# A GENERIC ALL-PAIRS CLUSTER-COMPUTING PIPELINE AND ITS APPLICATIONS

A. RADENSKI

*Computer Science Dept. Winston-Salem State University, Winston-Salem, NC 27110, USA*
*E-mail:radenski@computer.org*

B. NORRIS

*Computer Science Dept., University of Illinois at Urbana-Champaign, 1304 W. Springfield Ave., Urbana, Illinois 61801, USA*

W. CHEN

*Computer Science Dept. Winston-Salem State University, Winston-Salem, NC 27110, USA*

In this paper we propose a generic pipeline for *all-pairs computations* on a cluster of workstations. We use this generic pipeline to derive specific cluster algorithms for three different all-pairs problems: *n*-body simulation, bubble sort, and Gaussian elimination. We implement the generic pipeline and its derivatives on a cluster of Intel Pentium II workstations using C and the PVM cluster computing environment. We measure and evaluate the performance of the derived algorithms. The n-body and bubble sort algorithms achieve super-linear speedup for large problems.

## 1 Introduction

An *all-pairs computation* performs the same *operation* on every possible set of two elements chosen from a system of *n* elements [7, 4]. The operation changes a pair of elements independently of the remaining elements and is called for this reason *interaction* between the two elements. Examples of all-pairs computations include *n*-body simulation [4, 3], bubble sort [2], Gaussian elimination [1], and Householder reduction [4]. An all-pair sequential computation over a large number of elements may become prohibitively complex. Interactions between pairs of elements happen in the order specified by a precedence graph [4]; fortunately, some of the interactions between pairs of elements are independent of each other and can, therefore, be executed in parallel.

It is possible to specify an all-pairs computation as a *generic parallel algorithm* that implements process control and communication in a problem-independent manner. Such a generic algorithm can be glued together with domain-specific sequential code in order to derive particular all-pairs parallel computations. This methodology is facilitated by a concurrent message-passing language, *Paradigm/SP*, and by a compiler. *Paradigm/SP* [5, 6] is a high-level object-oriented language that allows the validation of parallel algorithms before they are converted into efficient cluster computing applications.

In Section 2 we propose a generic all-pairs pipeline algorithm for parallel computations on clusters of workstations. In Sections 3 we use this generic algorithm to derive specific parallel algorithms for three different problems: n-body simulation, bubble sort, and Gaussian elimination. In Section 4 we describe implementations of the algorithms on a homogeneous cluster of workstations using PVM, the parallel virtual machine software package. In the same section, we present performance measurements on an Intel Pentium II cluster of workstations. Concluding remarks are presented in Section 5.

## 2 A Generic All-Pairs Pipeline

We assume that an all-pairs computation in a system of *n elements* is defined by the type of the elements and by two sequential methods:

- a method *interact* to make two arbitrary elements exchange information and eventually update their states;
- a method to *integrate* the system of updated elements into a new system.

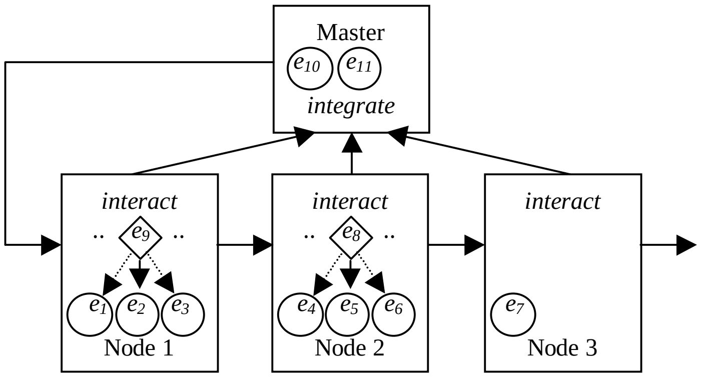

**Figure 1.** An all-pairs pipeline in progress.

An all-pairs computation on a system of n elements can be parallelized by means of a master and several pipelined nodes. The master sends all elements to travel left-to-right through the pipeline. Each node first retains its own subsystem of elements and makes them interact with each other, thus performing a sequential all-pairs computation on its retained elements. After that, the node continues to receive elements from its left neighbor, makes them interact with its own retained elements, and then sends them to its right neighbor. By traveling left-to-right through the pipeline, all elements finally become evenly distributed among the pipelined nodes. Then the master integrates the system of received elements by eventually performing additional problem-specific modifications on the elements. After that, the master may send the whole system through the pipeline again; the whole process is repeated a predefined number of steps. This type of computation can be efficient on a cluster of workstations provided the interaction between pairs of elements is computationally intensive and the individual elements are not very large.

We specify and test this generic all-pairs pipeline using the concurrent message-passing language Paradigm/SP [5, 6]. Procedures interact and integrate, and the element type are the principal parameters of the algorithm, together with the total number of elements n and the number of pipeline nodes p:

```
const n  = ..; {number of elements}          p  = ..;  {number of  nodes}
type element = ..;                           system  =   array[1..n] of element;
procedure interact(var ei, ej: element);  procedure integrate(var s:system);
```

Procedures *interact* and *integrate* are left unspecified in the generic algorithm because they vary significantly from one problem to another. The only assumption made in the generic algorithm is that procedure *interact* can operate and eventually update two individual elements, *ei* and *ej*, while procedure *integrate* can update the state of the whole system, *s*. The generic algorithm allows any desirable domain-specific *element* type without restrictions.

A net *c* of channels capable of transmitting messages of type *element* is declared and opened as shown below:

```
type channel = *(element);                type net = array [0..p] of channel;
var c: net;                               for k := 0 to p do open(c[i]);
```

The master sends all *n* elements of the system through its *left* channel to the leftmost node of the pipeline (see Fig. 1). As shown in Fig. 2, procedure *node* first retains a block of *n/p* elements in a local array, *e*. After that, the node continues to receive transient elements through its *left* channel. The node performs an application-dependent *interaction* between each transient element, *ej*, with every retained element, *e[j]*, eventually changing the states of the interacting elements. After the transient element interacts with all of the node’s retained elements, the node sends it through its *right* channel and receives another transient element through its *left* channel. After handling all transient elements, the node sends, through its *top* channel, all retained elements to the master.

As shown in Fig. 3, procedure *master* repeatedly sends the whole system of elements to the pipeline through its *left* channel, so that each node can retain a block of elements and make these retained elements interact with transient elements. After these interactions, the master receives back all retained elements. Through a bottom net of channels that connect each pipeline node to the master. In order to make this communication more efficient, the master employs a whole net of bottom channels, one for each individual pipeline node. At the end of each cycle, the master performs an application-dependent *integration* of the whole system (see Fig. 3).

```
procedure node(
  steps, first, last: integer;
  left, right, top: channel);
const
  max = n div p;
type
  block = array
    [0..max] of element;
var
  e: block; ej: element;
  i, j: integer;
begin
  repeat
    for i := 0 to last-first do
      begin
        receive(left, e[i]);
        for j := 0 to i-1 do
          interact(e[i], e[j]);
      end;
    for j := last+1 to n do
    begin
      receive(left, ej);
      for i := 0 to last-first do
        interact(ej, e[i]);
      send(right, ej);
    end;
    for i := 0 to last-first do
      send(top, e[i]);
    steps := steps - 1;
  until steps = 0;
end; {node}
```

**Figure 2.** Pipeline node.

```
procedure master(
  steps: integer;  var s: system;
  left: channel;  bottom: net);
begin
  repeat
    sendSystem(left, s);
    receiveSystem(bottom, s);
    integrate(s);
    steps := steps - 1;
  until steps = 0;
end; {master}
```

**Figure 3.** Master node.

```
procedure compute(
    steps: integer;  var s: system);
  var
    c, b: net;
  begin
    openChannels(c, b);
    parallel
      master(steps, s, c[0], b) |
      spawnNodes(steps, c, b);
    end;
  end; {compute}
```

**Figure 4.** All-pairs pipeline.

Finally, the algorithm from Fig. 4 creates the pipeline displayed on Fig. 1 by first opening all channels and then running in parallel one master and *p* pipeline nodes.

Given this generic parallel algorithm, one can derive a parallel algorithm for a particular problem by specifying the particular type of its *elements* and the problem-specific, sequential procedures *interact* and *integrate*. Finally, procedure *compute* is invoked in the master to solve concrete problem instances.

A complete specification of the all-pairs pipeline algorithm in the form of a *Paradigm/SP* generic module can be found in [9].

## 3 Deriving Specific Cluster-Computing Algorithms

We use the generic all-pairs pipeline to derive specific cluster algorithms for three different all-pairs problems: *n*-body simulation, bubble sort, and Gaussian elimination. We achieve this by defining the type of the *elements* for each specific problem, and by defining concrete sequential versions of methods *interact* and *integrate*. The sequential versions of *interact* and *integrate* are linked together with the generic parallel algorithm in order to obtain a specific parallel algorithm. Complete specifications of all derived algorithms can be found in [9].

### *3.1 N-Body Simulation*

We consider a discrete *n*-body simulation problem: compute the positions of *n* bodies in space at equal discrete time intervals, assuming that the bodies interact through gravitational forces only. From the generic all-pairs pipeline, we derive a *parallel n-body simulation* algorithm that is similar to the one presented in [4, Chapter 6]. The generic *element* from the all-pairs pipeline is specialized in the *n*-body simulation algorithm to represent a body with a particular mass *m*, relative distance *r* from the origin, velocity *v*, and cumulative force *f* acting upon the body:

```
element = class m: real; r, v, f: vector; end;
```

At each simulation step, procedure *interact* models force interaction between two particular bodies by (1) calculating the gravitational force between the two bodies at their current positions and (2), by correspondingly updating the cumulative forces acting upon each of the bodies:

```
procedure interact(var ei, ej: body);
var fij: vector;
begin  fij := force(ei, ej); ei.f := sum(ei.f, fij); ej.f := difference(ej.f, fij); end;
```

Procedure *integrate* simulates body moves as a result of body interactions by calculating the velocity and position increments at the end of each time interval.

## *3.2 Bubble Sort*

We consider a standard internal sort problem: given an array of *n* elements, sort them in ascending or descending order. Starting again with the generic all-pairs pipeline, we derive a *parallel bubble sort algorithm*. For the derivation of bubble sort, we introduce a new generic parameter, a relation *less* that is capable of comparing any two generic elements. Then we define *interact* to swap elements *ei* and *ej* if *less(ei, ej)* is *true*:

```
procedure interact(var ei, ej: element);
begin if less(ei, ej) then swap(ei, ej); end;
```

This version of *interact* makes smaller elements move towards the end of the pipeline while large elements are kept closer to the beginning of the pipeline. As a result, the whole system becomes sorted in descending order.

The derived bubble sort algorithm is specialized yet generic version of the all-pairs pipeline because it can be used to sort any type of elements. As an example, we derive an integer bubble sort algorithm by defining an element to consist of a simple integer component:

```
element = class v: integer; end;
function less(ei, ej: element): boolean; begin less := ei.v < ej.v end;
```

## *3.3 Gaussian Elimination*

Finally, we consider a standard numerical problem: find a solution for a system of *n* linear equations with *n* unknowns. Starting again with the generic all-pairs pipeline, we derive a *parallel Gaussian elimination* algorithm for solving such a system. We combine the system's *n* by *n* matrix and the right-hand side vector into a *n* by *n+1* matrix. Then, the original all-pairs *elements* are specified to represent *rows* of the extended matrix:

Given two rows *ei* and *ej*, method *interact* eliminates *xj* from e*i* by multiplying and subtracting *ej* from *ei*. We have enhanced procedure interact with partial *pivoting* in order to reduce the numerical instability of Gaussian elimination:

```
row = array[1..n] of real;
element = class a: row;  b: real;
  no: integer; {original row number} pos: integer; {current row position} end;
```

```
procedure interact(var ei, ej: element);
begin { pivot ei, and ej, then eliminate xj from equation i } end;
```

The LU factorization step of Gaussian elimination is done in parallel, and the final solution is obtained sequentially by back substitution, which is implemented in the method *integrate*. This sequential post-processing does not affect the parallel performance significantly. Like bubble sort, Gaussian elimination requires a single pipeline step.

## 4 Cluster Implementation and Performance Evaluation

We first derive and validate parallel algorithms in *Paradigm/SP*, then we convert them into efficient *C* code that runs in the PVM cluster computing environment [8]. We believe that this approach simplifies the development and debugging of cluster computing applications. Note that the cluster implementations of *n*-body simulation, Gaussian elimination, and bubble sort all use the same generic parallel implementation of the all-pairs pipeline. Each specific cluster algorithm defines its domain-specific sequential components: *element* type, functions *interact*, *integrate*, and functions to *initialize* and *finalize* the system of elements.

The number of operations per node is inversely proportional to its position in the pipeline, e.g., the first node performs the most work while the last performs the least. If nodes are mapped to processors in a one-to-one fashion, the parallel execution of the pipeline algorithm suffers from load imbalance. To remedy this problem, our cluster implementation of the generic pipeline algorithm maps nodes onto processors using reflected cyclic mapping, which corresponds to "folding" the pipeline. This ensures that work is divided more evenly among the processors throughout the computation.

We have obtained performance results on a dedicated 100Mbps Ethernet cluster of 400Mhz Intel Pentium II dual-processor workstations with 1GB RAM per workstation. Experiments were performed on $p = 1, 2, 4, 8, 16, 32$ processors with pipelines consisting of $(f + 1) *p$ folded nodes for different values of the folding factor $f$. Problems with larger number of elements can usually benefit from a larger folding factor. Table 1 shows the run time $T(p)$ and processor efficiency $E(p) = T(1)/(p*T(p))$ for the following randomly generated problems:

- n-body simulation of systems of size $n = 10,000$ with a fold factor $f = 7$;
- bubble sort of integer arrays of size $n = 30,000$ with a fold factor $f = 7$;
- Gaussian elimination of systems of $n = 6,000$ equations with a fold factor $f =3$;

The speedup for the same experiments is illustrated in Fig. 5. Speedup is defined as $T(1)/T(p)$.

**Table 1**. Wall-clock time in seconds and processor efficiency.

| **p** | **N-Body Simulation** | | **Bubble Sort** | | **Gaussian Elimination** | |
|---|---|---|---|---|---|---|
| | T(p) | E(p) | T(p) | E(p) | T(p) | E(p) |
| 1 | 114 | 1.00 | 23 | 1.00 | 1525 | 1.00 |
| 2 | 42 | 1.36 | 7 | 1.64 | 815 | 0.94 |
| 4 | 22 | 1.30 | 4 | 1.44 | 486 | 0.78 |
| 8 | 12 | 1.19 | 2 | 1.44 | 295 | 0.65 |
| 16 | 7 | 1.29 | 2 | 0.72 | 199 | 0.48 |
| 32 | 4 | 0.89 | 1 | 0.72 | 186 | 0.26 |

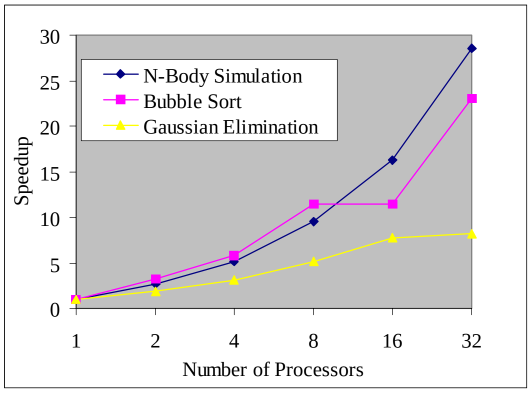

**Figure 5**. Speedup.

N-body simulation and bubble sort achieve super-linear efficiency because the total amount of available cache grows with the total number of processors. Gaussian elimination does not benefit much from a larger cache because its elements are considerably larger. Thus, only a small fraction of the node's retained elements can be retained in cache. The larger element size also leads to greater communication costs.

## 5 Conclusions

We believe that extending a *generic* parallel algorithm with *sequential* domain-specific code can result in good parallel performance in a cluster-computing environment. This parallel programming methodology leads to clean parallel solutions of a variety of problems that share the same parallel control structure. In ddition to providing good processor performance, genericity improves programming efficiency, allowing the application developer to focus on the sequential implementation of domain-specific details, rather than on the more difficult parallel code development. These conclusions are founded on our experience with several generic parallel algorithms: the all-pairs generic algorithm (described in the present paper), a generic master-server probabilistic algorithm, a cellular automaton generic algorithm, and a generic branch and bound algorithm [9].

The work presented in this paper is similar but not identical to earlier results in parallel raster image processing[10]. The dependency graph of an all-pairs computation is analogous to a raster image processing dependency graph. The two kinds of algorithms are implemented by means of similar pipelines. Parallel raster image processing does not involve *interactions* as those found in an all-pairs computation. The all-pairs algorithm proposed in this paper is a rather generic solution that may be applied to a large variety of problems.

This work has been supported by NASA grant NAG3-2011 and NSF Grant CCR-9509223.

## References

1. Amoura A., E. Bampis, J.-C. König. Scheduling Algorithms for Parallel Gaussian Elimination With Communication Costs, *IEEE Transactions on Parallel and Distributed Systems*, 9(7), July 1998, 679-686.
2. Arpaci-Dusseau A., R. Arpaci-Dusseau, D. Culler, J. Hellerstein, D. Patterson. High-Performance Sorting on Networks of Workstations, *Proc. 1997 ACM SIGMOD Conference*, ACM Press, 1997, 243—254.
3. Baiardi F., P. Becuzzi, P. Mori, M. Paoli. Load Balancing and Locality in Hierarchical *N*-Body Algorithms on Distributed Memory Architectures, *Lecture Notes in Computer Science*, 1401, Sringer, 1998, 284-294.
4. Hansen B. *Studies in Computational Science: Parallel Programming Paradigms*, Prentice Hall, Inc., Englewood Cliffs, NJ, 1995.
5. Radenski A. Module Embedding. Intl. Journal *Software - Concepts and Tools*, 19(3), 1998, 122-129.
6. Radenski A. Prototype Implementation of Paradigm/SP, www.rtpnet.org/~radenski/research/language.html, 1998.
7. Shih Z., G. Chen, R. Lee. Systolic Algorithms to Examine All Pairs of Elements, *Communications of the ACM*, 30, 1987, 161-167.
8. Sunderam V. PVM: A Framework for Parallel Distributed Computing, *Concurrency: Practice and Experience*, 2, No 4, 1990, 315-339.
9. Radenski A. Generic Parallel Message-Passing Algorithms and Their Applications, www.rtpnet.org/~radenski/research/algorithms.html, 1999.
10. Van Campenhout, J.M., Lasure, R., Kawahara, Y. PRIP - A Parallel Raster Image Processor, *Computer Graphics Forum*, 12, 1993, 95-104.