Computation Migration: A Survey

Sushnanth Rai

Department of Computer Science

University of Texas, Arlington

Rai@cse.uta.edu

Abstract

We introduce the concept of computation migration and its advantages over other programming model. Various granularities of computation migration are presented with case studies. We also present various formal models required to define the semantic characteristics of a distributed system based on computation migration.

1 Introduction

Migration as used in this paper is the movement of data and/or code from one location to another in a distributed environment. Unfortunately, the term-distributed system has been applied to a wide range of computing systems loosely coupled, closely coupled, tightly coupled, array processors, dataflow, neural nets, etc. Irrespective of granularity of the distributed systems the ultimate aim of migration is efficient program execution with optimal use of resources and tolerable performance.

On a distributed system when a computation attempts to access data from another processor, communication has to be performed to satisfy the reference. For example one of the important factors for efficient program execution on a distributed memory parallel system is the locality of data accesses. If there are many non-local memory accesses it is unlikely that the program would exhibit good speedup. Two mechanisms for accessing remote data are computation migration and data migration, which may be viewed as duals. Computation migration takes the advantage of spatial locality of the objects while data migration takes advantage of the temporal locality of the data objects.

Data migration[1,2] involves moving remote data to the host/processor where a request is made. For example in case of Distributed File System when a client requests access to file that is residing on the server the file is moved to the client's host where it gets cached. Infact more number of file blocks are moved than what is requested and hence subsequent requests are satisfied locally on client's host. Data migration can perform poorly when the size of the data is large. In addition it performs poorly for write shared data because of the communication involved in maintaining consistency: when there are many writes to a replicated datum, it is

expensive to ensure consistency. Some of the advantages are that data migration can improve the locality of accesses since after the data is fetched subsequent accesses are local and this model is good when there is high volume of read shared data.

Computation migration involves moving a thread of computation to host/processor where the data is located. It provides a flexible framework for designing distributed systems where the desired nonlocal computations need not be known in advance at the execution site. But spatial locality of data is not the only reason for doing computation migration. Computation migration can also be used for load balancing and fault recovery. Computation migration is also referred to as mobile computation or mobile agents where code along with its execution state travels on a heterogeneous network until it achieves its goal. This does not include cases where the code is loaded from the shared disk or from the web (e.g. Java applet). Some of the advantages of this model include:

- *Efficiency.* If repeated interactions with the remote site are required, it can be more effective to send the code to remote site and make it interact locally. This reduces the inter-machine communication cost thus using the network bandwidth efficiently even with high latency. Also it is possible to do load balancing in this model by dynamically assessing the system load and redistributing the work during their lifetime. This helps in achieving better throughput with efficient use of host resources.

- *Storage.* Storage requirements are reduced by not having to store a copy of the program at all sites. In this model loading is done on demand.

- *Fault Recovery.* Unit of execution (which includes code and execution state) can be moved during gradual degradation of performance of the system. If the state of the program can be stored persistently it also possible to move the program even after it crashes.

- *Flexibility.* With this model it is possible for automatic update of the software, as soon it is available at the software vendor irrespective of the number of clients. It could also be very well used in the web push model instead of the traditional pull model.

- *Simplified Programming.* Distributed programming can be simplified by implementing the migration constructs into the programming languages. This provides maximum portability and high transparency to the programmer since explicit distribution of the application into client and server pieces is no longer required. It also allows the programmer to write the programs in a shared-memory style.

The disadvantages of computation migration are that the cost of using it depends on the amount of computation state that must be moved. If the amount of state is large then migration might be fairly expensive. Also when computation migration is used satisfying locality of data and if the data is read shared then data migration might outperform computation migration. Also in heterogeneous distributed environment the differences in the architectures would make computation migration a difficult task.

Several other mechanisms for accessing remote data are widely used in distributed systems. Remote Procedure Call (RPC)[3] is used both in distributed and parallel systems for executing the procedure call on remote data. RPCs are similar to local procedure calls except the procedure

is now located at the remote location and communication details are taken care by the compiler generated stubs and is made transparent to the programmer. Message Passing Interface (MPI) a standard for parallel and distributed programming uses explicit message passing scheme. All of these mechanisms require the programmer to explicitly split the application into different parts, depending on the application domain, and place them at different processors/hosts before the execution. Even though these mechanisms hides communication details it still would expose the details of distribution.

Figure 1 compares RPC/messaging passing, data migration and computation migration models with respect to communication cost required to access the data on other hosts. RPC makes n consecutive accesses to each of m data items on processors. Both data and computation migration can reduce the communication overhead by making repeated accesses to the local data. As seen data migration involves communication overhead to maintain cache-coherency, which is not involved in computation migration. Hence the advantages listed above with the recent trends

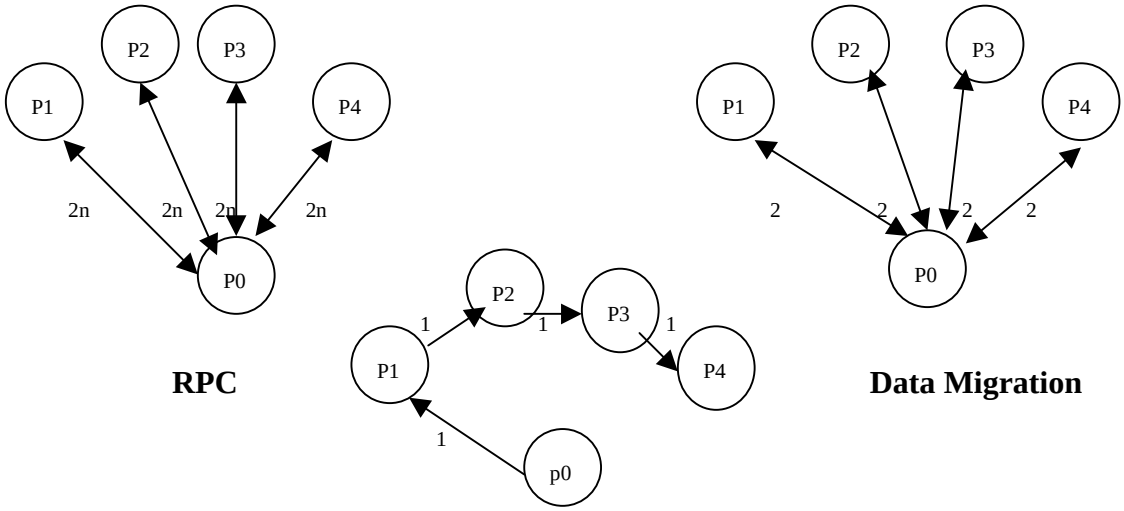

Figure 1: Communication cost in different mechanisms

in distributed computing motivates us to delve into computation migration in more detail. The following sections describe the techniques used in computation migration and also some mechanisms to overcome some of the disadvantages. Section 2 describes the details of how the computation migration is achieved. We will use two case studies to describe the details. Section 3 introduces various formal models which aids in implementing computation migration in programming language which would take care of all the advantages that has been listed above. Section 4 concludes describing the scope for future work.

2 Computation Migration

There are three questions that needs to be answered regarding computation migration - a) when

should the migration take place b) what needs to be migrated c) how should the migration take place i.e. what is the mechanism involved in migration. The answer to the first question depends on the context in which the migration is done. The advantages listed in the previous section imply some of the circumstances under which migration is done. In this section we will try to answer the remaining two questions.

The answer to what needs to be migrated depends on the granularity of migration. The migration can be done at the process level, thread level and environment level. In process level migration operating system process is migrated from one host to another and the operating system kernel is directly involved during the migration. In thread level migration threads are migrated from one node to another. Threads can be at kernel space, in which case the kernel would be involved during migration, or user space (library or programming language threads). Environment migration in one in which a group of processes/threads which represents users active environment at a given instance would be migrated. Hence environment migration subsumes process and thread migration.

2.1 Process Migration

Process Migration is one in which a process can be moved during its execution, and continue on another processor with continuos access to all its resources. This requires operating system to be directly involved in the migration and also provide facilities to aid migration. Process migration is normally an involuntary operation that may be initiated without the knowledge of the running process or any process interacting with it. All processes in the system would continue execution with no apparent changes in their communication or computation. DEMOS/MP[4] is one such operating system, which supports process migration.

The state of a process in a distributed system consists of computation state and communication state. As in conventional context switching the computational state involves all the necessary information required to execute the process when it resumes its activity except in case of migration the process under consideration is switched to another host and all the computation state should to transferred to the destination host. The communication state is something unique to process migration. In a distributed system it is possible that there are other processes communicating with the process that is to be migrated and these links should not invalid after the migration. Hence these links needs to be transferred and restored at the destination host. The difficulty in process migration is that the state of the process as we have defined would be distributed among a number of tables in the system making it hard to extract the information from the source processor and creating it in the destination processor. In some systems this might be impossible if the Communication State of the process is tightly coupled with the machine identifier.

Communication links of a process required for interprocess communication can be implemented by using a link table maintained by the kernel. A link table contains pointers to the communication endpoints of the peer process. To migrate a process, the link tables of those processes that have a link to the migrating process must be updated so that communication links remain intact after the migration. Typical process migration strategy executes the following steps

1. Remove the process from execution: The process is marked as "in migration". From this point onwards all the messages directed to this process would be buffered and forwarded to

destination process after it starts executing.

2. Ask the destination kernel to move process: The source system send a message to the kernel of the destination system containing information about the size and location of the process's resident state, swappable state and code.

3. Allocate a process state on the destination state: An empty process state is created on the destination processor.

4. Transfer the process state from the source processor into the empty destination processor.

5. Transfer the program: This included code, data and stack.

6. Forward the buffered messages to the destination process.

7. Clean-up the process state: On the source processor all the resources by the now migrated process would be reclaimed. A small foot-print of the process is left on the source process which acts as proxy process which would forwards the requests to the destination process

8. Restart the process at the destination processor.

Although all the advantages stated in section-1 cannot be achieved by process migration it helps in dynamic load balancing and fault recovery. Process migration provides the ability to stop a process, transport its state to another processor, and restart the process, transparently. If the information necessary to transport a process is saved in stable storage, it may be possible to migrate a process from a processor that has crashed, to a working one. Some of disadvantages in process migration are that it assumes a homogenous environment. Since the migration is done at the kernel level this strategy requires all the nodes in the distributed system to be running the same operating system. Also it does not consider the issues related to architectural differences in various processors participating in a distributed environment.

2.2 Thread Migration

In thread migration the smallest entity that can be migrated is a thread. A thread is a small set of instructions, which is part of a larger process and does some partial computation. Several threads within a process may cooperate and execute concurrently to achieve the desired task. A thread could be controlled by the operating system kernel or managed in the user space in terms of libraries or programming language constructs. Whatever the implementation may be the concept remains the same across all spaces i.e. threads provide a programming mechanism to partition the code into several independent units of execution which can run in parallel thus improving the efficiency of execution. Also as threads share the global state of the process with other threads they provide a shared-memory model of programming. A process can be considered as a single threaded application. Hence threads looks like an attractive and less expensive model for migration.

A concept normally used in thread migration is continuation. Continuations[6] are programming language semantics that models rest of the program. Formally a continuation is a function from whatever the rest of the program expects to be passed as an intermediate result-and this depends on what the rest of the program follows- to the final answer of the program. In continuation

semantics the denotations of the constructs in a language depend on the rest of the program following them. The idea here is that each construct decides for itself where to pass its result. This is useful for handling non-local jumps and state saving. For example consider the following fragment of LISP code which uses continuation:

$$(+ (f\ 1)\ (g\ 2))$$

$$\Rightarrow (k\ (+\ \ (f\ 1)\ (g\ 2)))$$

$$\Rightarrow \quad (f\ \ 1\ (\lambda\ (v1)$$

$$(g\ 2\ (\lambda\ (v2)$$

$$(+\ v1\ v2)))$$

While transforming a direct semantics program to the one that uses continuation an extra parameter is added to every function and this parameter represents the continuation. Thus in general the continuation denotation of a construct is a function of a continuation and state which yields the final answer being obtained by passing some intermediate results to some continuation i.e. to the rest of the program. Hence at any point in time continuation captures the state of the thread and if the continuation can be captured as a data-structure it is possible to transport this to another host restart the execution on that host. Continuations are implicitly present in all the programming languages and some languages like Scheme provides constructs to access the continuations as data-structures. Also this is very attractive approach since thread is a small set of instructions and the cost involved in capturing and transporting the continuation is small. Another advantage of this approach is that migration can be done at the language level without involving the kernel at all. This provides maximum flexibility and transparency since adding some extra constructs to the language is all that is necessary to augment the existing language to have distributed scope.

It is possible to define the granularity of thread migration depending on the application for which it is used. When migration has to be done for data referencing it could be useful to have partial thread migration i.e. migrating one or more activation frames at the top of the stack. The advantage is that amount of state to be moved is very small and would be more effective than data migration or RPC. This is done in an augmented version of Prelude[7] to support computation migration for non-local data referencing. The language is extended using a set of annotations that allows the programmer to migrate a procedure. The language runtime uses continuations for migration but this is transparent to the programmer. However the implementation of Prelude does not support migration for other purposes like load balancing, service oriented distributed computing etc.

If the migration is done for load balancing or remote processing it may be necessary to move the entire thread. Cilk[8] is multithreaded language developed on the top of C for handling load balancing in a parallel environment. A Cilk program consists of a collection of Cilk procedures, each of which is broken into a sequence of threads, which forms a directed acyclic graph. Each thread in Clik is a nonblocking C function, which can run into completion without waiting or suspending once it has been invoked. Cilk supports two types of data-structures called as

closures and continuations. As shown in figure 2 closure consists of a pointer to the C function, a slot for each of its arguments and a counter indicating the number of missing arguments. A closure is ready if it has all of its arguments and it is waiting if some arguments are missing. Continuation is essentially a global reference to an empty argument slot of a closure implemented as a data structure containing a pointer to the closure and an offset that designates one of the closure's argument slots. When some thread fills up this argument it passes the continuation explicitly to the thread that is on wait closure. The idea here is that closures and continuations, which represent the state of the thread, are used for communication among different threads and also for migrating the thread between processors.

Cilk uses an approach called as work-stealing for the migration of the thread. Initially all the processors start running a scheduler and all the threads would be assigned to a single processor. The idle processors would then steal the work (work that is in the ready closure state) from the other processors which has enough work. The stealing process involves transferring the closure from one processor to another, which is same as transferring the state of the thread.

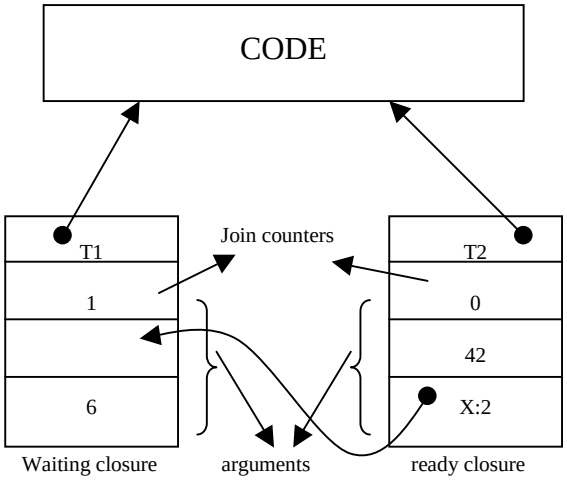

Figure 2. Closure data structure in Cilk

Cilk threads are designed for load balancing in a parallel system with computationally intensive tasks. The reason we have considered here for our discussion is to understand the implementation of continuations and its usage in migration. Its explicit continuation passing semantic model of programming requires extra effort from the programmer. Nevertheless it provides a programming model for migration which can be extended to many other application domains.

2.3 Environment Migration

Environment migration is a general idea and is not limited to code migration. The entity that can be migrated includes threads, processes, objects, devices etc. This concept is aimed at World-Wide Web. The geographic distribution of the web naturally calls for mobility of computation, as a way of flexibly managing latency and bandwidth. The main difficulty in migration over the web is the handling of administrative domains. Firewalls partition the Internet into administrative domains that are isolated from each other except for rigidly controlled pathways. System

administrators enforce policies about what can move through firewalls and how. It can be inferred that the earlier two models of migration did not address this issue and the implementation of those models assumed a trusted environment.

Migration on the Web requires more than the traditional notion of authorization to run or to access information in certain domains: it involves authentication and authorization to enter and exit domains. It would not be realistic to assume that the migrating entity can move from any point A to any point B on the Internet. It should take permission to exit its administrative domain and take permission to enter some other domain. Multiple levels of authorization may be required since there are mutilple levels: local computer, local network, wide-area network etc. through the information has to pass through and at each level authorization has to be obtained to move further. Hence environment migration model adopts a strategy where entities of migration are hierarchically structured and at any instance a part of this hierarchy can be migrated as a whole instead of individual threads or processes. The goal is to make migration scale-up to widely distributed, intermittently connected and well-administered computational environments.

Environment migration model is followed by Ambient[9]. An ambient is a bounded place where the computation happens. One of important characteristic of ambient is notion of boundary, which determines what should be moved. Boundary determines what is inside and outside the ambient. Examples of ambients are a web page (bounded by a file), a virtual address space (bounded by an addressing range) and a laptop (bounded by its physical case and data ports). Boundaries help is determining some flexible-addressing scheme that can denote entities across the boundary; examples include symbolic links, Uniform Resource Identifiers etc. Ambient may be nested and they can be moved as whole. An ambient has a name, which is used to control access to the ambient i.e. they are associated with capabilities, which determines the authorization.

Environment migration provides ultimate flexibility in terms of migration and would also be the ultimate goal of distributed processing. Ambient is just a specification for such a framework. An implementation of such a model would be able to satisfy all the advantages that we had stated in section 1. However it is not clear how can one define the granularity of the ambient in a programming language or whether it requires any additional facilities to achieve migration. The success of this model largely depends on how well it could be integrated into existing programming paradigms.

One of the issues related to migration is handling differences in computer architecture. This is because in migration, live code is moved from one host to another. One extreme is to assume a homogenous environment. DEMOS/MP and Cilk assume such an environment. Other extreme would be to convert the data and code to intermediate representation before the migration and converting it to destination processor representation after the migration. This is done is in [10].

3. Formal Models

Formal Models in general provide precise mathematical descriptions of the systems. There are three advantages of using formal techniques:

- Providing precise machine independent concepts
- Providing unambiguous specification techniques
- Providing a rigorous theory to support reliable reasoning

As we have seen already seen there are various computation migration models, which has been developed for specific tasks. These models have also influenced their implementation. However these models have received informal treatments. To use computation migration in the main stream distributed computing it is important to develop precise semantic definition for migration, which would provide important insight and highlight the weak parts of its definition. This would also help in developing formal statements for the features of such a system and provide proof for advantages it claims, which we have informally listed in section-1. In this section we shall examine some of the formal models which have been proposed in this area.

3.1 Pi-Calculus

Pi-Calculus[11] is formal model used to represent concurrent computation. The reason we are considering it here is that all the other models which defines migration is in someway or the other related to Pi-Calculus. In pi-calculus every expression denotes a process – a free-standing computational activity, running in parallel with other processes and possibly containing many independent subprocesses. Two processes can interact by exchanging a message on a channel. Indeed communication along channels is the sole means of computation, just as functions in lambda calculus. The only thing that can be observed about a process's behavior is its ability to send and receive messages. Central to pi-calculus is the notion of naming. Naming also provides independence; one naturally assumes that the namer and the named are co-existing entities (concurrent) entities. Also if naming is involved in communication (using channel) and in locating and modifying data then it would be possible to treat data-access and communication as the same thing which in terms of pi-calculus would make data as a special kind of process.

Syntax and operational semantics of Pi-calculus

Pi-calculus provides a small set of primitives for building concurrent programs:

P,Q,R ::= 0 inert process

X(y).P input prefix

-xy.P output prefix

P|Q parallel composition

(vx) P restriction

!P replication

The simplest pi-calculus expression is the inert process 0 i.e. a process with no behavior at all. If P is some process then the expression x(y).P denotes a process that waits to read a value y from channel x and then, having received it, behaves like P. Similarly –xy.P denotes a process that first waits to send the value y along the channel x and then, after y has been accepted by some

input process, behaves like P. P|Q denotes a process composed of two subprocesses, P and Q running in parallel. (vx)P indicates the creation of a fresh channel in P, which is unique within the process P and the messages sent and received, by P on x will never be mixed with messages sent or received on any other channel created elsewhere. !P indicates replicated process and stands for infinite number of copies of P, all running in parallel.

As mentioned earlier pi-calculus is used for modeling concurrency via communication over the channels. Processes can be moved or passed along the channels and channels can be moved over other channels. In a distributed setting channels can span across a network and atomic interaction between sender and receiver is difficult to achieve since it is important to take care local and remote failures. Although there is primitive idea of migration, there is no clear indication of the granularity of mobility. Nevertheless it provides a framework based on while calculus for migration can be built. PICT[13] is an experimental language built using this calculus

3.2 CHemical Abstract Machine(CHAM)

Just a Turing machine models theories of recursive functions; CHAM[14] is a semantic model for concurrent computations. It can be considered a computation model of pi-calculus. The system in CHAM is modeled like a chemical solution in which floating molecules can interact with each other according to reaction rules; a magical mechanism stirs the solution, allowing for possible contacts between molecules. The solution transformation is inherently parallel. CHAM provides molecule syntax and a set of transformation rules that specify how to produce new molecules from old ones. Membranes of molecules provide boundaries of computation. Molecules travel to reaction site as indicated by the rules where they find a matching molecule to react. Hence the communication site is centralized and as new molecules gets generated as a result of the reaction the communication site would become a bottleneck.

3.3 Distributed Join Calculus

The limitation presented in CHAM is resolved in Join calculus[15] by allowing dynamic creation of sites and restricting reaction patterns. As in pi-calculus names are central to Join calculus. Processes (that return no values) and expression (return values) are basic syntactic classes. Processes communicate by sending messages on channels or port names. Messages carried by channels are made of zero or more values and channels are values themselves. In contrast to other calculi, channels and processes that listen on them are defined by a single language construct. This feature allows us to consider channels as functions. Join calculus is more closer to programming language than pi-calculus. The basic primitives of Join calculus are same as that of pi-calculus. Join pattern defines a set of channels simultaneously sending/receiving messages, which would trigger a process to execute. Join patterns are used in synchronization.

Distributed Join calculus[16] adds the notion of location names to join calculus to support migration. Locations are first class values that statically identify a location. Like port names, they can be created locally, sent and received in messages, and they obey lexical scooping discipline. In the join-calculus, the execution of a process or an expression does not usually depend on its localization. Indeed, it is equivalent to run processes P and Q on two different machines, or to run process P | Q on a single machine. In particular, the scope for defined names and values is independent of localization: when a port name is known of some process, it can be used to form messages (either as destination or as content) without knowing whether it is locally-

or remotely-defined. Locality matters in some circumstances: side-effects such as printing values on the local terminal depend on the current machine; besides, efficiency can be dramatically affected because message-sending over the network takes much longer than local calls. For all these reasons, locality is explicitly controlled within the Distributed join-calculus; it can be adjusted using migration. In contrast, resources such as definitions and processes are never silently relocated by the system.

While processes and definitions are statically attached to their location, locations can move from one place to another. A process inside of the location triggers such migrations. As a result of the migration, the moving location becomes a sublocation of its destination location.

```
# loc mobile
#   init
#     let here = ns.lookup("here") in
#     go(here);
#     let sqr = ns.lookup("square") in
#     let sum(s,n) =
#       reply (if n = 0 then s else sum(s+sqr(n),n-1)) in
#     let result = sum(0,5) in
#     print_string("q: sum(5)= "^ml.string_of_int(s)^"\n");
#   end
```

The go (here) primitive migrates the whole mobile location on machine here, as a sub-location of location here, then it returns. Afterwards, the whole computation (calls to the name server, to sqr and to sum) is local to here. There are only three messages exchanged between the two machines: one for the lookup (here) request, one for the answer, and one for the migration. The Object-Caml[17] language provides extension supporting Join and Distributed join calculus.

3.4 Mobile Ambient Calculus

The ambient that we introduced in section 2.3 is based on ambient calculus[9]. This is aimed at the mobile computation over the Internet with security being the primary concern. The basic set of primitives in the calculus is same as that of pi-calculus that we have discussed. The main difference with respect to pi-calculus is that names are used to name ambients instead of channels. As mentioned earlier ambient is a bounded place of computation similar to molecules of CHAM, which are bounded by membranes. An ambient can be nested within other ambients, can be hierarchically structured and can be moved as a whole. If we want to move a laptop to a different network all the address spaces and file systems within it move accordingly. The name of the ambient is also used to gain control access to the ambient. A name is something that can be created and passed around and from which access capabilities can be extracted.

In addition to the basic primitives, n[P] is an extra primitive in the syntax of ambient calculus where n is the name of the ambient, and P is the process running inside the ambient. In n[P], it is understood that P is actively running, and that P can be parallel composition of several processes. P can also be running when the surrounding ambient is moving. In contrast to Distributed Join

calculus the notion of location is explicitly built into calculus. Changing the hierarchical structure of the ambient indicates migration and these operations are restricted by the capabilities of the ambients.

Figure 3 shows relationship among various calculi that we have discussed so far.

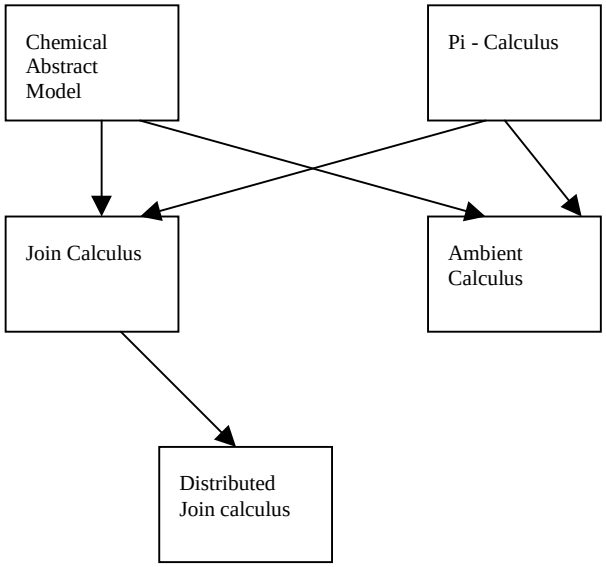

Figure 3: Development of Calculus for concurrent and distributed systems

4 Conclusion

In this paper we have discussed the current state of art technologies in the area of migration. Computation migration as such is not a new concept. However the recent trends in computer architecture, like the use of virtual machines, need for distributed computing and its related problems and advancements in programming languages has made computation migration a viable alternative for distributed processing. As we have seen in section 2 there has been variants of computation migration each for a specific task. However these implementations cannot be extended towards implementing service oriented distributed computing environment which requires explicit notion of nodes of computation for accessing distributed resources and also rules for establishing trust relationships between the sender and receiver of the migrated code.

Current model of distributed programming is based on client-server concept. Computation migration provides a new model for distributed programming reducing the programming complexity. For this we require new programming methodologies, programming libraries and programming languages. The formal model that we presented in section 3 is first step towards defining constructs and semantics of these languages. The success of this effort depends on how well it could be integrated into current languages and technologies to so that the existing

framework can be reused and would provide a smooth transition towards the new paradigm.

References

[1] Hsieh, W.C., Wang, P., and Weihl, W.E. Computation Migration: Echancing Locality for Distributed-Memory Paralle Systems. In the proceedings of 4th ACM PPOPP, May 1993:(239-248)

[2] Carlisle, M.C. and Rogers, A. Software Caching and Computation in Olden. In the Journal of Parallel and Distributed Computing 38, 248-255(1996)

[3] Birrel A.D., and Nelson. Implementing Remote Procedure Calls. ACM Transactions on Computer Systems, 2(1):39-59, February 1984

[4] Powell, M.L. and Miller, B.P. Process Migration in DEMOS/MP. In the proceedings of 9th ACM symposium on Operating Systems Principles, Oct 1983: (110-119)

[6] Dybvig, R.K. and Heib, R. Engines from Continuations. Computer Languages, 14(2):109-123 (1989)

[7] Weihl. W., Brewer. E., Colbrook. A., Delarocas, C., Hsieh, W., Joseph, A., Waldspurger, C., and Wang, P. PRELUDE: A System for Portable Parallel Software. Technical Report MIT/LCS/TR-519, MIT LCS, Oct 1991.

[8] Blumofe, R.D., Joerg, C.F., Kuszamaul, B.C., Leiserson, C.E., Randall, K.H., and Zhou, Yuli. Cilk: A Efficient Multithreaded Runtime System. In the proceedings of 5th ACM PPOPP, May 1995

[9] Cardelli, L., and Gordon, A.D. Mobile Ambients. In M. Nivat, editor, Foundations of Software Science and Computational Structures, number 1378 in LNCE,140-155. Springer-Verlag, 1998

[10] Steensgaard, B., and Jul, E. Object and Native Code Thread Mobility Among Heterogenous Computers. In the proceedings of ACM SIGOPS: (68-78), 1995

[11] Milner, R. The Polyadic Pi-Calculus: a tutorial. In Logic and Algebra of Specification. Springer Verlag, 1993

[13] Pierce B. C., Remy, D., and Turner, D. N. A typed higher-order programming language based on the pi-calculus. In the Workshop on Type Theory and its Application to Computer Systems, Kyoto University, July 1993

[14] Berry, G., and Boudol, G. The chemical abstract machine. Theoretical Computer Science, 96:217-248, 1992.

[15] Fournet, C and Gonthier, G. The reflexive chemical abstract machine and the join calculus. In POPL'96[2], pages 372-385.

[16] Fournet, C., Gonthier, G., Levy, J.J., Maranget, L., and Remy, D. A calculus of mobile agents. In CONCUR96, pages 406-421, 1996.

[17] Leroy, X. Objective Caml. Available at http://pauillac.inria.fr/ocaml