INTERNATIONAL JOURNAL OF COMPUTERS
Issue 3, Volume 2, 2008

Advantages of Self-Migration for Distributed Computing

Lubomir F. Bic and Michael B. Dillencourt

***Abstract*—**We compare four paradigms that have recently been the subject of recent research: mobile agents, distributed shared memory (DSM), coordination paradigms, and self-migrating computations. We place these paradigms in a common framework and demonstrate that self-migrating computations subsume the other three paradigms in terms of their capabilities to organize and coordinate computation, and map the concurrent activities onto a multicomputer architecture. We then demonstrate the advantages of self-migration in terms of algorithmic integrity, performance, the ability to generate parallel programs, and the ability to support incremental parallelization.

***Keywords*—**Coordination, DSM, mobile agents, parallel and distributed computing, Self-migrating computations.

I. Introduction

MOBILE agents, distributed shared memory (DSM), coordination paradigms, and self-migrating threads represent four lines of research that have each gained considerable attention in recent years. Mobile agents provide autonomous service entities capable of roaming communication networks in search of information and services. DSM aims at providing an abstraction for distributed memory computers such that applications could be written using shared memory programming paradigms. Coordination paradigms also focus on providing structured abstractions of the data or information space but, in addition, provide new conceptual models for expressing concurrency and coordination among the activities operating on the structured logical space. While these three research areas appear to be unrelated, there are similarities among them that are best understood by examining them in a common framework together with self-migrating threads. Self-migrating threads navigate through a logical space, based on their own internal program and state, and collectively solve a global problem through their individual efforts. The self-migrating threads draw heavily on ideas from the other three areas [1].

Self-migrating computations offer several important advantages over the other paradigms. Specifically, (1) they allow certain sequential computations to run faster by distributing the underlying data; (2) they facilitate the parallelization of sequential algorithms by preserving the essential structure of the original computations; (3) they lead to the parallelization of certain algorithms traditionally considered unparallelizable; and (4) they lend themselves to incremental parallelization of sequential programs.

Manuscript received December 9, 2008: Revised version received March 4, 2009.

L. F. Bic is with the Department of Computer Science, University of California, Irvine, CA 92617, USA (phone: 949-824-5248; e-mail: bic@ics.uci.edu).

M. B. Dillencourt is with the Department of Computer Science, University of California, Irvine, CA 92617, USA (e-mail: dillenco@ics.cui.edu).

II. Four Paradigms

A. Mobile Agents

Mobile agents are self-contained entities that can navigate autonomously through the underlying network and perform a variety of tasks in the nodes they visit. Fig. 1(a) captures the essence of most mobile agent systems, which focus on the following major aspects:

1) The computational model underlying mobile agents system is similar to a multithreaded environment, where individual threads consist of a program and a state, and communicate with one another via shared or distributed memory mechanisms. The main extension to this model is navigation. The computational model provides special commands or other linguistic constructs that enable agents to relocate themselves or their clones to other physical nodes in the network and to continue executing in the new environment.
2) To serve a useful function, a mobile agent must be able to interact with the environment of the host on which it currently resides. This is accomplished by providing an interface to the host's operating system, which permits the agent to access data and/or invoke services available on the current host.
3) To permit autonomous navigation, a layer of software consisting of daemons is superimposed on the underlying physical network. The task of each daemon process is to receive agents, interpret their behavior, and send them on to other daemons as necessary. The daemons themselves have no intelligence; all functionality is carried as part of the mobile agents. The daemons use existing physical links to communicate with one another. Hence the mapping of resulting daemon network onto the physical network is trivial; the former is a subset of the latter as determined by the user.

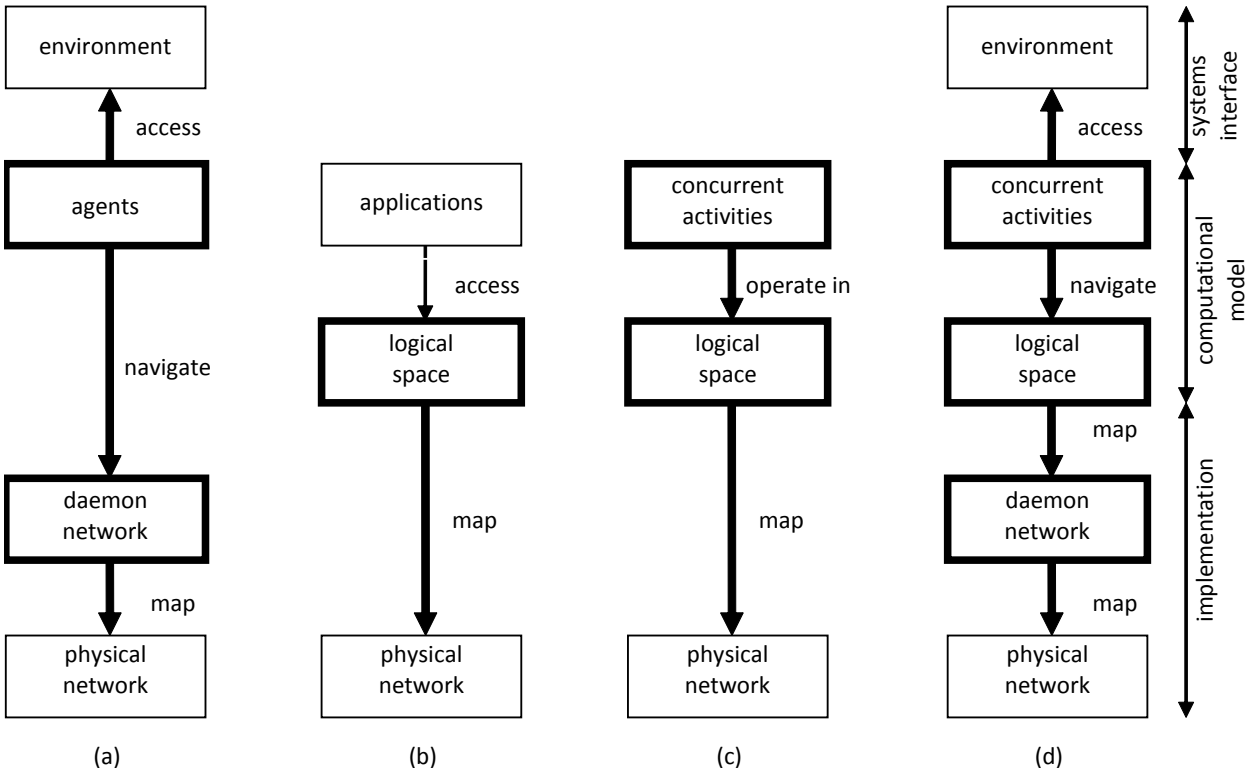

Fig. 1 Comparison framework. (a) Mobile agents. (b) DSM. (c) Coordination paradigms. (d) Self-migrating computations

A number of mobile agent projects have been carried out in recent years [2, 3, 4, 5]. Most focus on "intelligent" agents, i.e., those that can serve as personal assistants, roaming the Internet and perform arbitrarily complex services on behalf of their users. One of the first proposals was Telescript [6], which was centered on the design of a special-purpose language for expressing agents' behaviors, including their ability to move themselves around the Internet. More recent approaches rely on existing languages, such as Java (used by IBM Aglets [7]) or Tcl/Tk (used by Agent Tcl [8] and Tacoma [9].)

Another focus of mobile agent research has been on intelligent communication. This is best represented by the Messenger projects of the U. Geneva [10]. Mobile agents are viewed as "prototypic" mobile agents on top of which more complex "intelligent" agents can be built. The objective of this project is to replace traditional messages and communication protocols by mobile agents.

There are a number of advantages of mobile agents over traditional approaches. First, they offer a more natural metaphor for both users and programmers in that they replace the traditional client/server or send/receive points of view by self-contained activities that encapsulate both communication and remote computing. A second advantage is the inherently open-ended nature of mobile agents, which permits new functionality to be introduced at runtime as needed. Finally, mobile agents can significantly reduce message traffic in client/server type applications. Instead of engaging in a bandwidth and latency intensive message exchange with a server, the client may dispatch an agent to the server site, which performs all the necessary interactions locally. When the task is completed, it reports the answer to the original client. Hence only a single "round trip," traveled by the object, is necessary between the client and the server.

The heavy lines of Fig. 1(a) indicate the emphasis of mobile-agent system on the agents' navigational capabilities, the daemon infrastructure, including its mapping to the physical network, and the agents' interface to the host's environment.

B. Distributed Shared Memory

Distributed shared memory (DSM) systems provide the *illusion* of a common shared memory on a multi-computer, where each node only has a private local memory and can communicate with other nodes via a network or a switch. Fig. 1(b) captures the essence of most DSM systems, indicating the main emphasis of this line of research:

1) The shared space provided by a DSM system is a passive component, which is accessed by the various applications running on the system. The organization and structure of the shared logical space is what distinguishes different DSM approaches. These represent the trade-offs between the system's expressiveness and the resulting performance.
2) The implementation provides the mapping of the logical space onto the physical architecture. Its complexity depends on the size of the semantic gap that must be bridged.

One of first approaches to providing DSM was based on

paging [11]. Similar to a paged-based virtual memory in a single-processor system, the virtual shared space is partitioned into fixed-size pages. However, instead of moving them between primary and secondary memory as needed, they are moved between different processors. A number of implementations have been proposed to keep track of the migrating pages to facilitate performance while ensuring that memory consistency is not violated.

The above approach to DSM guarantees sequential memory consistency, which is the most convenient from the programming point of view but also the most costly to implement. Other approaches have taken a more restricted view of what a DSM is to gain better performance [12]. These restrictions require the memory consistency model to be weakened. For example, a causally consistent DSM guarantees that different processes see only causally-related accesses to the shared variables in the same order, while causally-unrelated accesses may be observed in a different order. In addition, the view of the shared memory may change. That is, instead of providing a one-dimensional flat sequence of data locations, thus mimicking the view of physical RAM, the shared portion may be restricted to only certain variables or data structures. In this case, special synchronization primitives are typically provided, with the understanding that the consistency of the shared data is guaranteed only in conjunction with these primitives. For example, release and entry consistency guarantee a consistent view of shared data only when a critical section is exited or entered, respectively.

Regardless of the particular scheme or implementation, the focus of all DSM-based schemes is the logical space organization and its mapping onto the underlying physical architecture, as shown by the heavy lines of Fig. 1(b).

C. Coordination Paradigms

Coordination paradigms are closely related to DSM and it is difficult to draw a clear line between the two research thrusts. We characterize coordination paradigms as approaches that go significantly beyond the scope of DSM by addressing not only the aspect of space but also integrate its operational aspects into a common model:

1) Like DSM, coordination paradigms provide the abstraction of a logical space, which consists of data and possibly functions, and which is structured specifically to facilitate the development of distributed applications. Unlike DSM, it is not always the data that is brought transparently to the current processes or thread as needed. Rather, a coordination paradigm may support the ability of an activity to relocate itself to another (physical or logical) domain to gain access to some data.
2) In addition to the above *spatial* aspect, coordination paradigms also incorporate a *temporal* aspect by providing specific mechanisms or constructs to operate on the logical space, thus coordinating the concurrent activities comprising the computation. These, in general, are closely integrated with the logical space. They typically include mechanisms for controlling synchronization, communication, and creation/destruction of the computational activities required to orchestrate the operation of a complex system. Hence, from the programming point of view, coordination paradigms may be viewed as extensions of the DSM concept.

The two abstract layers, which are the main focus of all coordination paradigms (as indicated by heavy lines of Fig. 1(c)), are then mapped onto the underlying computational structure—a network or a multiprocessor. The mapping, however, is typically outside of the scope of the coordination paradigm.

A large number of coordination paradigms have been proposed and developed in recent years, which can be subdivided into several broad categories. One approach to coordination utilizes channel-based communication between processes. Processes communicate directly with each other by reading from and writing to ports. Ports of processes are connected to ports of other processes via channels. This approach leads to a clean separation of computation and coordination functions. An example of the channel-based approach is the IWIM model [13].

Another approach to coordination is *medium-based* coordination. At a very abstract level, all medium-based approaches to coordination work on the same principle. There is a common medium or state space, shared by the processes. Processes can modify the state space, and these modifications affect the behavior of other processes. Computation is performed by the processes, and coordination is achieved through the shared state space.

One of the most prominent examples of the medium-based approach is Gamma [14], based on a chemical reaction metaphor. The state space is a multiset of objects. Gamma programs consist of matched (reaction conditions, action) pairs. Execution proceeds by replacing a collection of objects that satisfy a reaction condition by the result of applying the corresponding action. As programs are executed, they may cause multiset transformations that create the reaction conditions necessary to allow other programs to execute.

Another well-known example of coordination through a shared state space is the Linda system [15]. The state space is a pool of data called a *tuple space*. Processes may insert, read, and remove tuples from the tuple space using various primitives. They may also spawn new activities that leave new tuples in the tuple upon their termination. Processes select tuples associatively, by issuing requests for tuples that match certain templates.

In the Linda model, the state space is shared by all processes. PoliS [16] is an enhancement to the basic model intended to simplify the design of distributed systems. PoliS allows multiple named tuple spaces, called *places*, where each tuple belongs to exactly one tuple space. The execution threads in PoliS are autonomous active tuples, called *agents*. Because agents are tuples, an agent belongs to exactly one tuple space. An agent can read tuples inside its own tuple space and can write tuples to any tuple space. These simple

operations provide a uniform approach to spawning new activity, the migration of such activities, and the exchange of information among them.

Yet another approach to coordination is to provide templates for common communication patterns [17] or to provide an API that allows the composition of applications from exiting building blocks [18].

Despite the significant differences among the various coordination paradigms, they all share the common characteristics captured by the two highlighted layers of Fig. 1(c).

D. Self-Migrating Computations

MESSENGERS [19, 20, 21, 22] is a system based on the principles of self-migrating threads, called *Messengers*. The system distinguishes three separate levels of networks. The *physical network* is the underlying computational resource. The *daemon network* is a collection of Unix processes, whose task is to interpret the behavior of the self-migrating threads. The *logical network* is an application-specific computation network created at run time on top of the daemon network. Multiple logical network nodes may be created on the same daemon network nodes, thus running on the same physical node, and they may be interconnected by logical links into an arbitrary topology.

The self-migrating threads navigate through the logical network based on their own internal program and state. They are also capable of cloning themselves, both implicitly and explicitly, to follow multiple links or to perform different subtasks. This is accomplished by explicit *navigational* statements, which also permit the creation or destruction of logical links and/or nodes. A number of optional parameters may be specified as part of the navigational statements, including the specification of particular nodes, or links. Wild cards may also be used for partial matching. The self-migrating thread is replicated and propagated to all destinations that match the navigational specification.

Self-migrating threads may also perform arbitrary computations in the nodes they visit. This can take two forms. First, the object's internal program may contain *computational* statements, which permit arbitrary arithmetic, logic, and control operations to be performed. Second, the objects may invoke ordinary C functions as part of their behavior or spawn complete programs as separate concurrent Unix processes. The system also supports implicit mapping of the logical network onto the daemon network.

III. Advantages Of Self-Migration

The use of explicit commands to support the migration of computations leads to a new style of programming, referred to as Navigational Programming (NavP) [23]. Using this paradigm, a distributed computation is not viewed as a collection of stationary parallel processes communication with each other via messages. Instead, it is a collection of sequential threads, each of which computes, navigates through the underlying network, and communicates with other such threads. This style of programming is applicable specifically to scientific computing, it is easier to use than message passing, and it leads to increased performance.

To illustrate these principles further, consider the analogy of a train schedule. Fig. 2 shows graphically the course of four trains, Tr1 through Tr4. At time t1, each trains starts from a different station s1 though s4 and proceeds to a new station at each of the times t2 though t4 as indicated by the arrow.

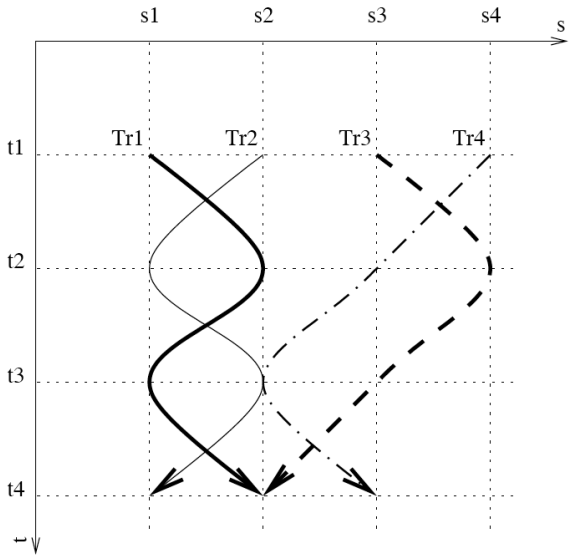

Fig. 2 Train schedules

The graphical information in Fig. 2 can be represented from two different points of view, depending on its intended use. Fig. 3 shows the information from each station's point of view, i.e., the arrivals and departures of trains at each station. This information would be useful for someone standing in a given station and it corresponds to the message-passing view.

	s1	s2	s3	s4
t1	Tr1	Tr2	Tr3	Tr4
t2	Tr2	Tr1	Tr4	Tr3
t3	Tr1	Tr2,Tr4	Tr3	
t4	Tr2	Tr1,Tr3	Tr4	

Fig. 3 Arrivals and Departures

Fig. 4 shows the same information from each train's point of view, i.e., for each train it shows which station it will visit at which time. This information would be useful for someone traveling on that train and it corresponds to the NavP point of view.

	Tr1	Tr2	Tr3	Tr4
t1	s1	s2	s3	s4
t2	s2	s1	s4	s3
t3	s1	s2	s3	s2
t4	s2	s1	s2	s3

Fig. 4 train itineraries

A. Distributed Sequential Computing (DSC)

When the data set of a computation is too large for the main memory of a single computer, it is advantages to distribute it over the collective memories multiple interconnected machines because it eliminates paging overhead. The main question is: where should the computation be performed? Without migration, the programmer must select one of the machines as the pivot. The data that does not reside in the local memory must be transferred to the pivot machine as needed for its computation. This generally results in a performance improvement because the network is faster than the paging disk. NavP offer an even better way: it allows the computation to move to the data it needs to access. This generally improves performance because it avoids the movement of large amounts of data.

To illustrate the principle, consider the following program fragment, performs some sequential computation over a large array A:

```
double A[huge];
for (i = 0; i < huge; i++)
  x = compute(x, A[i]);
```

Assuming A[huge] is too large for a single memory, it is partitioned into n arrays A[smaller] such that smaller<memory_size. Each partition A[smaller] is allocated on a different machine and the following modified code is started on machine 0:

```
for (i = 0; i < huge; i++) {
  hop(node(A[i]));
  x = compute(x, A[i]);
```

The hop statement makes sure that the computation always resides on the machine that holds the current element A[i]. Note that this statement is mostly a no-op; only when the array crosses machine boundary does an actual migration take place.

Fig. 5 illustrates the performance improvement achieved by DSC [24]. The curve shows clearly that the point at which the performance degrades rapidly due to paging can be postponed by using more machines and thus solve increasingly larger problems.

The ability to utilize the collective memory of multiple machines to avoid paging can be exploited using some of the other paradigms of Fig. 1, specifically DSM and some of the coordination systems. However, they do not allow computations to migrate and hence the data must be moved to the computation. In contrast, self-migrating computations can take full advantage of the underlying network by moving either computations or data, depending on which results is less overhead.

B. Algorithmic Integrity

Message passing is the most common approach to developing distributed programs (sequential or parallel). Unfortunately, given a centralized sequential program or algorithm, there is no easy way to derive a distributed message-passing program from it. Instead, a new program

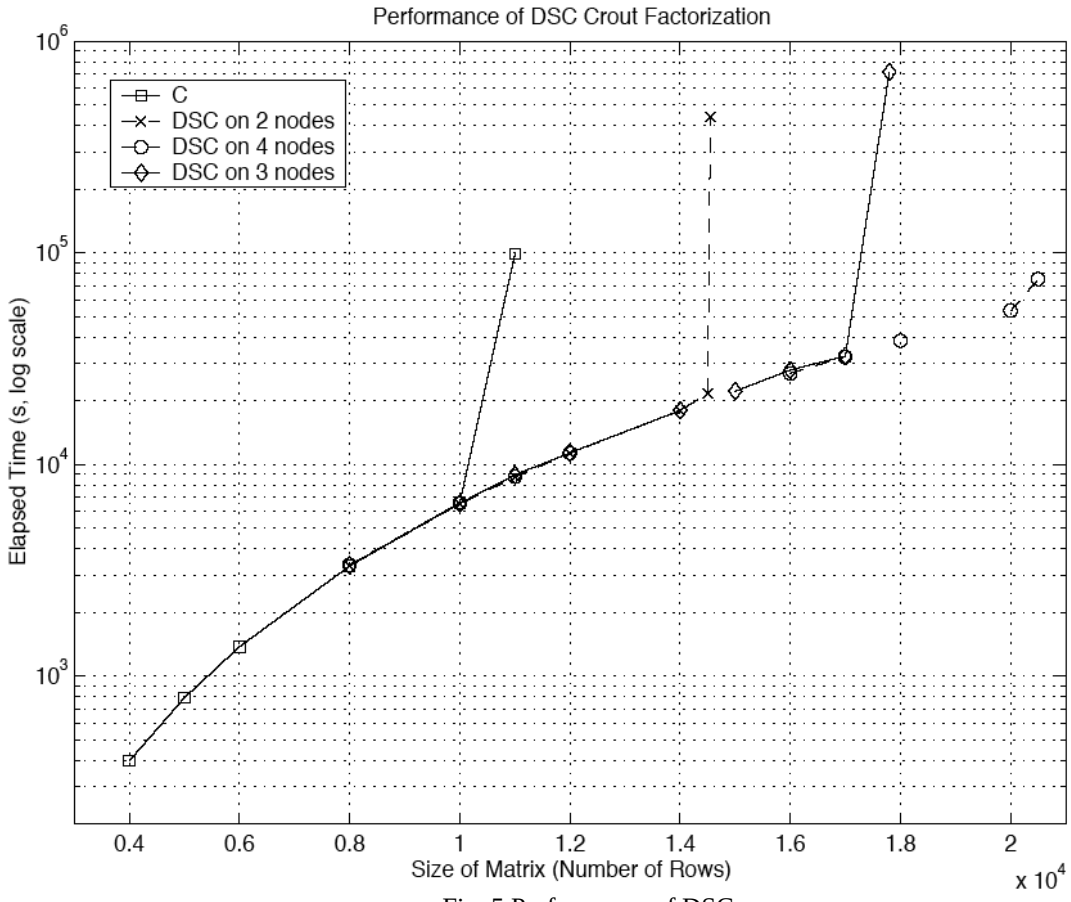

Fig. 5 Performance of DSC

must be developed, which generally bears little similarity to the original. NavP preserves the algorithmic integrity of centralized sequential programs because the process of distribution and/or parallelization requires the insertion of hop commands into the original sequential flow.

To illustrate this concept, consider the following program fragment:

```
v1 = diag(A)
v2 = f1(B,v1)
v3 = f2(A,v2)
```

Assume that matrix A resides on a node n1 and matrix B on a different node n2. The following message-passing code in SPMD style would accomplish the same task:

```
if (rank = n1)
  v1 = diag(A)
  send(v1, n2)
  recv(v2, n2)
  v3 = f2(A,v2)
else if (rank = n2)
  recv(v1, n1)
  v2 = f1(B,v1)
  send(v2, n1)
end if
```

We make two important observations. First, even though this is an extremely simple sequential algorithms and the resulting distributed version is still only sequential in its execution, the new program is much larger than the original. Second, the program structure has been significantly modified: to follow the original sequential flow, one must alternate between the if and else clauses by matching the respective send a receive statements.

Consider now the corresponding NavP code:

```
v1 = diag(A)
hop(n2)
v2 = f1(B,v1)
hop(n1)
v3 = f2(A,v2)
```

The only difference is the insertion of the two hop statements; the original sequential flow has been preserved.

Of the three other paradigms of Fig.1, only mobile agents have the ability to explicitly migrate their computations through the network. This could be used to preserve the algorithmic integrity of sequential programs when adapting them to a distributed environment. However, mobile agents generally do not support a separate logical space. Hence the computation could only be distributed with respect to the current physical network and would be dependent on the current network topology. Self-migrating computations offer a greater flexibility by supporting an application-specific logical space.

C. Parallelization of Sequential Algorithms

There are several classes of algorithms that are generally considered as unparallelizable due to their specific data dependencies. One such class are the so-called left-looking algorithm, characterized by the fact that the computation of any given array element uses all preceding elements in that array. The following code fragment represents such a left-looking algorithm.

```
do j = 2 to n
  do i = 1 to j-1
    a[j] = (a[j]+a[i])*j/(j+i)
  end do
  a[j] = a[j]/j
end do
```

Fig. 6 shows the data dependence: to compute the elements labeled as consumer (black), all preceding elements, labeled as producers (white) must be available. Thus it appears that there is no opportunity for parallelism, because the computation of the next element cannot start until all its predecessors have already been computed.

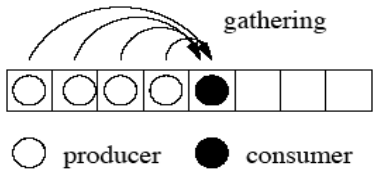

Fig. 6 A left-looking dependency

NavP offers an elegant way to parallelize code such as the above. The first step is to distribute the data over n machines. Fig. 7 shows the distribution of the array over five PEs as an example. The next step is to modify the original code by inserting the necessary hop statements to make sure the computation always resides on the machine that holds the currently accessed array elements. A second minor modification is to introduce "transport" variables to carry copies of relevant data as the computation migrates through the network. The following code shows these modifications, where the new variable mx carries a copy of the currently computed array element a[j]. As with the example of subsection III.A, most of the hop statements will turn into no-ops, since the next array element resides on the same machine.

```
do j = 2 to n
  hop(node[j]); mx = a[j]
  do i = 1 to j-1
    hop(node[i])
    mx = (mx+a[i])*j/(j+i)
  end do
  hop(node[j]); a[j] = mx
  a[j] = a[j]/j
end do
```

The above computation is distributed but it is still only sequential. To parallelize it, we need the following important insight: The computation of any given element a[j] does not take place on only its owner PE; rather it is distributed over multiple PEs. For example, the computation of element a[j] in Fig. 7 starts on PE4 but it continues on PE1, PE2, PE3, and finally terminates back on PE4. However, PE4 does not need to wait until a[j] returns before staring the computation of the next element a[j+1]; this can start as soon as the computation

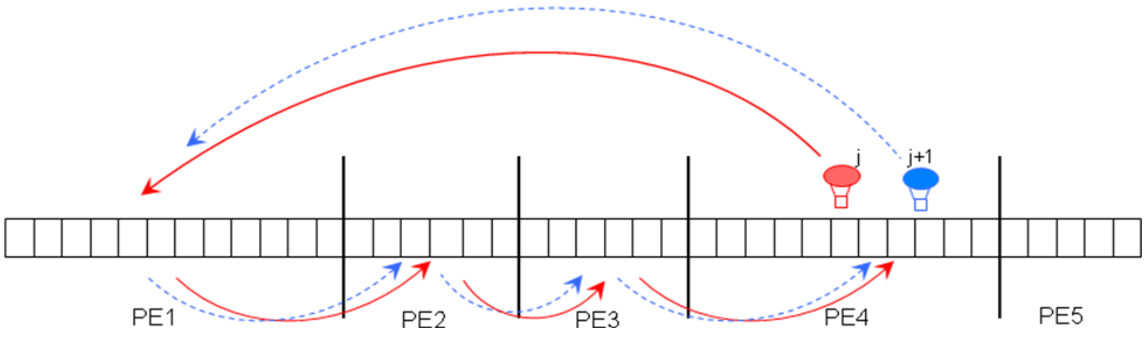

Fig. 7 A Mobile pipeline

of a[i] hops away to PE1. This results in potential parallelism due to the overlap of the individual array element computations. To achieve this parallelism, we make one final modification to the code, namely, by starting each iteration of the outer loop as an independent thread, similar to a *doall* operation:

```
parthread thrd(j)
  hop(node[j]); mx = a[j]
  do i = 1 to j-1
    hop(node[i])
    mx = (mx+a[i])*j/(j+i)
  end do
  hop(node[j]); a[j] = mx
  a[j] = a[j]/j
end thread
```

Fig. 7 illustrates the resulting parallelism by showing the paths of the two computations for a[j] and a[j+1]. We refer to the resulting structure as a *mobile pipeline* [25].

To further illustrate the difference between a conventional and a mobile pipeline, compare Figs. 8 and 9. With a conventional pipeline (Fig. 8), the data (a through e) is pumped through a series stationary computations (C1, C2, C3). With a mobile pipeline (Fig. 9), the computations, each implemented as a self-migrating thread (C1, C2, C3) follow each over as they pass over the stationary series of data (a through e).

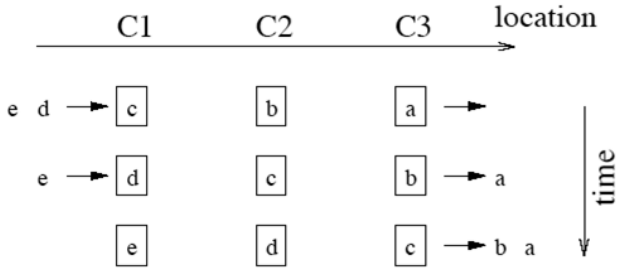

Fig. 8 Conventional Pipeline Structure

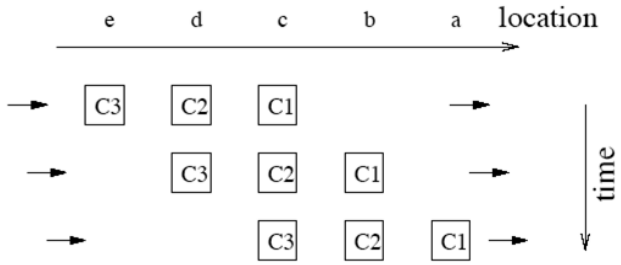

Fig. 9 Mobile Pipeline Structure

D. *Incremental Parallelization*

Traditional message-passing approaches to parallel program development require a brand new program to be developed. With self-migrating computations, in contrast, it is possible to start with a sequential program or algorithm and transform this incrementally into a parallel version. Each intermediate version is executable and has generally a better performance than its predecessor. Hence the programmer can improve the performance gradually, rather than having to commit to an all-or-nothing approach supported by message passing [26, 27].

Fig. 10 shows the steps of the incremental parallelization approach. The first step is to perform a data distribution of the underlying large data structures. Data distribution is important for the overall communication cost and parallelism. The problem of high communication cost caused by improper data distributions cannot be corrected by other later efforts. Shared-memory programming models such as OpenMP on DSM rely on their runtime systems to find data layouts; but they do not yet deliver as good a performance as MPI programs on distributed memory machines [28]. In the case of MP or MP-based SPMD models, the data layouts are either explicitly specified by the programmer as in HPF or automatically generated by parallelizing compilers [29, 30, 31, 32, 33]. These automatic approaches decompose the data mapping process into two steps: alignment and distribution, and attempt to find data layout choices either analytically or by resorting to integer programming. The underlying mathematical representation used is a so-called *component affinity graph* [34] where the nodes represent the dimensions of arrays and the weights associated with the edges are relations derived from the data reference patterns and thus suggest how the

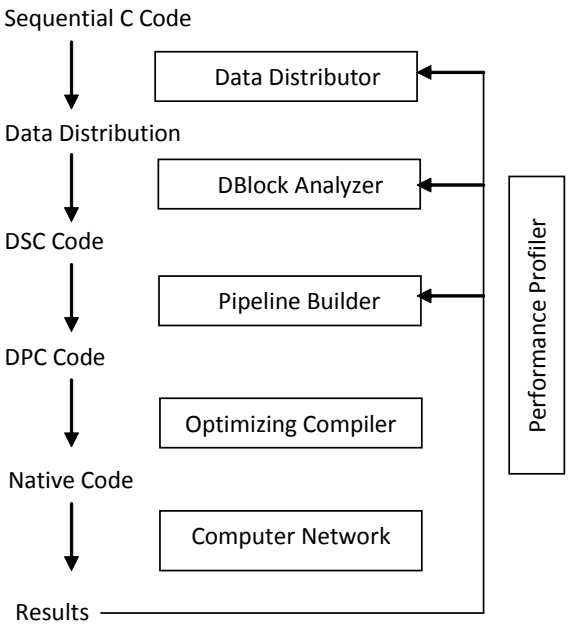

Fig. 10 Incremental Parallelization

dimensions of different arrays should be aligned and then distributed together. While promising to find good data layouts for some benchmarks, these automatic approaches are mainly confined to research prototype compilers.

What is common to all of the above approaches is that they are limited to regular data distributions (e.g., along columns, rows, or blocks). This is because, in the absence of code migration, the partitioning of the data dictates the partitioning of the programming task. In contrast, programs with mobility can follow the data, so their structure is not dependent on the data decomposition. As a result, navigational programs can take advantage of unstructured data patterns in order to further reduce communication overhead.

Our methodology [35] is based on constructing a *Navigational Trace Graph (NTG)*, an undirected weighted graph, which is fundamentally different from the component affinity graph [34] and its variants. In NTG, the nodes are individual entries of all distributed arrays and the weight associated with an edge represents the trace between the two incident array entries as the DSC thread navigates through them. By representing the trace relations at the level of individual array entries, both alignment and distribution problems are solved in a unified manner.

There are three kinds of edges in the NTG. First, Locality (or L) Edges are introduced between the neighboring entries of an array. These edges represent the locality of data access exhibited in many algorithms, and they aim at obtaining regular data layouts for each array. Second, a Producer-consumer (or PC) Edge with the weight p is introduced between the LHS (left-hand side) and every RHS (right-hand side) array entry. The weight represents the communication cost incurred if the two linked entries do not reside on the same PE. Finally, every array entry in one statement is connected with every entry in its successive (in time) statement with a Continuity (or C) Edge with the weight c. When several equally competitive data layout choices may be found using only L and PC edges, the presence of C edges will break the tie by favoring the choice that allows successive (in time) statements to be executed on the same PE. Once the NTG has been generated, we rely on the standard heuristics of the Metis tool [36] to partition the graph.

Once a data distribution exists, the next crucial step is to transform the code so that each reference to data is performed on a logical node where the corresponding data exists. A distributed block, or Dblock, is a block of code that accesses data distributed across multiple logical nodes [37]. The Dblock Analyser (Fig. 10) is the tool that resolves these blocks by inserting the necessary hops and transport variables to carry local copies of data. Dblock analysis is necessary for correctness, since any atomic operation must be performed on a logical node that also contains its operands. The key is to perform it in such a way that keeps communication overhead small.

The Dblock analysis consists of three key steps:

1) Dblock selection: Analyze the sequential program to identify the Dblocks to be resolved, choosing the Dblocks at appropriate granularities. A Dblock can be any block of code: a single statement, a loop, an if-then-else construct, etc.
2) Dblock placement: Determine the logical node(s) on which a Dblock will execute. Given a Dblock, we decide where the rendezvous of the locus of computation and the data it requires should happen by following the principle of pivot-computes [38]. This principle states that the computation of a Dblock takes place on the logical node that owns the largest piece of the distributed data. This logical node is called the pivot node.
3) Code augmentation: Modify the original code so that the rendezvous of the locus of computation and the data it requires occurs for all Dblocks. This step requires inserting hop statements and transport variables to carry copies of portions of local data between logical nodes.

The choice of granularity of the Dblock is crucial. For example, with a nested loop we have the choice of resolving the Dblock at one of three different levels: an individual statement, the inner loop, or the outer loop. Choosing the smallest level of granularity results in frequent small messages. Choosing the largest level of granularity requires moving large chunks of data among machines. Currently, we leave the choice of granularity up to the programmer but an important challenge for future research is developing heuristics for selecting the best level at which a Dblock should be resolved.

As indicated in Section III C, the fundamental notion of parallelism in the NavP view is that of the mobile pipeline. This task is performed by the tool Pipeline Builder (Fig. 10); the high-level steps are illustrated in Fig. 11. First, the sequential code Fig. 11(a) is converted to DSC (Fig. 11(b)), as

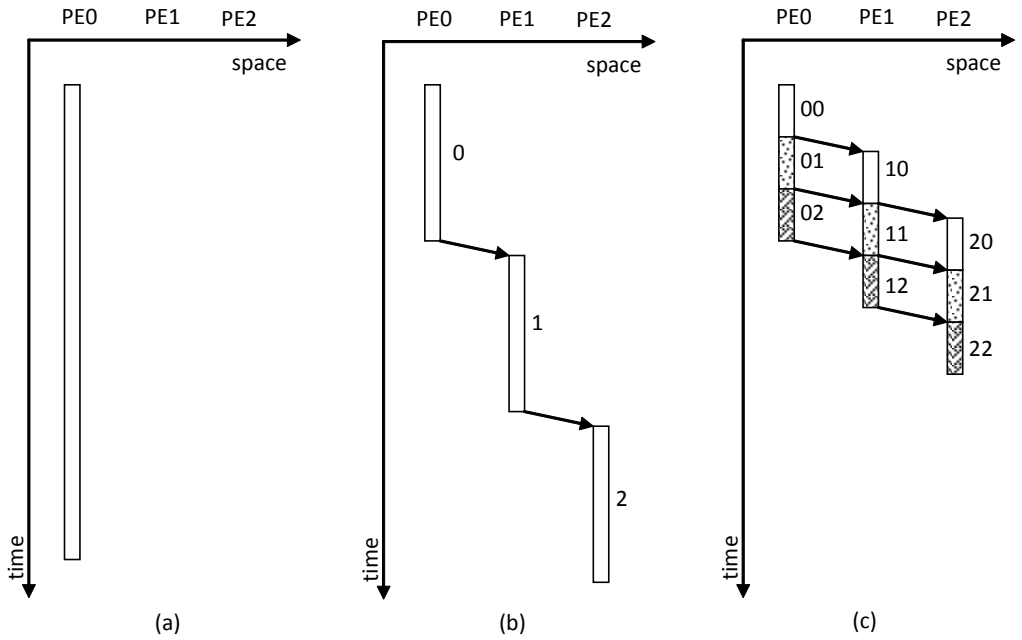

Fig. 11 Pipeline builder. (a) sequential thread. (b) DSC. (c) pipelined threads

described in the section III B. Next, the single DSC computation thread is cut into multiple shorter threads, and these shorter DSC threads are then composed to form a mobile pipeline (Fig. 11(c)). Each of the threads is scheduled to run as early in time as possible, subject to the constraint that all dependences must be respected. In this example, we assumed that the portion of the computation running on PE1 (labeled 1) depends only on some initial portion of the computation on PE0 and thus can start as soon as this portion (labeled 00) has completed. Similarly, the portion 11 can start as soon as 01 has completed, and so forth. These partially overlapping threads spread the parallel computation as they hop through the network, and they continue to maintain low cost of communication as before.

The next step in the incremental parallelization process (Fig. XX) is to compile the parallel code produced by the Pipeline Builder into native code executable on the underlying network cluster. The Performance Profiler then closes the feedback loop by generating important performance statistics such as the number and frequency of hops, the sizes of the thread-carried variables, and level of parallelism/load balancing across the PEs. This data can then be used by the programmer to incrementally improve the implementation. For example, if a particular portion of the code causes frequent hops to occur, this indicates that the mapping of the data should be modified (e.g., by increasing the number of the cyclic data blocks) or that a coarser level Dblock resolution should be used. This incremental process of refinement can be repeated until a program with the desired performance has been derived.

REFERENCES

[1] L. Bic and M. Dillencourt, Mobile Agents, DSM, Coordination, and Self-Migrating Threads: A Common Framework, Proc. 7th WSEAS Int'l Conf. on Data Networks, Communication, Computers (DNCOCO'08), Bucharest, Romania, Nov. 2008

[2] R. S. Gray, D. Kotz, G. Cybenko, and D. Rus, Mobile Agents: Motivations and State-of-the-Art Systems, Technical Report TR2000-365, Dartmouth College, Hanover, New Hampshire, April 2000.

[3] D. Kotz, R. Gray, and D. Rus, Future Directions for Mobile Agent Research, IEEE Distributed Systems Online, 3(8) (2002).

[4] D. Shiao, Mobile Agents: A New Model of Intelligent Distributed Computing, IBM Developer Works, China. 2004.

[5] Z. Minchev and D. Dimitrov, Intuitionistic Fuzzy Concept for Navigation of Mobile Agents in Unknown Environment, 9th WSEAS Int. Conf. on Fuzzy Systems (FS'08), Sofia, Bulgaria, May, 2008

[6] The Telescript Reference Manual, Tech Report, General Magic Inc., Mountain View, CA 940404, June 1996.

[7] G. Cabri, L. Ferrari, L. Leonardi, R. Quitadamo, Strong Agent Mobility for Aglets based on the IBM JikesRVM, Tech Report, Universita di Modena e Reggio Emilia, 2003.

[8] R. S. Gray, Agent Tcl: A flexible and secure mobile-agent system, Proceedings of the Fourth Annual Tcl/Tk Workshop (TCL 96), Monterey, CA, 1996

[9] D. Johansen, R. van Renesse and F. B. Schneider, An Introduction to the TACOMA Distributed System, Technical Report 05-23, Department of Computer Science, University of Tromso, 1995.

[10] G. Di Marzo, M. Muhugusa, C. Tschudin and J. Harms, The Messenger Paradigm and its Impact on Distributed Systems, ICC'95 Workshop on Intelligent Computer Communications, 1995.

[11] K. Li, A Shared Virtual Memory System for Parallel Computing, Proc. of the 1988 Int'l Conf. on Parallel Processing, 1988.

[12] G. Antoniu and L. Bouge. DSM-PM2, A portable implementation platform for multithreaded DSM consistency protocols, Lecture Notes in Computer Science, 2026:55, 2001.

[13] F. Arbab, The IWIM Model for Coordination of Concurrent Activities, in Coordination Languages and Models, Cesena, Italy, 1996.

[14] J-P. Banatre and D. Le Metayer, Programming by Multiset Transformation, Comm. CACM, 36(1):98-111, 1993.
[15] N. Carriero and D. Gelernter, Linda in Context, Comm. CACM, 32(4), 1989.
[16] P. Ciancarini, Distributed Programming with Logic Tuple Spaces, New Generation Computing, 12(3):251-284, 1994.
[17] A. S. Staines, A Fundamental Modeling Concept Approach for Modeling UML Design Patterns, NAUN International Journal of Computers, Issue 3, Vol. 2, 2008
[18] K.-Y. Wong, Y.-M. Choi, and S.-W. Lam, The Design, Implementation and Application of the Software Framework for Distributed Computing, NAUN International Journal of Computers, Issue 3, Vol. 1, 2007
[19] L. Bic, M. Fukuda, and M. Dillencourt, Distributed Computing using Autonomous Objects, IEEE Computer, 29(8), 1996.
[20] M. Fukuda, L. Bic, M. Dillencourt, F. Merchant, MESSENGERS: Distributed Programming Using Mobile Agents, Transaction of the Society for Design and Process Science (SDPS), Vol. 5, No. 4, 2001
[21] M. Fukuda, L. Bic, M. Dillencourt, and F. Merchant, Distributed Coordination with MESSENGERS, Science of Computer Programming, 31(2), 1998
[22] R. Utter, Diaktoros: Full State Migration with Mobile Agents, PhD Thesis, Dept. of Information and Computer Science, University of California, Irvine, 2006.
[23] L. Pan, L. Bic, M. Dillencourt, and M. K. Lai, NavP Versus SPMD: Two Views of Distributed Computation, Int'l Conf. on Parallel and Distributed Computing and Systems (PDCS 2003), Marina del Ray, CA, November 2003
[24] L. Pan, L. Bic, M. Dillencourt, and M. K. Lai, Distributed Sequential Computing, Advances in Computation: Theory and Practice, Vol. 16, Nova Science Publishers, Inc., New York, 2004.
[25] L. Pan, M. K. Lai, M. Dillencourt, and L. Bic, Mobile Pipelines: Parallelizing Left-Looking Algorithms Using Navigational Programming, 12th IEEE Int'l Conf. on High Performance Computing (HiPC-2005), Goa, India, December 2005.
[26] L. Pan, M. K. Lai, K. Noguchi, J. J. Huseynov, L. Bic, and M. B. Dillencourt, Distributed parallel computing using navigational programming." International Journal of Parallel Programming, vol. 32, no. 1, pp. 1-37, 2004.
[27] L. Pan, W. Zhang, A. Asuncion, M. K. Lai, M.. Dillencourt, and L. Bic, Incremental Parallelization Using Navigational Programming: A Case Study, International Conference on Parallel Processing (ICPP-2005), Oslo, Norway, June 2005.
[28] Y. C. Hu, H. Lu, A. L. Cox, and W. Zwaenepoel, OpenMP for networks of SMPs, in Proceedings IPPS/SPDP. IEEE Computer Society Press, 1999, pp. 302-310.
[29] J. Garcia, E. Ayguade, and J. Labarta, A framework for integrating data alignment, distribution, and redistribution in distributed memory multiprocessors, IEEE Trans. Parallel Distrib. Syst., vol. 12, no. 4, pp. 416-431, 2001.
[30] M. Gupta and P. Banerjee, Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers, IEEE Trans. Parallel Distrib. Syst., vol. 3, no. 2, pp. 179-193, 1992.
[31] K. Kennedy and U. Kremer, Automatic data layout for High Performance Fortran, in Proceedings of the 1995 ACM/IEEE supercomputing conference. ACM Press and IEEE Computer Society Press, 1995.
[32] P. Lee and Z. M. Kedem, Automatic data and computation decomposition on distributed memory parallel computers, ACM Transactions on Programming Languages and Systems, vol. 24, no. 1, pp. 1-50, Jan. 2002.
[33] A. Navarro, E. Zapata, and D. Padua, Compiler techniques for the distribution of data and computation, IEEE Trans. Parallel Distrib. Syst., vol. 14, no. 6, pp. 545-562, 2003.
[34] J. Li and M. C. Chen, Index domain alignment: Minimizing cost of cross-referencing between distributed arrays, in Third Symposium on the Frontiers of Massively Parallel Computation, College Park, Md., Oct. 1990, pp. 424-433.
[35] L. Pan, J. Xue, M. B. Dillencourt, and L. F. Bic, Toward automatic data distribution for migrating computations, Int'l Conf. on Parallel Processing (ICPP 07), Xian, China, Sept. 2007
[36] G. Karypis and V. Kumar, METIS, unstructured graph partitioning and sparse matrix ordering system. Version 2.0, University of Minnesota, Department of Computer Science, Minneapolis, Minn., Tech. Rep., Aug. 1995.
[37] L. Pan, L. F. Bic, M. B. Dillencourt, and M. K. Lai, Mobile agents - the right vehicle for distributed sequential computing, in Proceedings, 9th International Conference on High Performance Computing (HiPC 2002), Lecture Notes in Computer Science, S. Sahni, V. K. Prasanna, and U. Shukla, Eds., vol. 2552.Springer-Verlag, Dec. 2002, pp. 575-584.
[38] L. Pan, W. Zhang, A. Asuncion, M. K. Lai, M.. Dillencourt, L. Bic, and L. Yang, Toward Incremental Parallelization, IEICE Trans. Inf. & Syst., Vol. E89-D, No. 2, pp. 390-398, Feb. 2006.

Lubomir Bic received an M.S. Degree in computer science from the Technical University Darmstadt, Germany in 1976 and a Ph.D. in information and computer science from the University of California, Irvine, in 1979.
In 1979-80 he worked as a researcher at the Siemens Corporation in Munich, Germany. From 1980 until 2003 he was a faculty member in the Department of Information and Computer Science at the University of California, Irvine. Since then he has been a Professor and Chair of the Department of Computer Science at the University of California, Irvine. He is the author of 6 books and well over 100 publications in scientific journals and conference proceedings. His current research interests are in the areas of parallel and distributed computing.

Michael B. Dillencourt received an M.A. Degree in Mathematics from the University of Wisconsin in 1975, an M.S. Degree in Computer Sciences from the University of Wisconsin in 1976, and a Ph.D. in Computer Science from the University of Maryland in 1988.
From 1978 to 1988 he worked as a software engineer in private industry. Since 1989 he has been a faculty member in Information and Computer Science at the University of California, Irvine. He is the author of over 70 publications in academic journals and conference proceedings. His current research interests are in the areas of distributed computing and graph algorithms.

.