Parallel Graph Computation Using Autonomous Objects

Munehiro Fukuda

Institute of
Information Sciences & Electronics
University of Tsukuba, Japan
e-mail: fukuda@is.tsukuba.ac.jp

Lubomir F. Bic

Department of
Information & Computer Science
University of California, Irvine
e-mail: bic@ics.uci.edu

Technical Report ISE-TR-98-152

June 1, 1998

Abstract

Many distributed computing applications are based on a graph whose nodes and edges correspond to computations and communications. The self-navigation of autonomous objects is attractive to solve such graph-based problems in parallel, because of (1) the inherent parallelism by object propagations over network and (2) the encapsulation of graph algorithms inside objects. However, most autonomous-objects-based systems have put their paramount focus on new computation paradigms or aimed at network service tasks not requiring the performance. Among them, MESSENGERS is the first autonomous-objects-based system aiming at *general-purpose* distributed computing, thus addressing the performance issues. It has demonstrated its competitive performance with other message-passing systems in terms of coarse-grain computations, which are yet too large to solve fundamental graph problems in parallel. However, the software demands for large-scaled graph generations and analyses are increasing drastically. We expect that it is feasible to apply autonomous objects to parallel graph computation on workstation cluster systems with high-speed switchs or distributed-memory supercomputers, using the following schemes: (1) applying a thread migration technology, (2) reengineering navigational autonomy appropriate to graph computation, and (3) realizing static and dynamic load balancing of logical nodes. This report presents those three schemes in detail.

Contents

1 Introduction 4

2 Overview of MESSENGERS 6

2.1 Execution Model 6

2.2 Language Specification 6

2.3 Node Mapping Algorithm 9

2.4 Data Structure of Messengers and Logical Nodes 11

3 Using Threads 11

3.1 Why Threads 11

3.2 Self-Migrating Threads 13

3.2.1 Assumptions 13

3.2.2 Design Strategy 14

3.3 Implementation Details 15

3.3.1 System Components 15

3.3.2 Pointers 16

3.3.3 Migration 17

3.3.4 Thread Scheduling 18

3.4 Related Works 19

4 Navigational Calculus for Graph Constructions 21

4.1 Problems in Current Logical Network Constructions 21

4.2 Language Specification 24

4.2.1 Definitions 24

4.2.2 Creation 28

4.2.3 Hop 28
4.2.4 Jump 28
4.2.5 Change 29
4.2.6 Example 29
4.2.7 Network Environment Information 30
4.3 Navigations Using New Calculus 31
4.4 Related Works 31

5 Load Balancing **33**
5.1 Research Challenges 34
5.1.1 IP-oriented Node Addressing 34
5.1.2 Implicit Node Mapping 34
5.1.3 User-Oriented Node Addressing 35
5.2 Node Mapping 36
5.2.1 Node creations with a user-defined address 36
5.2.2 Node creations with an implicit mapping 37
5.3 Node Remapping 38
5.4 Related Works 39

6 Conclusion **41**

1 Introduction

The structure of many distributed problems is a graph whose nodes and edges correspond to computations and communications. Thus, one of the requirements to distributed systems is their ability of constructing graphs over the systems and executing fundamental graph algorithms in parallel.

Recently, mobile agents or autonomous objects are receiving popularity due to their navigational autonomy which enables them to roam over physical network and perform tasks at each node they visit. The navigational autonomy is also attractive to solve graph-based problems in parallel. This is because (1) objects propagate themselves through graphs mapped to LAN or WAN, which in turn explores inherent parallelism, and (2) they can better encapsulate algorithms, thus narrowing the semantic gap between distributed algorithms and their implementation.

Several computational paradigms based on autonomous messages/objects have been proposed, focusing on parallel graph-based computations. Echo [Cha82] is the paradigm that explores a given graph's property by propagating a wave of autonomous messages from a client node. These messages repeatedly navigate themselves along the incident edges until they visit all nodes. BPEM [BL87] views the knowledge semantic net as a graph and each query as a graph template. The autonomous objects carry and use the template as a "road map" to navigate the underlying graph. Wave [SB94] permits its objects to perform arbitrary computations, replications, and network propagations. Wave is the one actually implemented, emphasizing on its unique computational paradigm. It consists of daemon processes interpreting objects and having individual processes perform tasks, and therefore the performance was out of consideration.

"Intelligent" mobile agents are another related work based on autonomous objects. However, most systems developed so far, have focused on network service tasks irrelevant to performance, such as information retrieval and electronic commerce over WAN's. While they claim their potential performance advantages such as the inherent parallelism and the reduction of network communication, their current speed of interpretation and task invocation is still too slow to address the performance issues. In addition, they navigate directly in the physical network but do not have capability of creating logical graphs.

MESSENGERS [BFD96] is an autonomous-objects-based system developed at University of California, Irvine, not only permitting objects to construct logical graphs dynamically but also addressing the performance, thus aiming at *general-purpose* distributed computing on a LAN. It is the first system to provide objects with a virtual time environment. MESSENGERS has demonstrated its competitive performance with and better programmability than message-passing systems such as PVM in several

computation-oriented applications: matrix multiplication, convex hull, and madelbrot generation [Fuk97]. However, the granularity of computation is still coarse, requiring approximately 10,000 floating-point operations per each network navigation, due to the nature of interpretation, the simple logical network implementation, and slow physical communication. Many fundamental graph algorithms such as shortest path and minimum spanning tree searches do not involve such a large amount of computation at each node. Thus, only a limited number of graph-based applications obtain the advantage of parallel executions from MESSENGERS.

However, practical graph-based applications may sometimes need extremely large graphs, which are sometimes impossible to be constructed on a single workstation only. For instance, the analysis of stochastic Petri Nets needs the deduction of all reachable global states from a given initial state. The graph obtained from this state deduction is called a reachability graph and normally too large to be stored in a single workstation. Another example is the individual-based simulation that computes interactions among numerous different simulation entities behaving in a virtual world normally represented by a large graph [LS95]. Even the growth of large graphs is the final goal of applications. One such example is simulating the growing process of living organisms [PL96].

One of the recent popular system configurations is the workstation cluster connecting a large number of workstations with a high-speed interconnection switch, thus ameliorating physical communication cost. Also, distributed-memory supercomputers are available for MIMD-based parallel computing.

From these software demands and hardware improvements, we feel the necessity and possibility of constructing a new distributed environment that achieves parallel computing of graph-based applications, as maintaining the better programmability of autonomous objects. The performance is our main concern and addressed by the following three research challenges:

1. Replacing interpretive objects with threads that achieve full native-mode execution
2. Reengineering the navigational autonomy for fast graph construction and navigation
3. Realizing new schemes for logical node distribution and remapping

The rest of this report is organized as follows: Section 2 gives the overview of MESSENGERS. Section 3 discusses how autonomous objects are managed with thread migrations. Section 4 introduces a new navigational calculus suitable for graph-based computations. Section 5 discusses a load balancing scheme which both initially distributes and dynamically remaps logical nodes over the system. Finally, Section 6

clarifies the implementation plan and concludes the discussion.

2 Overview of MESSENGERS

We start with the overview of MESSENGERS that is a basis for further discussions on three research challenges: thread migration, navigational calculus, and load balancing for parallel graph computation.

2.1 Execution Model

To allow Messengers [1] to navigate autonomously through the network and carry out their tasks, the MESSENGERS system is implemented as a collection of daemons instantiated on all physical nodes participating in the distributed computation. A daemon's task is to continuously receive Messengers arriving from other daemons, interpret their behaviors, described as programs carried as part of each Messenger, and send them on to their next destinations as dictated by their behaviors.

The MESSENGERS system involves three layers of network. The lowest is the *physical network* (a LAN or WAN), which constitutes the underlying computational resource. Superimposed on the physical layer is the *daemon network* which involves selecting an arbitrary subset of the physical nodes to run the interpreter daemon on and specifying arbitrary inter-daemon links. On the top of the *daemon network* is the *logical network* specific to each application. Multiple logical nodes may be created on the same daemon node, thus runing on the same physical node, and they may be interconnected by logical links into an arbitrary topology.

Messenger programs, referred to as Messenger scripts, are written in a subset of C and are compiled into a form of byte code for more efficient transport and parsing [Bid96]. Each script is carried in its entirety by the Messenger as it propagates through the network and is replicated each time the Messenger needs to follow more than one logical link.

2.2 Language Specification

Messenger scripts distinguish three types of variables. *Messenger variables* are private to and carried by each Messenger as it propagates through the logical network. *Node variables* are resident in logical nodes and shared by all Messengers visiting the same

[1] The individual autonomous objects are denoted by mixed case (Messengers), while the system as a whole is denoted by small capitals (MESSENGERS).

logical node. *Network variables* are predefined at each logical node and give each *Messenger* access to the network information local to the current node.

A Messenger script consists of variable declarations followed by a sequence of statements, which can be one of the following types: (1) Computational statements enable the Messenger to perform arbitrary computations. They include all standard C assignment and control statements, involving arbitrary variables and constants; (2) Navigational statements distinguished as *create()*, *delete()*, and *hop()* endow the Messenger with mobility, permitting it to create and destroy logical nodes and/or links, and to move within the logical network; (3) Function invocation statements permit the dynamic loading and invocation of precompiled C functions to be executed in native mode. The *exec()* statement spawns a separate concurrent process for the invoked function, while the *func()* statement invokes the function as part of the current Messenger's behavior and returns its results to it.

For further discussion on navigational calculus in Section 4, we will here concentrate on only the navigational statements: *hop*, *create*, and *delete*.

The *hop* Statement.

The *hop* statement permits a Messenger to move around the logical network. Its syntax is as follows:[2]

$$hop(ln = n; ll = l; ldir = d)$$

where *ln* stands for "logical node", *ll* stands for "logical link", and *ldir* stands for the link's direction. Together, the triple (n, l, d) is a destination specification in the logical network where n can be an address, a variable, a constant (including the special node INIT), or a wild card $(*)$ that matches any name; l can be a variable, a constant, a wild card, or a "virtual link" (corresponding to a direct jump to the designated node); finally, d can be one of the symbols $+$, $-$, or $*$, denoting "forward," backward," or "either," respectively. The default for all three parameters is $*$ and thus may be omitted.

The semantics of the *hop* statement are as follows: From the current node c, replicate the Messenger to all nodes that match n and are connected to c by links matching l and d. The Messenger executing the *hop* in node c then ceases to exist.

Examples (showing the complete syntax and the equivalent default forms):

- $hop(ln = a; ll = x; ldir = *)$
  $\equiv hop(ln = a; ll = x)$
  from the current node c replicate the Messenger to all nodes a connected to c by

[2]The syntax shown here is slightly simplified. In its full generality, a single *hop* statement supports multiple hop specifications, similar to the *create* statement discussed below.

link x (regardless of link direction)

- $hop(ln = *; ll = *; ldir = -)$
  $\equiv hop(ldir = -)$
  from the current node c replicate the Messenger to all nodes connected to c by a backward-oriented link

- $hop(ln = *; ll = *; ldir = *)$
  $\equiv hop()$
  from the current node c replicate the Messenger to all nodes connected to c, i.e., all neighboring nodes

The *create* Statement.

The *create* statement permits a Messenger to create new logical nodes and/or links. Its syntax is as follows:

$$create(ln = n_1, ..., n_k; ll = l_1, ..., l_k; ldir = d_1, ..., d_k;$$
$$dn = N_1, ..., N_k; dl = L_1, ..., L_k; ddir = D_1, ..., D_k;$$
$$[ALL])$$

where each triple (n_i, l_i, d_i) specifies a *new* logical node, n_i, connected to the current node by a (possibly directed) link, l_i. The node n_i is created on the daemon node specified by the triple (N_i, L_i, D_i), which is a destination specification in the daemon network.

The semantics of *create* are then as follows: For each destination specification (N_i, L_i, D_i) determines the set of daemon nodes that match the given daemon name (N_i), daemon link (L_i), and the daemon link direction (D_i). If the optional parameter ALL is omitted, choose one of the possible daemons and create the logical node n_i on it. (The choice is made by a set of rules that are beyond the scope of this report [FBDM98].) The newly created logical node n_i can be named (using a variable or constant) or unnamed ($\sim$), and can be connected to the current node c by the link l_i, which could be named (using a variable or constant) or unnamed ($\sim$). When the optional parameter ALL is included, the new node n_i is created on *all* the daemons matching the destination specification and a copy of the Messenger continues executing in each of the newly created nodes n_i.

The defaults for the logical network parameters n_i, l_i, d_i are "$\sim$" and those for the daemon network parameters N_i, L_i, D_i are "$*$".

Examples. Assume that a MESSENGERS is executing in a logical node c mapped onto a daemon node C.

- $create(ln =\sim; ll =\sim; ldir =\sim;\ dn = *; dl = *; ddir = *; ALL)$
  $\equiv create(ALL)$
  create an unnamed node connected by an unnamed and undirected link to c on all neighboring daemons, i.e., those connected to C by any daemon link.
- $create(ln = a, b; ll = x, y; ldir =\sim, \sim;\ dn = *, *; dl = *, *; ddir = *, *)$
  $\equiv create(ln = a, b; ll = x, y)$
  create a connected to c by x on a neighboring daemon, and create b connected to c by y on another neighboring daemon

The *delete* Statement.

The *delete* statement has the same syntax as *hop* and performs the same navigational operations. However, in addition to moving among nodes, it also deletes all logical links it traverses. If a node becomes a singleton, it is also deleted.

2.3 Node Mapping Algorithm

MESSENGERS supports both explicit and implicit mapping of the logical network structure onto the daemon network and thus on the physical network. Explicit mapping is accomplished by specifying in each *create* statement the daemon node on which the new logical node is to be mapped. Implicit mapping is performed by the system, but the user can control the mapping through appropriate use of the following parameters and constructs:

- The *size* of the daemon network: This determines the number of physical nodes potentially involved in the computation and hence the degree of parallelism.
- The *topology* of the daemon network: This controls the fan-out during logical node creation, i.e., the number of separate physical directions into which the logical network extends.
- The *creation threshold value* for each daemon: This determines the number of logical nodes created by the same Messenger (and its replicas) in any given daemon before it must move onto a neighboring daemon. Hence the threshold controls the clustering of the computation.

  The exact rule is as follows. After N logical nodes have been created on the same daemon node (where N is the threshold value), the interpreter forwards the next new logical node to one of its neighboring daemon nodes. The threshold counter is then reset to zero. The neighboring daemon nodes are selected in round-robin fashion.

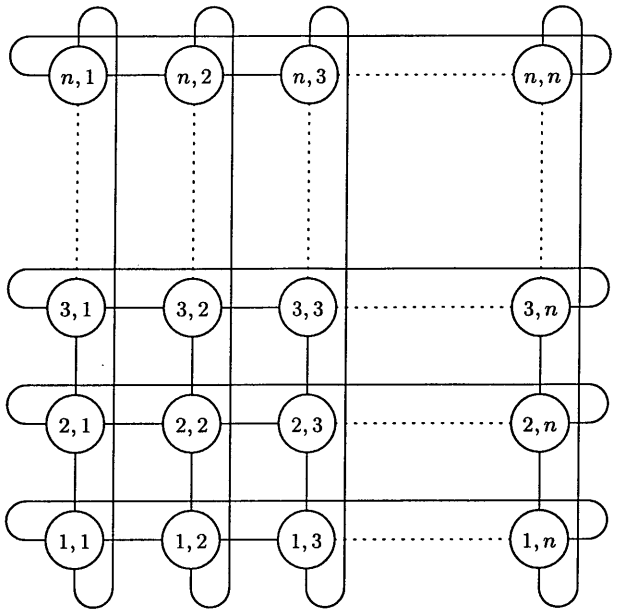


Figure 1: Mapping of a toroidal grid structure

- The *order* in which the nodes are created by the application: This, in conjugation with the threshold and daemon topology and size, determines the actual mapping of logical to daemon nodes.

To illustrate the above concepts graphically, consider the mapping of an $n \times n$ toroidal grid as illustrated in Figure 1. A Messenger can create this grid one row at a time. The first row is created by creating the node $(1,1)$, then $(1,2)$ (along with the link from $(1,1)$ to $(1,2)$), then $(1,3)$, (along with the link from $(1,2)$ to $(1,3)$), and so on. After $(1,n)$ is created, the Messenger creates the link from $(1,n)$ to $(1,1)$. It then creates the node $(2,1)$, along with the link from $(1,1)$ to $(2,1)$. The remaining rows are created in a similar manner. The final step is creating the remaining vertical links. The mapping control parameters interact with this creation scheme as follows. If the threshold value is set to n, all logical nodes in a given row will be mapped onto the same daemon node. If the threshold value is set to 1 and there are n daemon nodes, each logical node in each row will be mapped onto a different daemon node, and all logical nodes in a given column will be on the same daemon node. It is also possible to (1) cluster multiple columns or rows, or (2) cluster any portions of rows or columns.

2.4 Data Structure of Messengers and Logical Nodes

Each Messenger is managed in a *Messenger Control Block* (MCB) as shown in Figure 2. The MCB includes the Messenger's ID local to the current daemon, the file name containing the Messenger's binary code, and several pointers specifying the code area, the messenger variables area, and the current logical node's data structure, called *Node Control Block*. The messenger variables area is allocated with a static size to each Messenger, and thus dynamic memory allocations are not permitted for the messenger variables.

Each logical node is maintained in an NCB. It includes the node address, the node name, and pointers to the node variables area and a chain of logical links emanating from this logical node. When a new node is created, its node address is given by the system as a two-integer value which consists of the IP address of and the positive integer unique inside the corresponding physical node. Similar to the messenger variables area, the node variables area is allocated with a static size to each logical node.

The same NCB is accessible from Messengers residing on the same logical node node, and thus its node variables are used as communication media among Messengers.

3 Using Threads

This section discusses one of our research challenges: instantiating autonomous objects with threads.

3.1 Why Threads

Autonomous objects are generally programmed in network-independent, interpretive scripts. The main drawback is their interpretation cost, which can be however ameliorated by performing most of the computation in native mode. The interpreter process facilitates such native-mode computation by loading precompiled code dynamically or generating native code from scripts through a just-in-time compilation, and thereafter executing it as a part of interpreter itself or having another process execute it.

However, the native code must be coarse enough to hide the overhead incurred from dynamic linking operations, new process generations, and compilations. For instance, we have obtained the following empirical data for MESSENGERS' *func* statement:

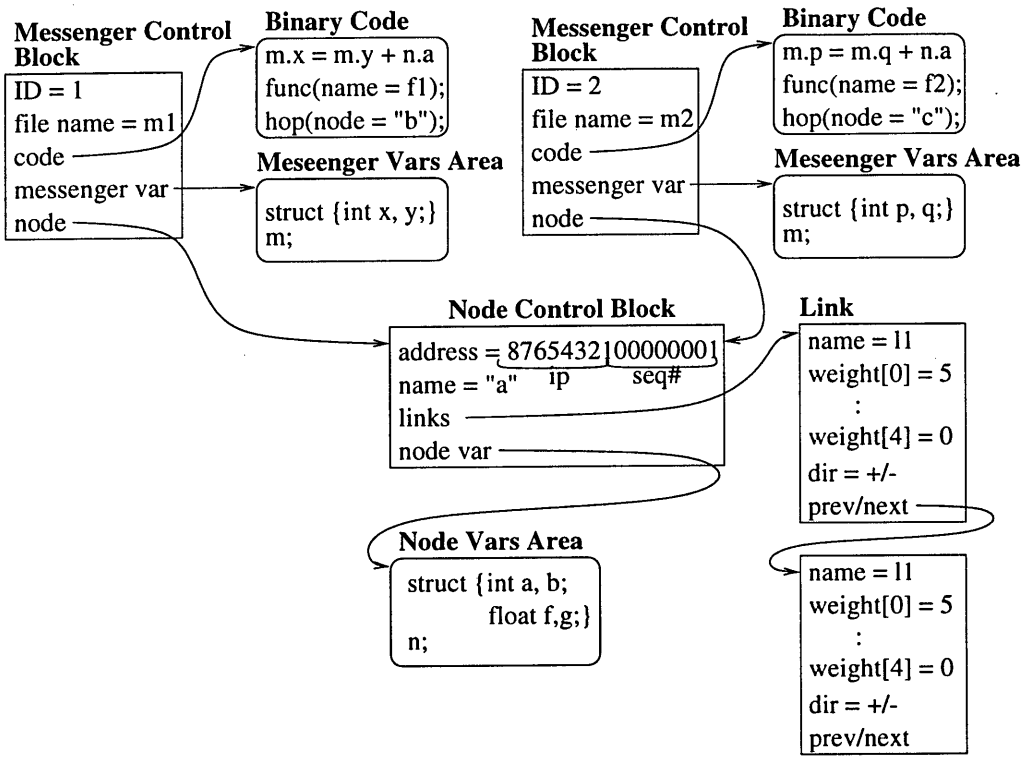


Figure 2: Structure of Messengers and logical nodes

1. The *func* is approximately 40-times slower than an ordinary C function call.
2. It always causes the context switch of Messengers, which costs 50% more than that of threads if the number of execution instances is below 2,000 per workstation.

In addition to the function invocation cost, another performance concern is the concurrency of script interpretation. The script interpretation for each object is interleaved in a time slice or a certain block of the script. For instance, Wave [SB94] switches objects to be interpreted statement by statement. MESSENGERS divides a object into *function block*s [WBDF98], each containing a code between any two of MESSENGERS-unique statements, (i.e. navigational or function-coordinating statements), and initiates a context switch block by block. Since those interpreters work as a process in a single thread, they are blocked when encountering an I/O operation in scripts. Some other systems such as Agent TCL [Gra96] and Tacoma [JvS95] spawn an independent interpreter shell process for each autonomous object and thus emulate the context switch with that of conventional Unix processes upon an I/O operation. However, the process context switch is extremely slow. Java [GM95] provides multithreading, which is however available inside each Java program, (i.e. each applet), but not among different applets.

As a solution to those performance concerns, we instantiate autonomous objects with threads. Each autonomous object is programmed in C language with some special library functions realizing navigational autonomy, and then directly executed by a thread. Therefore, we can avoid the switching overhead between interpretive and native modes, and maximize the concurrency among autonomous objects. In the following, we discuss the details of our strategy to realize autonomous objects with threads.

3.2 Self-Migrating Threads

To begin with, we assume the following environments when using threads.

3.2.1 Assumptions

Since a thread executes a given function code in native mode, the executable code must be provided at any destination nodes to which the thread migrates. The Distributed Shared Memory (DSM) system has no problem, since such an executable code is accessible from any nodes. The homogeneous system not supporting a DSM but a Network File System (NFS) has no problem, either. This is because the same executable code can be loaded at any nodes. However, the homogeneous system

supporting no DSM nor NFS needs a help to load a given function code at a destination node. We assume that the executable image has been transferred by mobile "interpretive" agents such as MESSENGERS prior to thread migrations.

The heterogeneous system falls into a more sever problem, since each workstation has a different instruction set. For this system, we again assume that mobile agents transfer function source code to destination nodes, where they compile it into the executable.

We assume the same layered networks as the current MESSENGERS system uses, namely physical, daemon, and logical networks. Similarly, we provide three distinguishable types of variables: *Messenger*, *node*, and *network variables*. Threads residing on the same logical node can share its *node* variables but not those on different nodes.

3.2.2 Design Strategy

The main hurdle to implement thread migrations is how to capture and migrate the state of each thread. The state includes a CPU state, a stack, I/O descriptors, and dynamic memory spaces associated with the migrating thread. We must also prevent race conditions among multiple threads. The following shows our strategies to realize thread migrations:

- **Migration at top level of execution:**
  Upon an invocation, a thread is given four pointers specifying *messenger*, *node*, *network* and *argument* variables areas. The thread is prohibited to use any other local variables in its top level function, while allowed to use any variables in the lower level functions. We also permit thread migrations on "only" the top level of execution [WBDF98]. In addition, I/O descriptors are restricted to be declared in node variables only, and thus need not be transferred to remote nodes. With these four restrictions, the thread state to be captured is the initial four pointers given to the thread and the data pointed by them.

- **Dynamic memory salvation:**
  The contents of pointers are private to each workstation and thus no longer valid after a thread migration. However, pointers are necessary to allocate dynamic memory spaces. We provide library functions to keep track of dynamic memory spaces and pointer contents. Each logical node has a pointer table which registers the size, base address and pointers in use of the node variables area. Similarly, each autonomous object has a pointer table for its *messenger* variables area. Also, upon a dynamic creation of a new memory space, a pointer table is attached to this space. Such tables are handled by our library functions. When

leaving the current node, an autonomous object carries all memory spaces and the corresponding tables attached to them. Upon an arrival to the destination, it looks through each pointer table, allocates the required space, and recovers each pointer registered in the table by computing the displacement from the base address to this pointer. The final implementation may have the language processor analyze all pointer operations and insert the library functions in the code automatically.

- **Non-interruptibility:**
  Threads share I/O descriptors and *node* variables. Since I/O operations can be serialized by the *flockfile()* POSIX/SOLARIS thread library function, *node* variables are only ones to which accesses must be serialized. This is performed by activating only one thread per logical node. In other words, we guarantee that multiple active threads are not residing on the same logical node, and thus never compete the same *node* variables. They may cause a context switch upon an I/O or memory heaping operation. However, I/O operations can be serialized as explained above and the memory allocation is guaranteed to be thread safe on the POSIX/SOLARIS environment. Therefore, we can permit threads on different logical node to run concurrently, while serializing the thread execution per each node. A context switch among threads on the same logical node occurs only when each one migrates somewhere. For inter-threads communication via node variables, threads must voluntarily relinquish a CPU by migrating to the same node.

3.3 Implementation Details

In the following, we give the implementation details of our thread management strategies.

3.3.1 System Components

The system consists of three components shown below:

1. **Daemon:** Each processor runs a daemon process which maintains a logical network and schedules threads running on it. It is also in charge of handling thread migrations with other daemons.
2. **Preprocessor:** Each autonomous object is described in C. A language preprocessor inserts into this source code additional library functions that maintain pointer contents and migrate the thread state. Thereafter, a conventional C compiler generates the executable module from the modified source code.

3. **Library:** Two libraries are provided: one for pointer maintenance and the other for thread migration. The former keeps track of pointer contents and maintains a pointer table for each memory space allocated to the calling thread. The latter captures and migrates the calling thread state: its program counter, messenger variables, dynamic memory spaces and their pointer tables.

3.3.2 Pointers

A pointer table is assigned to a new memory space upon its dynamic allocation. The table records the memory size, the base address, and pointers in use. All table operations are handled by the following library functions:

new_my_tbl(size, addr)	attach a new table to the calling thread upon a malloc()
free_my_tbl(my_tbl_id)	free the table from the calling thread upon a free()
new_node_tbl(size, addr)	attach a new table to the current node upon a malloc()
free_node_tbl(node_tbl_id)	free the table from the current node upon a free()
to_node_tbl(my_tbl_id)	attach a copy of the table from the thread to the node
from_node_tbl(node_tbl_id)	attach a copy of this table from the node to the thread
reg_ptr(ptr, tbl_id)	register the specified pointer to the table
free_ptr(ptr, tbl_id)	erase this pointer from this table

Figure 3 shows the code of a thread which allocates a dynamic memory space, attaches it to the current logical node, and departs for somewhere. Several library functions (shown on the right side) are appended to the original source code (listed on the left side).

```
(1) thread_A () {              // library function to support pointers
(2)   msgr.p = malloc(size);   msgr.my_tbl_id = new_my_tbl(size, msgr.p);
(3)   *msgr.p = data;          reg_ptr(msgr.p, 1);
(4)   node.p = msgr.p          to_node_tbl(msgr.my_tbl_id); reg_ptr(node.p, 2);
(5)   if (msgr.x > 0)
(6)     msgr.p = NULL;         free_my_tbl(msgr.my_tbl_id);
(7)   thr_move();
(8) }
```

Figure 3: Exchanging a dynamic space between two threads

Figure 4 describes how this thread code handles pointer tables associated with it. The thread A in the figure has a pointer table for its *messenger* variables area, which has a preassigned table ID #1. Similarly, the current node has a pointer table for its *node* variable area, accessible with the table ID #2. When the thread A allocates a new memory space *dyn*0 (line 2), it attaches a new pointer table for *dyn*0 and obtains its table ID. Since the thread uses the *messenger* variable, *msgr.p* as a pointer (lines 2 and 3), it registers this variable in the pointer table #1, (i.e., the one for the

messenger variables area.) Thereafter, the thread records the address of *dyn*0 into the *node* variable, *node.p* (line 4). Therefore, a copy of the table for *dyn*0 is made and attached to the node. In addition, *node.p* is registered in the table #2, (i.e., the one for the node variables area.) Depending on the content of *msgr.x* (line 5), the thread *A* nullifies the *msgr.p* (line 6), which requires the thread to free *dyn*0's pointer table. Upon a thread migration (line 7), the thread *A* duplicates and carries *dyn*0's contents if it has not freed the pointer table by bypassing line 6.

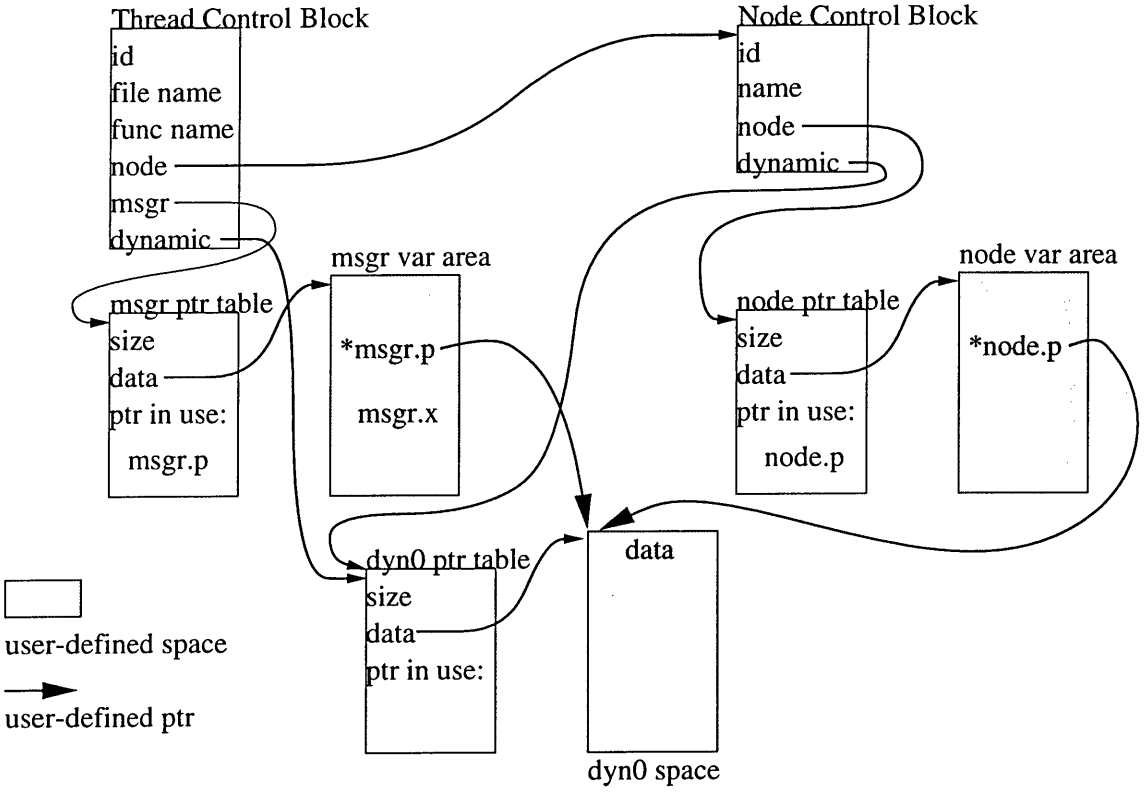


Figure 4: Pointer maintenance tables

3.3.3 Migration

Thread migrations over logical network is performed through library calls. The definition of the library functions are given in Section 4. The network navigation is distinguished into two types: the intra- and inter-daemon navigations. The former migrates a thread between two logical nodes, both located inside the same daemon. The arguments given to a thread are always four pointers to *messenger*, *node*, *network*, and *argument* areas. Therefore, the intra-daemon navigation only switches the second pointer, (i.e., the node poitner), from the source to destination *node* variables areas. No state capturing occurs.

The latter, (i.e., the inter-daemon navigation), migrates a thread between two logical nodes, both located on different daemons. The state to be captured includes:

1. its Messenger variables area
2. all dynamic memory spaces
3. all pointer maintenance tables
4. its executable file name and function name
5. the displacement from the function entry point to the current program counter, (which we refer to as a "function displacement")

The daemon's work, when receiving a thread, includes:

1. updating the contents of all the pointers in Messenger variables and dynamic memory areas according to the pointer maintenance tables,
2. copying into a new function stack four pointers to Messenger, *node*, *network* and *argument* areas,
3. locating the appropriate function entry point from the executable file name and function name,
4. computing the thread-resuming point by adding the "function displacement" to the entry point, and
5. jumping to that point

3.3.4 Thread Scheduling

Each logical node has its own lock variable, for which all threads residing on it compete with each other. In other words, the logical node is regarded as a monitor. Only a thread to acquire the lock is allowed to run and access the *node* variables. Whenever it navigates itself somewhere, it releases the lock to resume another thread. Each daemon gets prepared for a priority queue maintaining runnable threads, which reside on different logical nodes.

For inter-daemons communication, we have obtained a good performance by multithreading, where a daemon instantiates for each remote daemon a thread that maintains a TCP/IP socket permanently and exchanges data whenever the socket is ready to be read or written [WBDF98]. To avoid a confusion, we refer as "socket threads" to those in charge of TCP/IP communication, while referring as "user threads" to those instantiated to execute autonomous object code.

the main thread	spawns all other threads and controls the daemon
user threads	execute a user's autonomous object code
socket threads	communicate with each remote node

Table 1: Types of threads running inside the daemon

The daemon itself works with the main thread that not only spawns these user and socket threads but also maintains a logical network and takes care of thread migrations. In summary, we have three types of threads to be scheduled, as shown in Table 1.

3.4 Related Works

Thread migrations have been studied for the purpose of the alleviation of remote memory accesses by dispatching such threads that frequently refer to remote memory. In the following, we differentiate our self-migrating threads from other major multithreading systems, in terms of navigational autonomy, state capturing, parallelism, and dynamic data transfers.

- **Navigational Autonomy:** In most existing systems, the migrations are rather passive and triggered when threads refer to remote data. To avoid thread migrations from thrushing between nodes, the compiler need be smart enough to decide upon which remote access a thread must be migrated. Such decisions usually depend on each application, and may need some compiler directives given from users.
- **State Capturing:** It is quite expensive to capture the entire stack of a migrated thread. Therefore, Olden from Princeton [RCRH95] transfers only the current layer of stack, while Nomadic Thread from USC [JG96] manages a thread in a simple activation flame and always dispatches this flame to a remote node. Upon a termination, Olden has a remote thread merge its termination value into the original stack. Nomadic Thread has a child thread consult with a special "result" thread in order to locate the parent thread possibly migrated somewhere and return the termination value to it.
- **Dynamic Data Transfer:** The shared-memory-based thread migrations such as Olden has no problem in accessing dynamic data structures such as lists and trees, since pointers are visible at any processors. In Nomadic Thread, pointers to dynamic data consists of two fields: the ID of a CPU containing this data and a memory address where it is located. Thus, when a thread accesses data through a pointer, it is migrated to the CPU specified by the pointer and refers to the data at the given address.

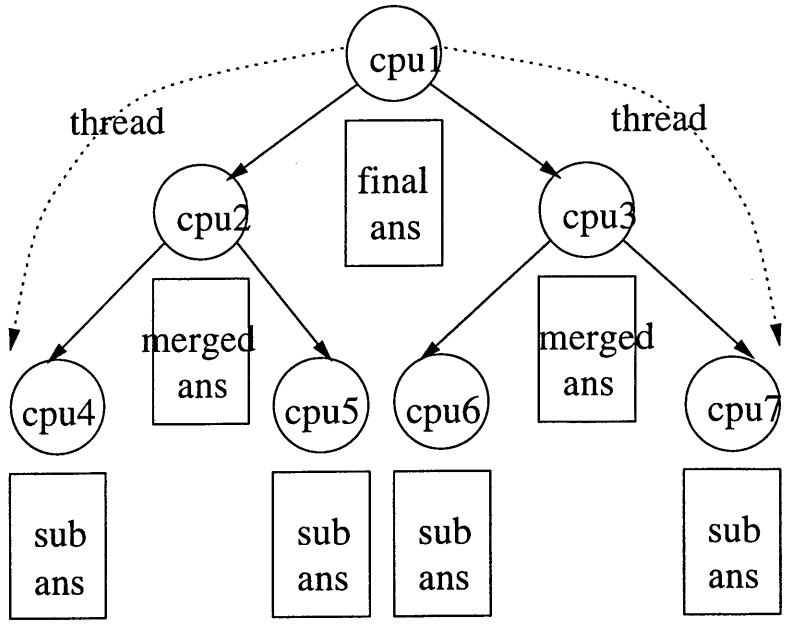


Figure 5: Thread migrations in a divid-and-conquer algorithm

However, pointers do not always address efficient accesses to dynamic data structure. Consider a merge sort, based on the divide-and-conquer algorithm where a sequence of data is recursively divided into two sub sequences until it has only two data items, (i.e., a divide phase), and those sub sequences are merged back to a single sorted data, (i.e. a conquer phase), as shown in Figure 5. Assume that two threads are migrated to two nodes, containing lift and right child subsequences respectively in a divide phase. They return only the pointers to their sorted subsequences to the parent in a conquer phase. The problem is that those two sorted subsequences still remains on remote nodes and thus remote accesses occur upon when those two results are unified. This problem is resolved in our self-migrating threads with the idea of pointer table.

The distributed-memory-based thread migrations have been studied in UPVM [CKO+94], Ariadne [MR96], and Emerald [RTL+91]. David Cronk et al. reviewed pointer operations in these systems and proposed the use of pointer table [CHM97]. Our pointer table management is different in permitting threads to move dynamic memory spaces from their private space, (i.e. the *messenger* variables area), to the shared space, (i.e. the *node* variable area), or vice versa. Therefore, it is even possible for different threads to exchange their dynamic spaces.

- **Parallelism:** In addition to the occurence of I/O blockings or timer interrupts, a thread may be switched to another one when it needs data provided from

remote threads. For instance, Nomadic Thread provides split-phase operations to access I-structures and causes a context switch whenever a thread is blocked to read an I-structure from remote threads. Olden switches a thread to another one till it can receive a return value from its children through *future* and *touch* operations. Our self-migrating threads initiate a context switch upon every migration including a dummy hop to the current node.

In summary, our self-migrating threads navigate over network with user's full intention, and simplify the state capturing with a few restrictions to stacks, while supporting pointers necessary to use dynamic memory space. They maintain fine grain parallelism with their context switches upon I/O operations and network navigations.

4 Navigational Calculus for Graph Constructions

This section discusses a new navigational calculus to facilitate a large-scaled graph construction on parallel and distributed machines.

4.1 Problems in Current Logical Network Constructions

Most mobile agents and thread migrations are generally based on the explicit addressing, in which they enumerate destinations with IP or memory addresses [BDF98]. In contrast to this addressing is the implicit addressing, in which Wave and MESSENGERS propagate their objects to multiple nodes that meet the selection criteria they indirectly enumerates. Their objects navigate over physical network as generating application-dependent graph or logical network on it. This addressing gives a great flexibility to construct and navigate over various graphs. For instance, $l\#a$ in Wave or $hop(node = a; link = l)$ in MESSENGERS propagates the calling object to all vertices named "a" along all edges named "l" from the current vertex, thus without enumerating all such the destination vertices explicitly.

However, for parallel graph generation, the implicit address still has several functional problems yet perfectly addressed by none of the systems we discussed so far. Furthermore, of importance in parallel computing is the efficiency, which contradicts the flexibility. In the following, we show such functional and performance problems to be solved for the network navigations.

1. **Edge Connections**
   The implicit addressing cannot always locate a single logical node, since logical nodes and links may have the same name. The easiest solution to this problem

is allowing the explicit addressing as an option. Generally, the explicit addressing locates each logical node with the system-unique machine name and the intra-machine sequential number. For instance, Wave and MESSENGERS use IP address as their system-unique machine names. In such naming, the system but not the user has an authority to give each logical node a system-unique address upon its creation. Therefore, when creating a new link to an existing node, an autonomous object must previously visit this node to obtain the address from the system.

Figure 6 gives such examples. Figure 6 (A) shows that a Messenger must first create a new node, "d" whose address is 0x8765432100000001 prior to establishing a link connection from a node "c" to the node "d". This in turn means that two *Messengers*, residing on nodes "b" and "c" respectively, cannot create a new link to the same new node, "d" in parallel. Figure 6 (B) shows a problematic construction of a mesh. One of mesh construction algorithms is generating horizontal links first and then vertical ones. During the horizontal links construction, the Messenger must memorize all the addresses of grid points, so that it can connect vertical links to appropriate grids. The larger mesh to be created the more addresses to be memorized. Therefore, using node addresses is not appropriate to construct large graphs.

Hence, we need an efficient addressing that permits autonomous objects to give a unique address to a new node by themselves.

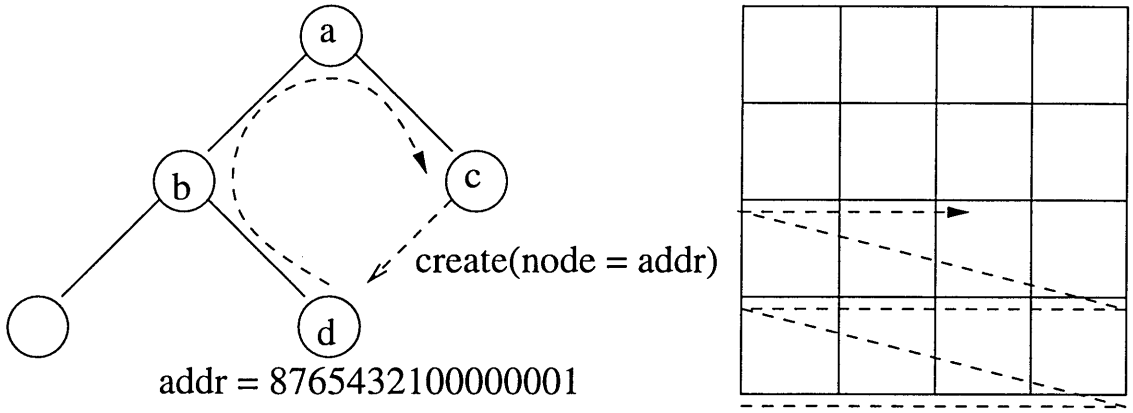


Figure 6: Edge connection

2. **Simultaneous Executions of Different Navigational Statements**
   Wave provides two distinct navigational commands, *create* and *hop*, each represented as # and $CR(\#)$, which in turn means that a network can grow but not shrink. MESSENGERS gives one more navigational command, *delete* to permit a network to change freely. Currently, those distinct commands must be exe-

cuted separately, while each of them can be applied for multiple link creations, propagations, or deletions. This may be still a restriction to parallel network modifications. Figure 7 gives a problematic example. Given a line segment AB, assume that an autonomous object resides at the end A and wants to delete the segment AB and to create a new one AC. Unless the object were permitted to perform the link deletion and creation in parallel, it would have to first delete AB, hop back from B to A, and then create AC in sequential. In summary, the navigational calculus must support simultaneous executions of different network operations such as create, hop, and delete.

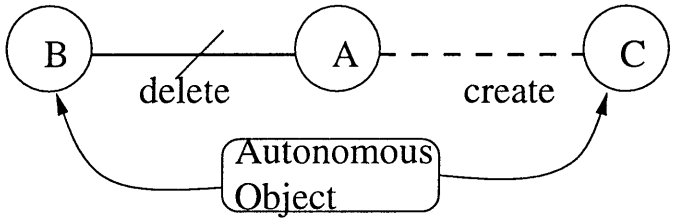


Figure 7: Simulataneous graph modificaitons

3. **Atomic Modifications**
   Graph applications may need to change not only the topology of their graphs but also the attributes of their graph vertices and edges. Since those attributes are shared among autonomous objects, references and modifications to them are critical sections. For instance, given a link with the weight W, assume that two objects residing on the different ends of the link need to increment this weight W respectively. The expected value should be $W + 2$, while it may be $W + 1$ if the system does not guarantee the atomicity for a sequence of operations: read, modify and write. Thus, operations to logical nodes must be atomic. Since links may be created over different workstations, such atomic operations require distributed synchronization among them.

4. **High-Speed Navigations**
   Navigations along links use only local information, (i.e. the name and weight of links emanating from the current node). The advantage is that autonomous objects can be programmed to navigate over a logical network without the knowledge of its global topology. On the other hand, the disadvantage is that string and integer comparisons must be achieved for each of all links incident to the current node in order to find the links matching selection criteria. This incurs a large amount of overhead for network navigation.

   Autonomous objects may want to locate a link with its identification local to the current node other than its link name and weight, which may not be unique even among links incident to the same node. Figure 8 shows such an example,

where an autonomous object quickly locates one of four links emanating from a given node with its local link ID #4 other than its link name, "xyz".

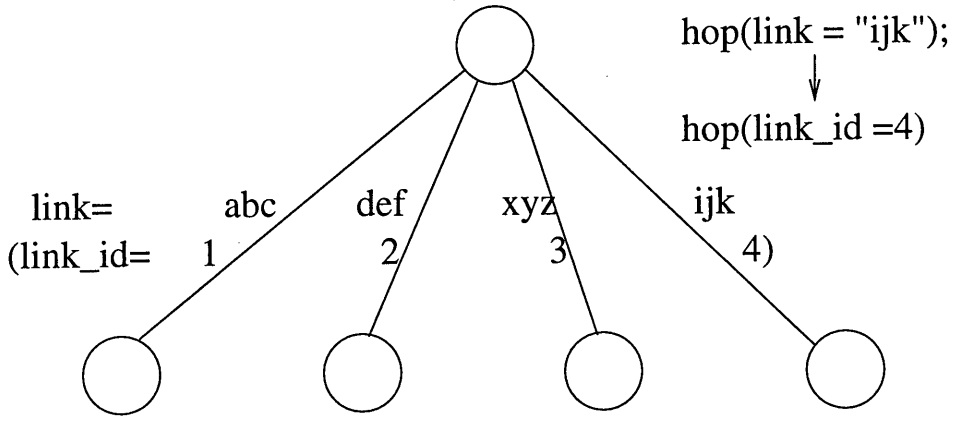


Figure 8: Fast link navigations

4.2 Language Specification

In this subsection, we propose a new navigational calculus which addresses all problems mentioned above, and thereafter demonstrate its efficiency using several examples.

4.2.1 Definitions

The new definition is based on our initial definition of navigational calculus [FBDM98]. Given a logical network N, our calculus finds the subset of all logical nodes $n \in N$ that match the destination spec or $DEST = \{n|match(n, dest\_specs)\}$, where $dest\_specs = (daemon, addr, node, src\_id, dst\_id, link, weight)$. Table 2 shows the meanings of destination spec keywords.

The destination nodes are the ones residing on a given *daemon*, addressed with *addr*, having a name specified in *node*, and accessible along links that have *src_id* and *dst_id* as their link IDs at the source and destination node as well as a string *link* name and an integer *weight*. Table 3 lists arguments available for each navigational keyword.

Table 4 summarizes the meaning of arguments in each keyword.

In addition to the destination specs *DEST()*, we also define a set of navigational

keywords	meanings
daemon	decides destination daemons
addr	decides a destination node with the system-unique address
node	decides destination nodes with the name in its argument
src_id	decides a link with the ID unique to the current node
dst_id	decides a link with the ID unique to the destination node
link	decides links with the name given in its argument
weight	decides links with the weight given in its argument

Table 2: Keywords used in our navigational language

keywords	arguments
daemon	$\sim$ | @symbol | string
addr	$\sim$ | integer
node	$\sim$ | string
src_id	$\sim$ | integer
dst_id	$\sim$ | integer
link	$\sim$ | string
weight	$\sim$ | integer | @symbol

Table 3: Keywords and their arguments to decide the destination criteria

actions or *ACT()*. The *ACT* accepts the arguments summarized in Table 5.

ACT() receives the pointer to a function, which computes and returns an new weight of the link being traversed. The function is given four arguments: the value of the current weight and pointers to the messengers variables area, the node variables area of the source node and the arguments area.

Using the above calculus and actions, namely *DEST* and *ACT*, we define a new navigational statement as shown in Figure 9.

The first *p_int*, which does not follow a keyword, specifies the number of navigational operations issued in parallel. For instance, *thr_move(3; action = @create, @hop, @jump; node = "a", "b", 15)* involves three individual navigations, which are a network creation, a propagation, and a direct jump. Each individual navigation receives arguments appearing at the same position from each keyword. In the above example, the actions *@create*, *@hop*, and *@jump* receive "a", "b", and 15 respectively as their argument in the *node* keyword. The calculus has the abbreviation rule shown below:

- The N repetitions of the same argument, arg, can be simplified in $arg * N$.

daemon

arguments	meaning
~	don't care, (a daemon chosen by the system)
@n	one of neighboring daemons
@.	the current daemon
@a	each of all daemons
@e	each of all neighboring daemons
string	the daemon with this string as its name

addr

arguments	meaning
~	don't care, (a node name/address not given nor tested)
0	the init node
integer	the node with this integer as its address

node

arguments	meaning
~	don't care, (a node name/address not given nor tested)
string	the nodes with this string as its name

src_id/dst_id

arguments	meaning
~	don't care, (a src_id/dst_id not given nor tested)
integer	the link with this integer as its source or destination ID

link

arguments	meaning
~	don't care, (a link name not given nor tested)
string	the links with this string as its name

weight

arguments	meaning
~	don't care, (a weight not given or tested)
@>	links whose weight is the maximum at the current node
@<	links whose weight is the minimum at the current node
@+	links with a positive weight
@−	links with a negative weight
integer	links with the this integer value as its weight

Table 4: Arguments and their meanings

arguments	meaning
@create	create a new logical link and a new node at the destination
@hop	propagate to the destination along links
@jump	jump directly to the destination
(*func)()	do the same as hop and modify the traversed links with func()

Table 5: Navigational actions

```
link_ID = thr_move( p_int
             ; action = action_argument{[,.., action_argument]}[*[p_int]]
            [; daemon = daemon_argument{[,.., daemon_argument]}[*[p_int]]
            [; addr   = addr_argument{[,.., addr_argument]}[*p_int]]
            [; node   = node_argument{[,.., node_argument]}[*[p_int]]
            [; src_id = id_argument{[,.., id_argument]}[*p_int]]
            [; dst_id = id_argument{[,.., id_argument]}[*p_int]]
            [; link   = link_argument{[,.., link_argument]}[p_int]]
            [; weight = weight_argument{[,.., weight_argument]}[p_int]]
          );

p_int := positive integer constant | positive integer variable
```

Figure 9: Navigational Statement

- If the notation, $arg * N$, is followed by no more arguments, it can be further simplified in $arg*$.
- The symbol $\sim$ can be simply omitted.
- The notation, $\sim *N$ is thus equivalent to $*N$.
- If a keyword has the notation, $\sim *$ or $*$ as its argument, it can be simply omitted.

With this abbreviation rule, we can transform the following three keywords and their arguments into some simplified forms.

$(keyword =\sim, \sim, \sim, a, b, c) \equiv (keyword =,,, a, b, c) \equiv (keyword = \sim *3, a, b, c) \equiv (keyword = *3, a, b, c)$

$(keyword = a, b, c, \sim, \sim, \sim) \equiv (keyword = a, b, c,,,) \equiv (keyword = a, b, c, \sim *) \equiv (keyword = a, b, c, *) \equiv (keyword = a, b, c)$

$(keyword =\sim, \sim, \sim, \sim, \sim, \sim) \equiv (keyword =,,,,,) \equiv (keyword = \sim *) \equiv (keyword = *) \equiv ()$

4.2.2 Creation

The *thr_move* navigational statement constructs a logical network with the *@create* constant. New logical nodes to be created all satisfy a destination spec partially determined by *DEST(daemon, addr, node)*. Two or more different nodes can share the same node, while they must have their system-unique address. To enforce this rule, no more node is duplicated when the *addr* keyword receives an *integer* value that has been already used as a node address over the system. The *thr_move* statement propagates the calling thread to each destination node, as establishing a new link on its way from the source to the destination. The attributes of the link are determined by *src_id*, *dst_id*, *link*, and *weight*.

Among them, *src_id* and *dst_id* are link IDs dedicated to the source and destination nodes respectively. For instance, assume that a link has been created from nodes A to B with the source ID #1 and destination ID #2. At the node B, the link is identified with the source ID #2 and destination ID #1. The link ID must be unique to each logical node, which in turn means that no link creation using an existing ID is successful and terminates the calling thread. Link IDs must be also positive when they are given by a user. This is because the system automatically gives negative integer values as a new link's source and destination IDs when they are not explicitly given by a user.

4.2.3 Hop

The *thr_move* propagates the calling threads to destination nodes with the *@hop* constant. All the destinations must satisfy a complete spec determined by *DEST(daemon, node, src_id, dst_id, link, weight)*. The propagation must be done along links that emanate from the source node and satisfy *src_id*, *dst_id*, *link*, and *weight* keywords. The *thr_move* terminates the calling thread if *DEST* has no destination at all.

4.2.4 Jump

The *thr_move* makes the calling threads jump directly to destination nodes with *@jump* constant. All the destination must satisfy a spec only determined by *DEST(daemon, addr)*. This direct jump is performed with regardless of the existence of links. The *thr_move* terminates the calling thread if there exists no node with a given address.

4.2.5 Change

If the *action* keyword receives a pointer to a function, *thr_move* propagates the calling threads in the same manner as the hop operation. However, unlike the hop, it also modifies the weight of links traversed by the threads. The new weight is returned by a user function which receives four arguments: the weight of the traversed link and pointers to the messenger variables, the source node variables, and the arguments areas.

There is a special case where the *action* keyword receives a null pointer. This deletes the links traversed by the thread. When all links from the source node are deleted and it becomes a singleton, the source node is also deleted.

4.2.6 Example

The following *thr_move* function issues four independent navigations. The first navigation is a direct jump to the node identified with the system-unique address #1. The second one establishes a new link with the source id #5 and the name, "link_x", navigates the calling thread along this link, and creates a new node "node_a" at the other end of the link. The third operation finds the link whose weight is the minimum at the source node, and navigates the thread along the link, as incrementing its weight. The fourth one finds the link with the maximal weight and deletes this link after navigating the thread along it. Figure 10 describes this network navigations.

```
int func(int weight; *void p1, p2, p3)
{       return(weight + 1);         }

thr_move(4;
         action = @jump, @create, func,    0;
         daemon = ~,     ~,       ~,       ~;
         addr   = 1,     ~,       ~,       ~;
         node   = ~,     node_a,  ~,       ~;
         src_id = ~,     5,       ~,       ~;
         dst_id = ~,     ~,       ~,       ~;
         link   = ~,     link_x,  ~,       ~;
         weight = ~,     ~,       @max,    @min);
```

According to the abbreviation rule, the above *thr_move()* is simplified into the following form:

```
thr_move(4;
```

```
action = @jump, @create, func,     0;
addr   = 1;
node   = ,      node_a;
src_id = ,      5;
link   = ,      link_x;
weight = *2,              @max,     @min);
```

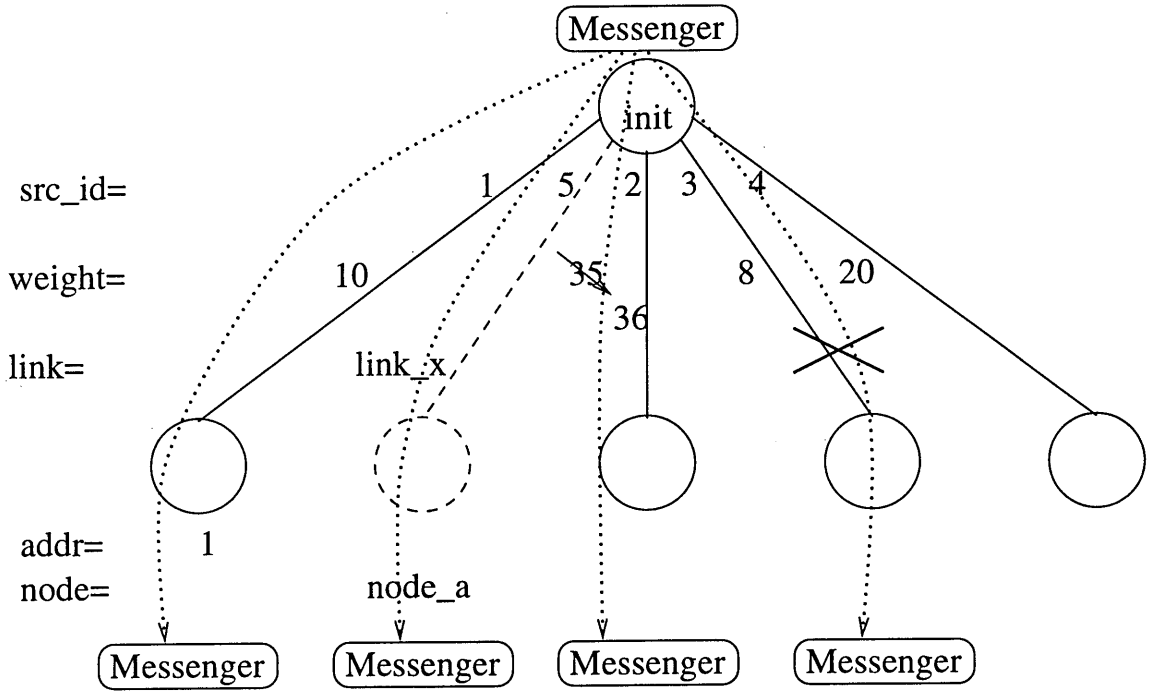


Figure 10: Network navigations by a new calculus

4.2.7 Network Environment Information

The network environment information can be obtained via the following predefined variables. Those variables are set to point the appropriate network structure by the run-time system.

$address	the system-unique address of the current node
$node	the current node name
$nlinks	the number of links emanating from the current node
$link[i]	the name of the local link with source ID i
$weight[i]	the weight of the local link with source ID i
$neighbor[i]	the neighboring node address accessed from the link with source ID i

4.3 Navigations Using New Calculus

Now, we demonstrate how the new calculus addresses the functional and performance problems discussed in Section 4.1.

Edge connection:
With the new calculus a user can assign a system-unique address to each new node. If the system has no such node specified with a user-defined address, it creates one with this address. Otherwise, the system simply creates a new link to the existing node accessible with the given address. Using this feature, four nodes "a", "b", "c", and "d" in Figure 6 can be constructed with the following code:

```
thr_move(1, action = @create; addr = 1; node = ''a'');
thr_move(2; action = @create*, addr = 2, 3; node = ''b'', ''c'');
thr_move(1, action = @create; addr = 4; node = ''d'');
```

Different navigations:
Since the new calculus allows different types of navigations in the same *thr_move* function, the graph modification in Figure 7 can be performed by a single statement as shown below:

```
thr_move(2; action = 0, @create; node = ''b'', ''c'');
```

High speed navigations
The new calculus permits the the thread to select a link with an ID local to the current node. From the implementation point of view, an appropriate link data structure can be quickly selected without performing string comparisons for all links emanating from the source and thus the navigation will be accelerated drastically. The following statement is a faster navigation along the link "ijk" in Figure 8:

```
thr_move(action = @hop, src_id = 4);
```

4.4 Related Works

We focus on two major works related to graph description languages: (1) the one constructing a computational network with a net list and mapping it over a specific parallel machine and (2) the other defining the dynamic growth of graphs in L-systems and presenting it in computer graphics.

- **Net-List:** Given a set of nodes, the net list enumerates node-to-node static connections for all these nodes. Such node-to-node connections may be represented

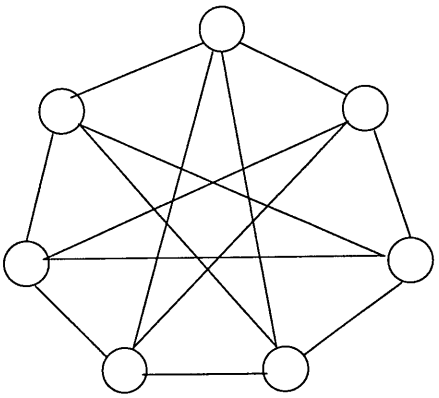
with some regular expressions. For instance, if there is a regularity in that node $\#i$ is connected to node $\#i + 1$, the connections may be simply described:

```
edge(i): node(i) => node(i+1)
```

other than enumerated:

```
edge(1): node(1) => node(2);
edge(2): node(2) => node(3);
....;
```

Figure 11 shows how a 7-body static graph is described in a net list.



```
type ring_edge(i)      node(i) => node((i+1) mod n);
type chordal_edge(i) node(i) => node((i+(n+1)/2) mod n);
```

Figure 11: Describing a 7-body static graph in a net list

The mapping problem arises when the topology of a graph is different from that of the underlying physical network. However, various static mapping algorithms have been well studied for net-list graphs. P-prep [Ber87] and Oregami [LRG+95] are such systems that efficiently map a given graph to a target architecture, as applying existing algorithms to the graph and following tuning-up instructions from users.

The main disadvantage is that those systems focus on only the initial static mapping from logical to physical networks, and thus they are not capable of dynamic/incremental graph generation nor dynamic graph remapping. Another disadvantage is the poor programmability and understandability of the net list. The larger graph we have, the more difficulty we have in tracing.

- **Lindenmayer-Systems:** L-systems [RS74] are based on strings rewriting. Each alphabet of a base string ω is recursively rewritten through a given production rule p. They introduce a virtual turtle and make it draw a graph, as reading such growing strings and interpreting each alphabet of the strings as a edge/node generating instruction. For instance, assuming that F, $+$, and $-$ mean an edge creation, right and left turns, respectively, the following L-system description

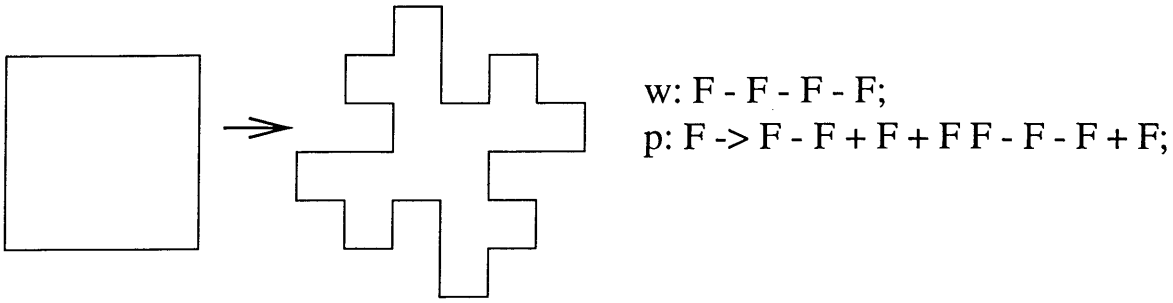


Figure 12: A growing graph descibed with an L-system

starts with a square denoted by ω and recursively rewrites each edge into a more complicated graph through a rule p, as shown in Figure 12

$$\omega : F - F - F - F$$

$$p : F \rightarrow F - F + F + FF - F - F + F$$

The main advantage of L-systems is their ability of generating a string sequence for a complicated graph through their concise production rules. A graph can grow with continuous repetitions of string rewriting, and can be therefore used for simulating developmental processes of natural organisms [PL96]. Furthermore, each alphabet of a string, (i.e. F in the example above) may be rewritten in parallel, and thus parallel processing is possible by assigning a portion of a string to each processor.

On the other hand, L-systems have several disadvantages: (1) a graph is drawn by the sequential turtle interpretation of a string, (2) a generated graph is acyclic in general, due to the lack of rule to connect branches, and (3) a graph is maintained in a string, and thus it is not easy to solve various graph problems such as a shortest path and a minimum spanning tree.

Our new calculus addresses these disadvantages of both net-list descriptions and L-systems, except dynamic graph remapping, which we will support with a load balancing scheme discussed in the next section.

5 Load Balancing

In this section, we first points out problems in IP-oriented node addressing, implicit node mapping, and user-defined node addressing when we introduce dynamic load

balancing. Then, we propose a new node mapping and remapping algorithm to address these problems.

5.1 Research Challenges

5.1.1 IP-oriented Node Addressing

Logical nodes must have their own system-unique address, while they can share the same name. We showed that such address should be given by users from the programming point of view. This is also true when we introduce a node remapping algorithm. In other words, nodes should be maintained with logical address. We explain this reason, using the current node addressing scheme in MESSENGERS. The MESSENGERS system identify each logical node with the IP address of and the sequential number local to the workstation it belongs to. Figure 13 describes how a series of six nodes are generated in MESSENGERS. For simplicity, let these processors' IPs be 1, 2, and 3. Also assume that each processor has a threshold value to create nodes consecutively as shown in the figure. Consider the third node that was created on the processor 1 and given 100000003h as its address, (abbreviated as 1.3). Then, let us assume that the processor 1 migrates the node 1.3 to the processor 3. Since all the nodes neighboring to 1.3 are informed of the node 1.3's migration, Messengers navigating along links from those neighbors to 1.3 could correctly arrive at the processor 2. However, non-neighboring nodes are out of knowledge about the node migration and thus must use IP information. Messengers jumping directly to 1.3 would still arrive at the processor 1 and should be forwarded to processor 3. Such forwarding is resulted from IP-oriented addressing. Therefore, the node address must be independent of IP or physical information.

5.1.2 Implicit Node Mapping

In Section 2, we demonstrated that the MESSENGERS implicit node mapping algorithm is conveniently working for generating simple networks. However, we still need to cope with more complicated network generation and mapping. When constructing an $n \times n$ logical grid on n processors, MESSENGERS can map to the same processor all the nodes on the same row (or column). However, it is much more complicated to map a $\sqrt{n} \times \sqrt{n}$ sub-mesh on each processor as shown in Figure 14(A). Another example is a tree mapping as shown in Figure 14(B), which recursively allocates each sub-tree to a different processor every certain depth of the tree.

The explicit node mapping allows each application program to specify a processor to generate the next new node, however it makes the thread code inflexible for system

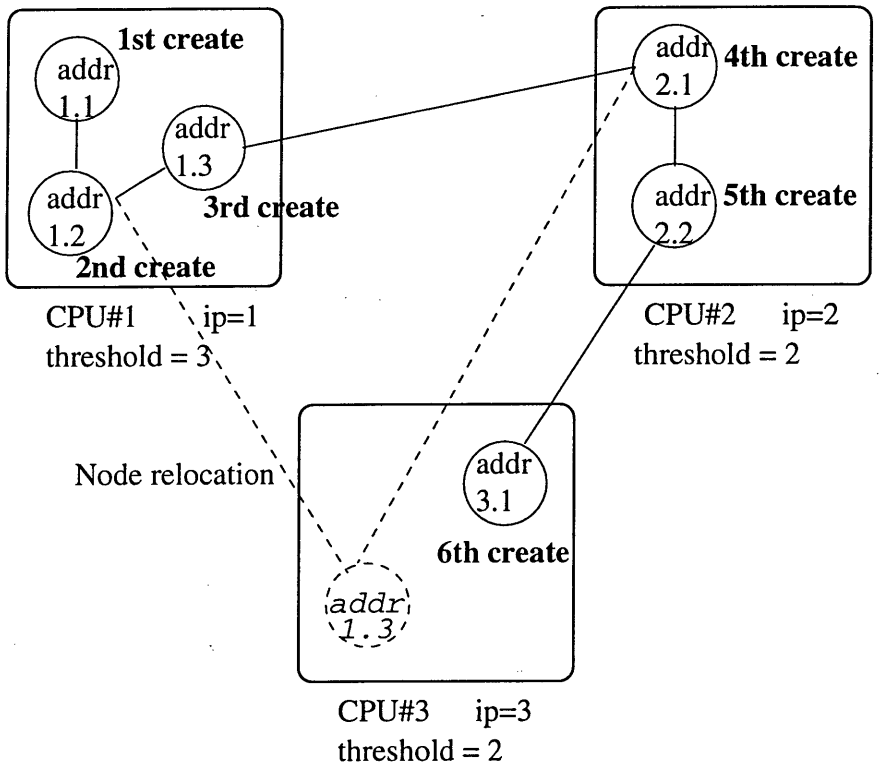


Figure 13: Node generations in IP-oriented addressing

reconfiguration. Therefore, we need some node mapping instructions which do not affect thread code.

5.1.3 User-Oriented Node Addressing

When a user gives a system-unique address to each logical node, the arising problem is how to detect whether the given address is already assigned to an existing node or not, and how to locate the node with that address. We need a new node mapping algorithm that not only uniformly distributes logical nodes but also quickly locates a node with a specific address.

Since threads may gather to a specific processor, thus resulting in load imbalance, some node remapping algorithm is necessary to keep good load balancing. The algorithm must however maintain the same property as the node mapping algorithm, which in other words locates a node with a specific address even if it has been remapped to another processor.

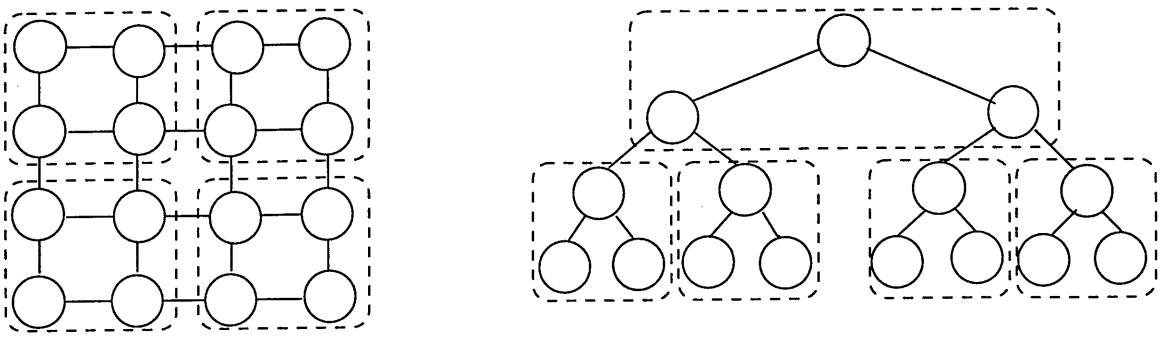


Figure 14: Complicated Node Mapping

5.2 Node Mapping

We propose a new explicit node mapping algorithm which uses a *node mapping function* defined independently from application programs, while maintaining the implicit node mapping used in the current MESSENGERS system.

5.2.1 Node creations with a user-defined address

Assume that each processor or workstation is given a logical sequential number, (i.e. processor ID). Similarly to MESSENGERS, such information may be defined in each user's configuration file. The user also specifies a node mapping function in this file. This function receives as its argument the user-defined address of the next logical node to be created and returns the ID of a processor to allocate this node.

The daemon process running at each processor constructs a binary tree of logical nodes that are mapped to its local processor. (We refer to it as a local B-tree.) The daemon calls the node mapping function whenever it creates a new logical node with a user-defined address. In case when the function returns the current processor ID, the daemon searches for this node in its local B-tree. If the daemon finds the node, it simply creates a logical link to this node. Otherwise, it creates a new node with the user-defined address, register the node in the local B-tree and then establishes only a new link to that node. In case when the node mapping function returns a remote processor ID, the daemon sends the calling thread to that processor, which performs the tree search and creates the node and link.

Figure 15 describes how to map a logical mesh onto four processors as shown

in Figure 14(A). Assuming that a thread is residing on the logical node 6, it is now making a link to the logical node 7 with a navigational command, "*thr_move(1, action = @create, addr = 7)*". Obviously, it does not know if the node 7 exists. The *node_mapping()* function is defined in a user's configuration file. Then, the daemon running at the processor 1 calls the function with its argument, 7 (step 1 in Figure 15). The function returns 2, and thus the calling thread is sent to the processor 2, which then searches for the node 7 in its local B-tree (step 2). The processor 2 finds the node 7 and thus creates only a new link to it (step 3).

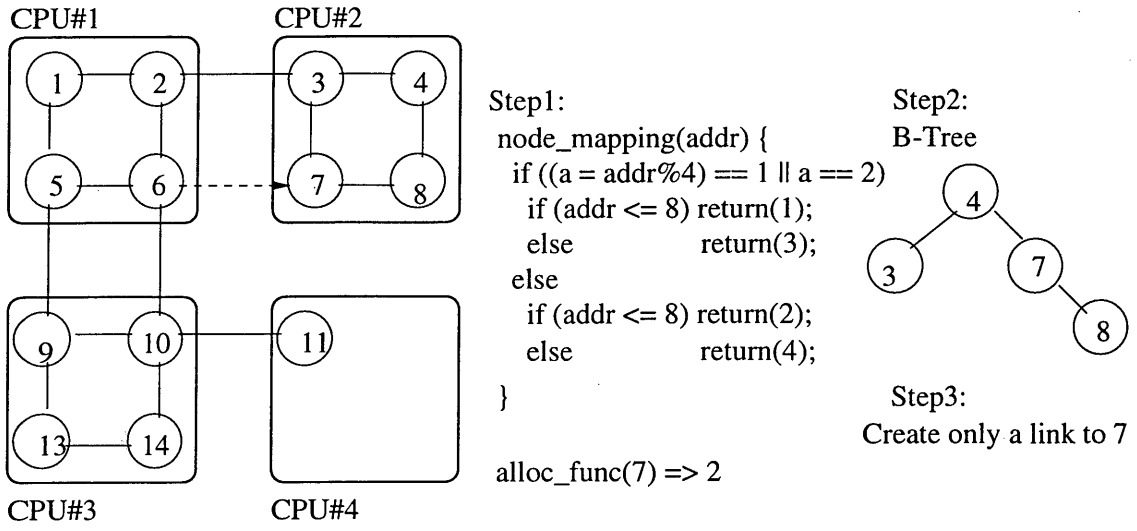


Figure 15: Node creates with a user-defined address

5.2.2 Node creations with an implicit mapping

The thread may create a new node without a specific user-defined address, in which case the MESSENGERS' implicit node mapping chooses a processor to generate the node, using a threshold value. Note that such a node must be still given a system-unique address from the system. For this purpose, negative integers are reserved for the system to assign an address to a new node automatically, while positive integers are used for users to define an address by themselves. Each daemon has a preassigned range of negative integer values to be used as system-generated addresses. This works as a user's node mapping function does, and therefore an existing node is quickly located with its system-generated address.

To given a concrete sequence of the implicit mapping, assume that a thread initiates *thr_move(i = 1, action = @create, daemon = @n)*. This node is generated as follows: the current daemon migrates the calling thread to one of the neighboring daemons, which creates a new node and chooses one from its unused negative integers as its system-generated address. The logical node is then registered in this daemon's

local B-tree, so that it can be located in the similar manner for nodes defined with a user-defined address.

5.3 Node Remapping

The node mapping function and the local B-tree do not work for nodes that have been remapped to somewhere else from its original daemon. To locate remapped nodes, we get prepared for another binary tree at each daemon that keeps track of those nodes and their new daemons. (We distinguish it as a migration B-tree from the local B-tree.) When the daemon migrates a logical node to another daemon, it not only registers this node in its migration B-tree but also tells all its neighboring daemons to perform the same registration. Therefore, when a daemon cannot find a logical node using the node mapping function and the local B-tree, it then uses the migration B-tree and transfers threads to an appropriate daemon. When a thread jumps to the old daemon from non-neighboring ones, the old daemon forwards this thread to the new daemon, as informing the source daemon of the new destination. The source daemon then updates its migration B-tree and sends the subsequent threads directly to the new destination.

The use of migration B-tree has the advantage in not only removing inter-daemons synchronization but also mitigating message retransfer, (i.e. thread retransfer), incurred by node remapping. If the system would not support threads to be forwarded to their new destination, the daemon must request all its neighbors to stop sending threads before starting a node migration. It would also have to wait until it receives all in-transit threads, so that no more threads will be sent to the old node. Our scheme does not require such blocking, since threads can be forwarded to the new node. In addition, the source daemon that erroneously transfers a thread to the old node is informed of this thread's retransfer and registers the new destination in its migration B-tree. It thereafter sends the following threads directly to the new destination.

Figure 16 shows an example of node remapping. Assuming that the processor 1 has originally created the logical node 3, it is currently moving the node to the processor 3. Before the processor 1 informs its neighboring processor 2 of this node remapping, a thread has invokes *thr_move(1, action = @jump, addr = 3)* at the processor 2. The processor 2 calls the *node_mapping()* function, which returns 1. Thus the thread is transferred to the processor 1. The processor 1 no longer has the node 3 in its local B-tree but registers it in the migration B-tree showing that the node 3 has been moved to the processor 3. The thread is therefore forwarded to the final destination, the processor 3. The processor 3 tells the processor 2 to register in its migration B-tree that the node 3 is now on the processor 3.

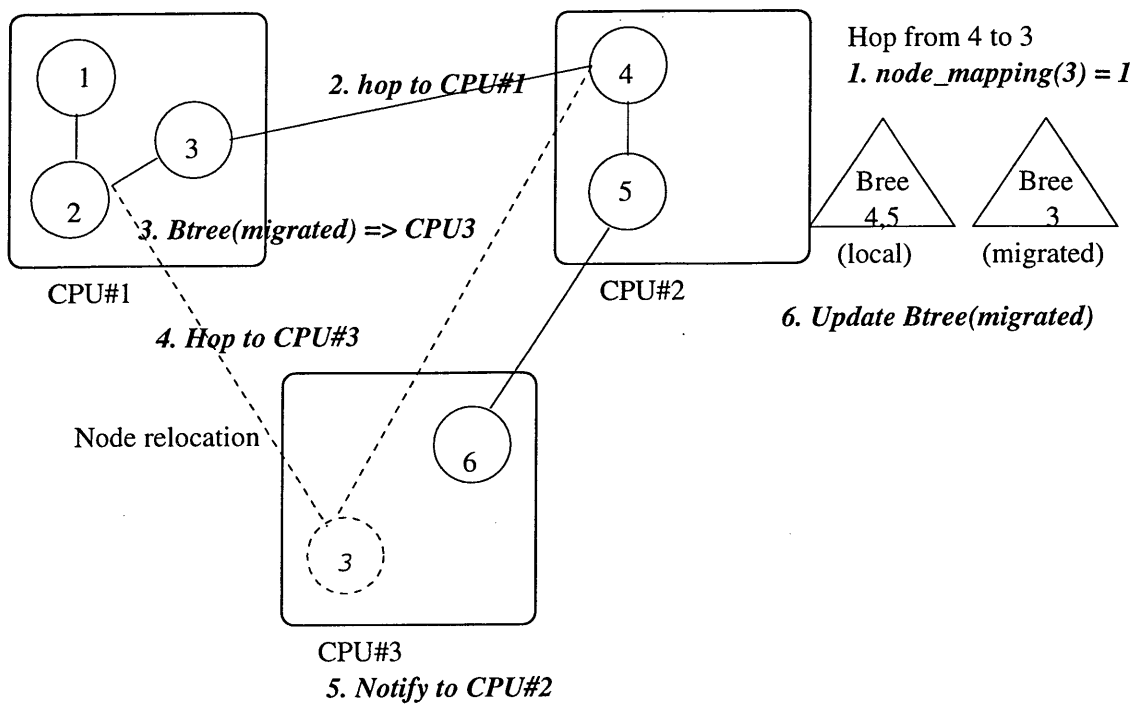


Figure 16: An Example of Node Remapping

5.4 Related Works

There are three works highly related to our node mapping and remapping scheme:

- **Distributed Hashing:** Given a set of processors, the distributed hashing selects one to maintain a new data item, so that a number of data items are uniformly stored in each processors' local memory. It is effective for the analysis of Petri Nets. A system described with Petri Nets has global states or so-called markings, each of which is defined with the distribution of tokens on places in the net. The system analysis sometimes requires constructing a graph of all markings reachable from the initial system state, which is called a reachability graph. NASA Langley Research Center proposed the use of distributed hashing for constructing such reachability graphs and uniformly distributing markings over processors [CGN95]. The distributed hashing is considered as a specific form of our node mapping function. However, there is a big difference between two of them. Distributed hashing does not care the connectivity among markings in the reachability graph, but only focus on their uniform distribution. Our node mapping functions are defined by users, taking such node connectivity into consideration. Another difference is that distributed hashing does not permit marking migration, whereas our local and migration B-trees keep tracks of mi-

grated nodes.

- **Distributed B-Tree:** The distributed B-tree decides the range of data items to be stored in each processor's local memory. Using a binary tree, each processor maintains all data items dynamically mapped to it. Upon a search for a given data item, the processor that is supposed to include this item looks through its binary tree, and then creates the item only when it is not in the tree. University of Erlangen-Nürnberg proposed such a distributed B-tree to maintain the Petri Nets' reachability graphs [ADK97]. When a processor includes more markings than the other processors, it can migrate all markings under a given subtree to its neighboring processor, which is chosen according to marking ranges. This migration scheme therefore not only maintains the property of B-tree but also move a set of markings quickly. Our local B-tree corresponds to the distributed B-tree functionally. However, in the distributed B-tree, each processor must keep the range of markings, which is irrelevant to the actual connectivity among markings. Another difference is that our node remapping allows nodes to be migrated to any processor with the support of the migration B-tree, while the distributed B-tree restricts the migration destination to neighboring processors.

- **DSM Ownership Management:** Distributed shared memory (DSM) with a page-basis memory management permits each processor to search the current owner of a given page and obtain its ownership upon modifying the page content. The old owner then creates a link to the new owner for this page, so that it can keep track of the link when again taking back its ownership. Occasionally, a page ownership is repeatedly passed from one to another processor, which results in generating a daisy chain of links. To prevent this chaining, the new ower tells all processors on the chain to make a direct link to it. This management algorithm was first implemented in Ivy [Li88]. Our scheme is similar to the DSM ownership algorithm in forwarding a thread (or a page ownership request) to the real processor owning the destination node (or the requested page) and informing the source processor of the real destination. The differences between two of them are three-folded: (1) DSM knows the number of pages over the system *a priori* and assigns a different contiguous range of pages to each processor, while our node (re)mapping scheme deals with an unknown number of logical nodes and maps them to each processor using the node mapping function; (2) DSM recursively performs a page directory look-up at each probable page owner till reaching a real owner, whereas our scheme recursively performs a migration B-tree search at each probable node owner till reaching a real owner; (3) In DSM, a page ownership request informs only the probable page owners of the new owner. In our scheme, a node migration informs all the neighboring daemons of the new destination daemon, which, when receiving forwarded threads, notifies their source daemons of the real destination.

6 Conclusion

We proposed three main ideas to apply autonomous objects for parallel graph computation: (1) instantiating autonomous objects with threads, (2) developing an efficient navigational statement for thread migrations, and (3) implementing node-oriented load balancing.

We need to first evaluate the efficiency of each functionality. For the use of thread, we must investigate which of the following two schemes is better: (1) all autonomous objects at each daemon are simply executed as a sequence of precompiled functions by the main thread, or (2) each autonomous object is instantiated with an independent thread and executed concurrently. The former is the current work of the MESSENGERS project. The main thread emulates object context switches efficiently but may be still blocked upon I/O operations. The latter is the one we proposed in this report. Threads relinquishes upon I/O operations, thus keeping the CPU busy.

Another performance improvement we are expecting is the quick search for destination specs with user-defined node addresses and link IDs. It is necessary to evaluate how much better performance will be brought by this scheme than the inefficient string-oriented search for destination nodes. We also need to describe various graphs to prove the flexibility of the entire features which our navigational calculus provides.

For the node-oriented load balancing, we should investigate how much overhead will be resulted by moving a thread from one to another logical node on the same daemon. If such an intra-daemon thread migration incurs performance degradation unacceptably larger than a simple thread context switch, users may not want to map multiple logical nodes on a single daemon, which thus makes our load balancing scheme useless. Provided the performance degradations is negligible, we must then evaluate the cost of a node migration, which is used to decide when a node should migrate for better performance.

After clearing all the performance concern, we will realize a new computing environment with all the efficient schemes on parallel machines such as Tsukuba University's CP-PACS [BNNI97] and Maestro [YYPK+98], and UC Irvine's cluster of SUN/Solaris workstations.

References

[ADK97] S. C. Allmaier, S. Dalibor, and D. Keische. Parallel graph generation algorithms for shared and distributed memory machines. In *Proc. of the 1997 Parallel Computing Conference - ParCo'97, Bonn, Germany*, Amsterdam, 16-19 September 1997. North-Holland.

[BDF98] Lubomir F. Bic, Michael B. Dillencourt, and Munehiro Fukuda. Mobile network objects. In *Encyclopedia of Electrical and Electronics Engineering*, page to appear. John Wiley & Sons, Inc., 1998.

[Ber87] Francine Berman. Experience with an automatic solution to the mapping problem. In Jamieson, editor, *The Characteristics of Parallel Algorithms*, pages 307–334. MIT Press, 1987.

[BFD96] L.F. Bic, M. Fukuda, and M. Dillencourt. Distributed computing using autonomous objects. *IEEE Computer*, Vol.29(No.8):55–61, Aug. 1996.

[Bid96] B. Bidyuk. MESSENGERS-C compiler manual. Report MSGR-06, University of California, Irvine, 1996.
`http://www.ics.uci.edu/~bic/messengers`.

[BL87] L.F. Bic and C. Lee. A data-driven model for a subset of logic programming. *ACM TOPLAS*, 9(4), Oct. 1987.

[BNNI97] T. Boku, H. Nakamura, K. Nakazawa, and Y. Iwasaki. The architecture of massively parallel processor CP-PACS. In *Proc. of the 2nd IEEE Int'l Symposium on Parallel Algorithms/Architecture Synthesis*, pages 31–40, Aizu-Wakamatsu, Japan, 17-21 March 1997.

[CGN95] G. Ciardo, J. Gluckman, and D. Nicol. Distributed state-space generation of discrete-state stochastic models. Technical report 198233, ICASE, NASA Langley Research Center, Hampton, VA, 1995.

[Cha82] E.J.H. Chang. Echo algorithms: depth parallel operations on general graphs. *IEEE Trans. Softw. Eng.*, SE-8(4), 1982.

[CHM97] David Cronk, Matthew Haines, and Piyush Mehrotra. Thread migration in the presnse of pointers. In H. El-Rewini and Y. N. Patt, editors, *Proc. of the 30th Hawaii Int'l Conf. on Systems Sciences - HICCS'97*, pages 292–298, Los Alamitos, California, 7-10 January 1997. IEEE Computer Society Press.

[CKO+94] J. Casas, R. Konuru, S. Otto, R. Prouty, and J. Walpole. Adaptive load migration systems for pvm. In *Proc. of Supoercomputing '94*, pages 390–399, Washington D.C., November 1994. ACM/IEEE.

[FBDM98] Munehiro Fukuda, Lubomir F. Bic, Michael B. Dillencourt, and Fehmina Merchant. Distributed coordination with MESSENGERS. *Science of Computer Programming*, 31(2):to appear, in 1998.

[Fuk97] Munehiro Fukuda. *MESSENGERS: A Distributed Computing System Based on Autonomous Objects.* PhD thesis, University of California, Irvine, CA 92697, June 1997.

[GM95] J. Gosling and H. McGilton. *The Java Language Environment.* Mountain View, CA 94043, October 1995.

[Gra96] R. S. Gray. Agent Tcl: A flexible and secure mobile-agent system. In *Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96)*, Monterey, California, July 1996.
`http://www.cs.dartmouth.edu/~agent/papers/index.html`.

[JG96] Stephen Jenks and Jean-Luc Gaudiot. Nomadic Threads: A migrating multithreaded approach to remote memory accesses in multiprocessors. In *Proc. of the 1996 Conf. on Parallel Architectures and Compilation Techniques (PACT'96)*, pages 2–11, Bosto, Massachusetts, 21-23 Octobr 1996.

[JvS95] D. Johansen, R. van Renesse, and F. B. Schneider. An introduction to the TACOMA distributed system version 1.0. Technical Report 05-23, Department of Computer Science, University of Tromsø, June 1995.
`http://www.cs.uit.no/DOS/Tacoma/index.html`.

[Li88] Kai Li. A shared virtual memory system for parallel computing. In *Proc. of the 1988 Int'l Conf. on Parallel Procesing*, 1988.

[LRG+95] Virginia M. Lo, Sanjay Rajopadhye, Samik Gupta, David Keldsen, Moataz A. Mohamed, Bill Nitzberg, Jan Arne Telle, and Xiaoxiong Zhong. OREGAMI: Tools for mapping parallel computations to parallel architectures. In Behrooz A. Shirazi, Ali R. Hurson, and Krishna M. Kavi, editors, *Scheduling and Load Balancing in Parallel and Distributed Systems*, pages 284–308, Los Alamitos, California, 1995. IEEE Computer Society Press.

[LS95] Helmut Lorek and Michael Sonnenschein. Using parallel computers to simulate individual-orieted models in ecology: A case study. In *Proc. ESM'95 European Simulation Multiconference: Modelling and Simulation*, pages 526–531, Prag, 5-7 June 1995. SCS International.
`http://offis.offis.uni-oldenburg.de/projekte/ecotools/`.

[MR96] E. Mascaarenhas and V. Rego. Ariadne: architecture of a portable threads system supporting thread migration. *Software - Practice and Experience*, Vol.26(No.3):327–356, March 1996.

[PL96] Przemyslaw Prusinkiewicz and Aristid Lindenmayer. *The Algorithmic Beauty of Plants.* Springer-Verlag New York, Inc., New York, 1996.

[RCRH95] Annie Rogers, Martin C. Carlisle, John H. Reppy, and Laurie J. Hendren. Supporting dynamic data structures on distributed-memory machines. *ACM TOPLAS*, Vol.17(No.2):233–263, March 1995.

[RS74] G. Rozenberg and A. Salomaa, editors. *L Systems, LNCS 15.* Springer-Verlag, Berlin, Germany, 1974.

[RTL+91] R.K. Raj, E Tempero, H.M. Levy, A.P. Black, et al. Emerald - a general-purpose programming language. *Software-Practice & Experience*, 21(1), Jan. 1991.

[SB94] P.S. Sapaty and P.M. Borst. An overview of the WAVE language and system for distributed processing of open networks. Technical report, University of Surrey, UK, 1994.
`http://ww.ira.uka.de/132/wave/wave.html`.

[WBDF98] Christian Wicke, Lubomir F. Bic, Michael B. Dillencourt, and Munehiro Fukuda. Automatic state capture of self-migrating computations in MESSENGERS. In *Proc. of the 2nd Int'l Workshop on Mobile Agents - MA'98*, page to appear, Stuttgart, Germany, 9-11 September 1998. Springer-Verlag LNCS.

[YYPK+98] T. Yamazaki, N. Yonezawa, S. Yamagiwa P. Kulkasem, M. Ono, A. N. M. Al-Khoury, and K. Wada. SVCP: A cache coherency protocol with explicit update subscription. In *Proc. of the 1998 Int'l Conf. on Parallel and Distributed Processing Techniques and Applications (PDPDA'98)*, page to appear, Las Vegas, Nevada, 13-16 July 1998.