

ABSTRACT OF THE DISSERTATION

Navigational Programming

by

Lei Pan

Doctor of Philosophy in Information and Computer Sciences

University of California, Irvine, 2005

Professor Lubomir F. Bic, Co-Chair

Professor Michael B. Dillencourt, Co-Chair

This research develops the concept and methodology of navigational programming for distributed parallel computing. *Navigational programming (NavP)* is the programming of self-migrating computations in distributed memory. Shared variable programming is made possible through the use of computation mobility. The building blocks of NavP include an underlying system, called MESSENGERS, that consists of a compiler and a daemon system, and a methodology that provides the following four steps for its programmers:

- (1) *Data distribution.* An affinity graph is generated from instrumenting the sequential algorithm, and the graph is partitioned in order to get a data mapping to guide the programmer in data distribution. The graph partition uses heuristics to minimize the overall edge cuts in the graph; this corresponds to a heuristical minimization of the overall communication cost for a distributed

implementation.

- (2) *Distributed sequential computing (DSC)*. Navigational statements (e.g., `hop()`) are inserted into the sequential code, and the resulting DSC thread navigates to large-sized data carrying small-sized data for computation to happen. This step follows the principle of *pivot-computes*, which requires that a sub-computation should happen at the computer node that hosts the large-sized data. This code transformation preserves *algorithmic integrity*, which says that the codes before and after the transformation are structurally the same.
- (3) *Distributed parallel computing (DPC)*. The DSC computation thread is transformed into multiple “shorter” migrating threads, and these threads are composed into DPC programs using pipelining or phase-shifting techniques. Synchronization statements (e.g., `wait()` and `signal()`) are inserted into the code at this step. This step of code transformation exhibits *composition orthogonality*, which means the code intersection among the composing DSC threads is minimum.
- (4) *Loop back* for feedback and refinement.

The above steps can be applied repeatedly or hierarchically along the temporal or spatial dimensions until a satisfying performance is achieved or a resource constraint is reached.

NavP distinguishes itself from other existing approaches in that it provides a new view to distributed computation: the NavP view. Under the NavP view, a distributed computation is described following the movement of its locus. The NavP view is the same as the Lagrangian view in fluid dynamics. In contrast, all existing approaches, such as message passing (MP) or distributed shared memory (DSM), use

the SPMD (single program multiple data) view, in which computations are described at stationary locations. This is the same as the Eulerian view in fluid dynamics. The problem with the SPMD description is that the quantity of interest – distributed computations along with the small-sized data they need – “floats around” in the network and hence is fundamentally Lagrangian. The use of the SPMD view could result in one of the two situations: (1) The programmers are forced to dramatically restructure the sequential code to mimic the NavP view in order to obtain efficient parallel programs, as in MP. The `Send()` and `Recv()` statements in MP code are temporally `goto` statements. (2) Communication becomes unnecessarily expensive as large-sized data is moved to meet with the computation loci, as in DSM.

NavP can be used to exploit both data and pipeline parallelism, the latter of which is often considered hard to parallelize. The advantages of navigational programming fall into the two categories of ease of use and high performance. Some of them are:

- (1) NavP preserves the sequential code structure and hence the code transformations are easy and highly mechanical, and the resulting parallel code costs less to maintain. In contrast, one needs to do reverse engineering to understand what an MP parallel code actually does. NavP puts an end to the use of `goto` statements in distributed programs. The programmability of NavP is comparable to that of DSM. Also, the optimization done to the sequential code (e.g., for better cache performance) is preserved in the NavP code.
- (2) NavP is amenable to incremental parallelization. The intermediate implementations are also elegant and fast. The NavP incremental parallelization is conducive to different ways of thinking that lead to better implementations.
- (3) The methodology of NavP leads to parallel programs that perform as good as or

better than the corresponding MPI programs. NavP provides multi-threading. And the code that handles such local behaviors as efficient multi-threaded task scheduling is factored out from the application program and put into the daemon system. Hence NavP is a uniform way of efficiently programming a mixture of shared- and distributed-memory architectures.

- (4) NavP provides one-sided communication. Multi-threading and one-sided communication are inline with the current trend in distributed computing (e.g, MPI-2 and a hybrid use of MPI and OpenMP).
- (5) NavP is good at capturing coarse grained parallelism through the use of mobile pipelines.

For proof of concept, case studies are done with important and nontrivial real-world algorithms. These include: Crout factorization, Cholesky factorization, Jacobi iteration, Gauss-Seidel iteration, matrix multiplication, and sorting. NavP programs exhibit the same or better performance and scalability than our MPI and the available ScaLAPACK implementations.

The future work on NavP includes two distinct but related directions: (1) To make NavP a new way of manual parallel programming and an alternative to MPI. To provide a NavP library consisting of reusable mobile agents with relatively simple behaviors to facilitate the development of more sophisticated NavP-based applications. (2) To build tools that automate the NavP steps based on the manipulations of affinity and dependency graphs.