The tuple templates of Linda operations are matched by associatively searching tuples within *TS* which have the same structure. Examples of syntactically correct, matching Linda primitive operations might be:

out("element", 3, 4, value) in("element", i+1, j, ?result)

If the variables *value*, *i*, *j*, and *result* were all declared to be of the same type (integers for example), then the two tuples manipulated by the above *in* and *out* operations would be regarded as having the same *structure*, *viz.* a string constant followed by three integer fields. The actual expressions (*value*, *i+1*, and *result*) would contribute their current run-time values to the *out* operation's tuple and the *in* operation's template. The formal field (*result* in the template used for the *in* operation in this example) would return the *value* of a tuple whose first three fields match those of the template. For example, if the values of *i* and *j* were 2 and 4 respectively, and the tuple *("element", 3, 4, 12)* were present in *TS*, then *result* would have the value 12 after execution of the *in* operation.

It is possible to detect at compile time that a Linda input *in("row", ?i)* could be matched to any of the following tuples

("row", 4) ("row", 10) ("row", 500)

with the consequent actual to formal assignment for the variable *i*.

It is likewise clear at compile time that the template *("result", ?i)* will not match any of the following tuples, no matter what the run time values of variables are, because the type, order, or number of fields differ.

("row", 6.42) (j, "row") ("matrix size", 50, 20)

Nor will it match a tuple such as *("col", 4)* whose type, order, and number of fields agree, because the value of the compile time string literal field of the tuple and template differ.

Since operations on one tuple group can never match tuples in another group, the partitioning of

tuples into disjoint groups at compile time can be done safely. Tuple operations can first be coarsely classified into mutually exclusive groups based on their field structure (type, order, and number of fields). A subsequent finer partitioning can be done based on field information; tuples having the same field structure, but different compile time constant values in a particular field, cannot be matched.

Once compile time constants have been examined and tuple groups have been formed, the constant values are no longer of any use since all tuples (and tuple templates) within a particular group will have identical constant values in their common constant fields. Discarding such constant fields is a further compile time optimization. For example, Linda operations which refer to the tuples

("row", i, j) ("row", i+1, j+1) ("row", ?m, ?n)

will be modified to calls to the same *"row"* tuple group using the tuples

(i, j) (i+1, j+1) (?m, ?n)

Efficient searching and matching strategies can now be devised for particular tuple groups. Taken together, the dramatic reduction in the scope of a tuple search and the reduction in the number of (mostly string) fields provide a major improvement in the run time overhead of tuple matching. Analysis of the actual to formal relationships of the corresponding fields of a tuple template and its *TS* group can lead to further efficiencies in run-time matching. Zenith [ZEN 90] suggests a number of instances in which a general tuple matching algorithm can be reduced to a far simpler operation.

It is possible to take the *TS* analysis further by considering which components of the application program make use of each tuple group. This provides information for the placement of *TS* groups relative to the program components which they serve in the processor network, and allows an hierarchical *TS* dependency structure to be built, thereby facilitating an hierarchical naming scheme for the distributed *TS*. For example, a Linda application program comprising three parallel component processes, P_1, P_2, and P_3 coordinates its parallel activity using three different tuple structures which can be grouped at compile time into three independent tuple groups. All three processes make use of tuple

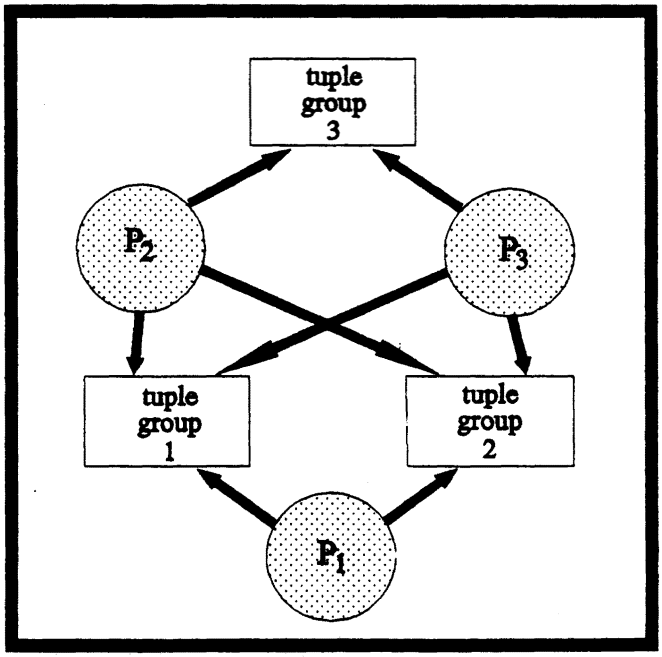

Figure 2 - Three parallel application processes referencing three tuple groups.

groups 1 and 2, while only P_2 and P_3 make use of tuple group 3. Figure 2 demonstrates the interaction of component processes and *TS* groups for this example, and figure 3 shows the hierarchical relationship which results from the partitioning of *TS*.

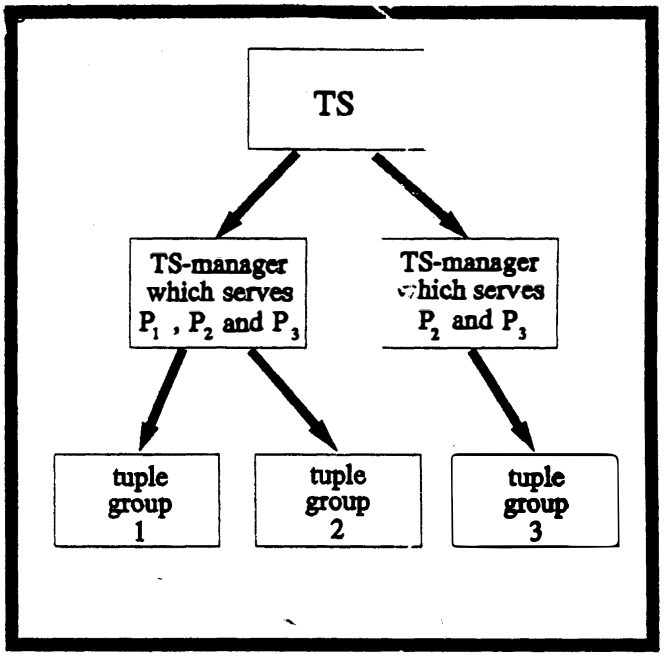

Figure 3 - The hierarchical relationship of TS.

4. The Helios environment

Helios [PER 89] is a UNIX[3]-like[4], distributed, parallel operating system. The Helios nucleus, which must be present on all Helios processors, provides a small kernel (for managing message passing, hardware resources, and list handling) and a number of basic *servers* which integrate the processor into the global environment. Helios servers are based on the conventional *client/server* model, in which a server task manages a resource on behalf of its clients. The minimum set of servers required by a Helios processor includes a loader, a processor manager for managing the computing resources of the processor and for responding to requests to access executing tasks, and a number of I/O controller (IOC) processes. Additional operating system servers might be loaded on particular processors of the network to support specific facilities. These include the window server, the disk server, the RS232 server, the console server, the network server responsible for distributing and controlling the nucleus, and so on. Most importantly, Helios provides a server library facility which can be used to implement additional servers for the system using a standardized general server protocol.

To facilitate communication between distributed tasks, the process manager of the Helios nucleus spawns an IOC process for each new task, which acts as the task's intermediary with the rest of the system. The IOCs on one processor route requests to named objects on behalf of their tasks by referencing a central name table. If a name is present in the table, the IOC passes the request directly to the server whose port is represented in the entry, if not, a distributed search is initiated. Provided the name exists elsewhere, an entry is installed in the name table so that subsequent uses need not cause a search. Each physical link of the processor also has an IOC, responsible for handling distributed searches and requests from remote tasks to local servers.

Helios supports an hierarchical naming scheme for all objects in the network. Each sub-network (or cluster) is given a unique name, and the names of objects within sub-networks (processors, files, file systems, servers, tasks, and so on) must not conflict when they are identified by their

---

[3]UNIX is a trademark of AT&T.

[4]The Helios operating system includes a UNIX-compatible library, which is based on the POSIX standard [IEE 88].

position in the network hierarchy. All objects in Helios present a directory interface through which any information specific to the object may be examined and manipulated. This form of network addressing is a logical extension of the conventional hierarchical file system adopted by many operating systems. Most Helios commands which access the hierarchical directory structure are generic utilities which do not differentiate between different types of object in the hierarchy. Figure 4 is an example of the hierarchical naming scheme presented for a sub-network. In this example, the cluster comprises three processors (*00, 01,* and *02*) and an I/O server. The Helios nucleus on each processor comprises a *tasks* directory, a number of link IOCs, and so on. Objects within the *tasks* directory are the currently active tasks on that processor.

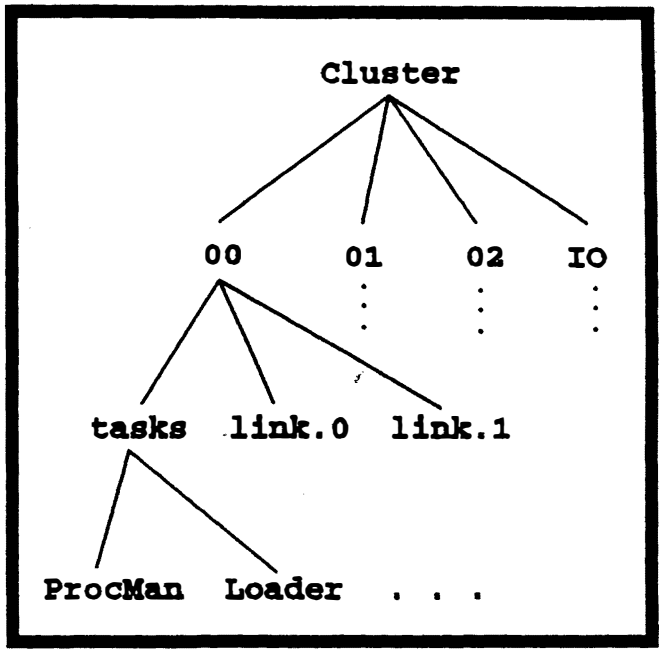

Figure 4 - Helios objects represented as a hierarchical directory.

Network naming is a totally distributed service in Helios, and a distributed name server is at the heart of the naming scheme. It provides an hierarchy of names for an otherwise arbitrary topology structure.

Helios servers may be localized or distributed. All servers adhere to the same *general server protocol* (GSP). They are written as a set of calls to a distributed server library, plus a set of application specific functions [GAR 89]. The server library provides support for a message decoder and despatcher, which waits for messages on a specified port, validates them as GSP messages, and forks a worker process to execute a service procedure. The forking of a service procedure is an important aspect of Helios's support for distributed servers. The server essentially consists of the despatcher process until such time as a request arrives from the server's request port (looked up in the name table by the name server on behalf of a client process). To handle the request, the despatcher process spawns a separate process to execute the required function. This happens for each request. Normally, this process returns a reply at the end of the desired service and terminates. However, if the function performed by the spawned service process is an *open* operation (as in "open a file"), the service process remains active after a reply has been sent, and acts as a proxy server for any stream messages which are directed to it, until it is *closed.*

5. The tuple space server under Helios

The hierarchical naming structure of Helios provides an ideal support environment for a *TS* which can be grouped in such a way as to expose hierarchical relationships. *TS* in the Helios-Rhoda

system has been implemented along similar lines to a directory based file server, in which each "file" corresponds to a tuple group capable of manipulating streams of tuples with the same *type signature* (as grouped by the tuple analysis module in the Rhoda pre-processor). By adopting the Helios environment, we gain directory and sub-directory structures, and their concomitant protection mechanisms, at no additional cost to the implementation; they are already part of the existing Helios server protocol and libraries. *TS* is implemented as a Helios server, using the standard GSP. The Rhoda *TS* server integrates very smoothly with rest of the Helios system because it honours this protocol, and the generic utilities which operate on other Helios objects are able to operate on *TS* structures as well. Each tuple group falls under the control of a *TS*-manager, but different tuple groups might be placed under the control of different *TS*-managers distributed across the network.

The Helios strategy of routing all GSP requests to a single port, and then spawning (by way of the despatcher) independent processes to service each of them, enables several clients to access the same server concurrently. In the *TS* server, such GSP protocols are occasional events, which open a tuple group and create a proxy process within the server to manage access to the tuple group on behalf of a particular client. Thereafter, Linda operations are reduced to direct communications between the client and the proxy process.

All client processes which produce or consume tuples with a particular type signature will open the same tuple group. To gain access to a tuple group, a client process must declare a tuple group descriptor, specifying a name for the group and a type signature for tuples which conform to the group. Thereafter, it is able to open the tuple group, use it, and close it again, simply by making appropriate *TS* server calls and supplying the appropriate name of the tuple group along with each such operation. The pre-processor prefixes each *TS* operation in the source code with a tuple group descriptor for this purpose. The first reference to a *TS* server initiates a dynamic network search and establishes a connection path between the client and its proxy service process, enabling the two to exchange messages without regard to the system topology. Thereafter the client process has a point-to-point virtual link to a dedicated proxy process, which manipulates the tuple group on its behalf, until it requests a close operation, at which time the proxy process terminates. Since several clients are able to access the same tuple group simultaneously, the *TS* proxy processes assume the responsibility for locking the tuple group and coordinating requests during operations which update the group.

Each tuple group within the *TS* server is a data structure which contains control information such as its name, size, number of clients, protection attributes, mode information, parent directory, a locking semaphore to ensure exclusive update access, and so on. It also keeps track of current tuple values, and keeps a queue of blocked clients together with their transaction templates.

To handle the blocking semantics of the Linda primitives *rd* and *in*, proxy server processes are suspended until a suitable tuple arrives. This has the effect of suspending the client as well, while it awaits a reply from the server. A *TS* proxy server handles an unmatched request by queueing it, along with a semaphore, in the waiting queue for the tuple group it supports. The proxy then suspends itself by waiting on the semaphore. Each time a new tuple arrives for a particular tuple group as a result of an output operation, the waiting queue for that group is searched, comparing the new tuple to pending requests. To satisfy the different semantics of the Linda *in* and *rd*

operations, a pass is made through the queue, locating each matching *rd* transaction which can be completed, up to the first matching *in* transaction. The *in* transaction must also be completed, and will consume the new tuple. If there is no pending *in* operation, the tuple is added to *TS* in the normal way. Completion of a pending transaction is achieved by allowing the output primitive to complete its transaction, and then waking those proxy processes whose outstanding transactions can be satisfied.

Figure 5 illustrates the integration of the *TS* server into the hierarchical Helios naming structure. In this example, a *TS* server has been initiated on processor *00*, and one or more client processes have opened two tuple groups, named *rows* and *results*. Client processes residing anywhere within the network are able to open either of these tuple groups, and a proxy service process will be spawned on processor 00 (within this *TS* server) for each such request. So, process *A*, executing on processor *02*, which uses both of these tuple groups, will cause two independent proxy processes to be spawned within this *TS* server. Process *B*, executing on processor *00*, which uses tuple group *results*, will cause yet another proxy process to be spawned. There might well be additional *TS* servers residing on other processors in the network and managing access to other tuple groups, provided their names do not conflict with the name of this server in the naming hierarchy. Processes *A* and *B* could well be making use of these additional servers as well.

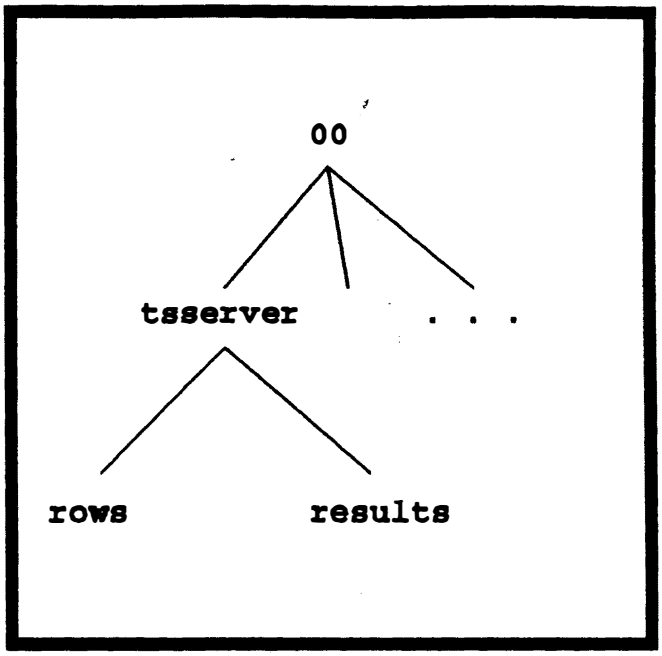

Figure 5 - A TS server in the Helios hierarchical structure.

The Rhoda system makes use of the Helios processor manager to implement the *eval* primitive operation. The processor manager is a server which is present as part of the essential nucleus on all Helios processors. It sees to the creation and management of tasks on that processor, and is able to load and execute programs on behalf of clients executing on remote processors.

During the Rhoda pre-processing phase, each source function that is invoked by *eval* is transformed into a free-standing executable program by encapsulating it in a suitable code skeleton. Since an *in* or *rd* template can never match an active tuple, tuples generated by *eval* operations will be placed into their own active tuple groups. When a *TS* server is invoked, it must be supplied with the names of the processors on which it may execute active tuples. The *TS* server spawns a manager task for each such target processor, and establishes a link to that processor's processor manager. These manager tasks are responsible for monitoring the *TS* server's active tuple group, and remotely invoking processes to evaluate tuples when necessary.

Remote program invokation is a relatively expensive operation, particularly if the executable code has to be fetched from a central filing system[5]. The Rhoda implementation alleviates this overhead by modifying the skeleton that encapsulates *eval*-ed functions so that, once invoked, they repeatedly fetch and execute active tuples until a request for an active tuple matching their particular type fails. This has the same effect as reinvoking the function for every tuple of that type, but is considerably more efficient.

For monitoring purposes, each Rhoda *TS* server also provides statistical information, which appears to a client process wishing to monitor *TS* as just another set of tuples, which can always be read (i.e. they are created "on the fly" when they are requested).

6. Observations and conclusions

A number of desirable qualities are present in the Helios-Rhoda implementation:

The system is able to execute on any transputer network with an arbitrary topology.

With *TS* implemented as a distributed server (essentially present as part of the system nucleus on all processing nodes) no processors are dedicated to supporting *TS*, or are excluded by the presence of a *TS*-manager from supporting part of the application task force.

The division of *TS* into individually addressable tuple-groups reduces the potentially expensive operation of associative matching to a far simpler operation, and provides a natural mechanism for partitioning *TS* space into distributable sub-spaces.

The hierarchical structure of *TS* partitions and the inheritance of the normal filing system security mechanisms allow concepts such as private tuple spaces and tuple spaces within tuple spaces [LEL 90] to be exploited.

Our approach differs from the Yale precompiler in that we view a parallel job as a single program comprising a number of sections. Our system requires that all the components are compiled and analysed together. Once the tuples have been partitioned and common fields have been factored out, the discarded information can no longer be retrieved unless the whole job is recompiled. By contrast, the Yale effort [CAR 90] supports separate compilation, and provides a pre-linking stage which analyses and specializes the tuple space access procedures. Their goal is to optimize the accesses, but to carry enough run-time information so that the original unoptimized data can be reconstructed. This will allow new participants to join the computation dynamically.

Some aspects of the system are still under development, notably the distributed *TS*-managers, and the system has only been tested with relatively small numbers of processing nodes (up to 16). The performance we have observed is encouraging. We have used the Rhoda implementation to support a number of parallel algorithms, including a state-space search and a 2-D FFT transformation. The Rhoda system is also being used as a platform for implementing a parallel version of a popular scientific and engineering matrix manipulation package[WEN 91], and as a means of parallelizing existing animation and graphics rendering applications. Figure 6 shows the almost linear improvement in speed obtained for an existing ray tracing application moved onto the Rhoda system, and for a queens placement algorithm, as the number of worker processes is

[5]This can be improved by program caching.

increased.

We have been happy with the performance of our *TS* transport layer to date, and our experience also confirms the claims in the literature [CAR 89a] [CAR 89b] [GEL 88] [AHU 86] that it is easier to write parallel programs using the Linda model than it is with traditional tools. The high level abstract programming environment of the Linda operators has enabled us to think about parallelism in ways which were not always obvious when we were constrained by the concepts of semaphores and point-to-point messages.

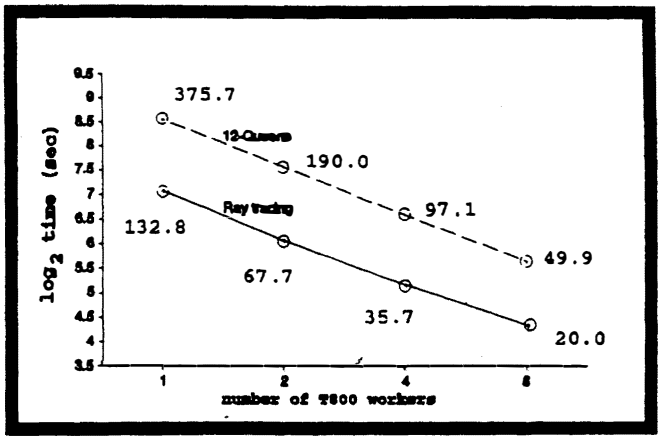

Figure 6 - Performance of ray tracing and queen placement algorithms running on the Rhoda system, as the number of worker processes is increased.

From an implementor's point of view, Helios encourages a client/server programming model, and this has had a definite influence on our design. Without the presence of simple operating system mechanisms for creating and controlling tasks, topology independent message routing, and support for hierarchies of structures, we would not have envisaged the system as it is currently structured. This conviction is strengthened by our experience of designing a previous *TS* prototype on a network of PC's running DOS, an environment which constrained our thinking severely. Moreover, Helios's Unix-like development environment and the ANSI-C language support have isolated us from the awkwardness of the transputer's underlying RISC-like architecture, and this has improved our productivity. It is unlikely that we would have made similar progress using TDS and Occam, the customary systems programming tools used with transputers.

References

[AHU 86] Ahuja, S., Carriero, N. and Gelernter, D. (1986), Domestic Parallelism - Linda and Friends, *Computer (USA)*, 19(8), 26-34.

[CAR 89a] Carriero, N. and Gelernter, D. (1989), How to Write Parallel Programs: A Guide to the Perplexed, *ACM Comp. Surveys*, 21(3), 323-357.

[CAR 89b] Carriero, N. and Gelernter, D. (1989), Linda in Context, *Comm. ACM*, 32(4), 444-458.

[CAR 90] Carriero, N. and Gelernter, D. (1990), Tuple Analyses and Partial Evaluation Strategies in the Linda Precompiler, in Languages and Compilers for Parallel Computing, Eds. Gelernter, Nicolau and Padua, Pitman, 114-125.

[DHM 90] de-Heer-Menlah, F.K. (1990), *Analyzing Communication Flow and Process Placement in Linda Programs on Transputers*, Technical Document 90/4, Dept. Computer Science, Rhodes University.

[GAR 89] Garnett, N. (1989), *Writing a Helios Server*, Technical Report No. 8, Perihelion

Software Ltd.

[GEL 85] Gelernter, D. (1985), Generative Communication in Linda, *ACM TOPLAS*, 7(1), 80-112.

[GEL 88] Gelernter, D. (1988), Getting the Job Done, *Byte*, 13(12), 301-308.

[IEE 88] IEEE (1988), *The Proposed POSIX Standard*, IEEE Std 1003.1.

[LEL 90] Leler, W.M. (1990), Linda meets Unix, *Computer*, 23(4), 43-54.

[PER 89] Perihelion Software Ltd. (1989), *The Helios Operating System*, Prentice-Hall, London.

[WEN 91] Wentworth, E.P., Clayton, P.G., (1991) Matlab Meets Rhoda, IEEE Symposium on Parallel Processing - Design and Implementation, CSIR June 1991.

[ZEN 90] Zenith, S.E. (1990), *Linda Coordination Language; Subsystem Kernel Architecture (on Transputers)*, YALE Research Report DCS/RR-794.