A Portable Parallel N-body Solver*

E Christopher Lewis† Calvin Lin† Lawrence Snyder† George Turkiyyah‡

Abstract

We present parallel solutions for direct and fast n-body solvers written in the ZPL language. We describe the direct solver, compare its performance against a sequential C program, and show performance results for two very different parallel machines: the KSR-2 and the Paragon. We also discuss the implementation of the fast solver in ZPL, including factors pertinent to data movement.

1 Introduction

Parallelism is an important means of obtaining high performance, but parallel programs are notoriously difficult to write. To reduce these programming costs, many high level languages have been proposed by the computer science community. However, the portability and performance of these languages have typically been established only for toy programs, and these languages have not been embraced by engineers and scientists interested in high performance computing. This paper demonstrates the feasibility of writing portable parallel programs in a high level language, ZPL [5], to solve a realistic problem: the N-body solution kernel of a high Reynolds number wind engineering simulation. Using ZPL, the parallel application has a clean and concise solution that achieves good performance on two widely different parallel architectures: the Kendall Square KSR-2 and the Intel Paragon.

The context of the problem is a wind engineering simulation for studying wind effects on buildings [7]. The objective is to understand the temporal and spatial distributions of the velocity and pressure fields around buildings and building complexes, and to assess the significance of geometric effects (building shape, nearby buildings, etc.) on the wind response so that improved design recommendations can be developed. The simulation uses a Lagrangian particle-based numerical scheme (vortex method) appropriate for fluid flows characterized by high Reynolds numbers and complex geometries. When coupled with a fast solver for computing vortex interactions, vortex methods appear to have several computational advantages over grid-based methods because they do not suffer from numerical diffusion; they are simpler to implement, particularly for complex geometries; and they are likely to exploit the architectures of distributed-memory computers more effectively.

The simulation uses a random vortex method and is coupled with two N-body solvers for computing the vortex interactions at each time step: One solver computes vortex interactions in a thin region around the ground and building boundaries (a "numerical" boundary layer), while another handles the interactions in the exterior region of the flow. This latter solver is by far the most computationally expensive part of the solution. Parallel

*This research was supported in part by ONR Contract N00014-92-J-1824, NASA grant NAG 2-831, and NSF Contracts CDA-9211095 and DDM-929622.

†Dept. of Computer Science and Engineering, FR-35, University of Washington, Seattle, WA 98195

‡Dept. of Civil Engineering, FX-10, University of Washington, Seattle, WA 98195

solutions to this component of the simulation are particularly important because meaningful 3D problems typically require hundreds of thousands of vortices.

This paper describes our preliminary implementation of portable, parallel solutions for this N-body solver. After a brief introduction to ZPL, we describe the implementation of a simple direct ($O(n^2)$) version and show its performance characteristics. We then describe the development of a fast version ($O(n)$) that exhibits a multigrid-like structure. We describe the data structures we use and show how the evaluation of vortex interactions can be effectively expressed in ZPL. We conclude by examining adaptive techniques that would make the fast solver applicable to problems with more irregular point distributions.

2 The ZPL Language

ZPL is an array sublanguage that provides support for data parallel computations [5]. As a sublanguage of Orca C—a lower level, more general language that supports MIMD parallelism—ZPL is free to be extremely clean and concise, avoiding any complicating features that do not pertain directly to data parallelism. In addition to most of the standard control flow constructs and data types that are found in languages such as Pascal, ZPL has the notion of *ensemble* arrays, which are given special support: Ensemble arrays are distributed across processors and can be manipulated as whole entities using new language constructs—*regions*, *directions*, and the *At* operator (`@`)—that eliminate tedious and error-prone array indexing and clearly expose communication to the compiler. ZPL also provides reduction and scan operators, as well as support for the clean specification of boundary conditions. (ZPL also has standard arrays, which we refer to simply as "arrays.")

A region is an index set that is applied to an entire statement or block of statements. The following code fragment shows the declaration of a region, `R`, and illustrates its use: The elements of ensemble array `B` that are in the index set {1..N} × {1..M} are assigned to the corresponding elements of the ensemble array `A`.

```
region R = [1..N, 1..M];

[R]  A := B;
```

A *direction* is a vector that is used with the At operator to shift an ensemble array reference by some user-defined distance. For example, the following code fragment shifts the elements of `A` to the right by one column.

```
direction west = [0,-1];

[R]  A := A@west;
```

Directions are also used to define neighboring regions. For example, `[west of R]` refers to the region that borders `[R]` to the left.

Scalar data types can be *promoted* to ensemble arrays. Similarly sequential functions can be *promoted*, *i.e.*, applied to ensemble array arguments, which encourages code re-use.

ZPL programs have sequential semantics, which allow them to be developed and debugged on workstations. The portability and good performance of ZPL programs stem from the parallel programming model—the Phase Abstractions [1, 3, 6]—upon which it is built. This programming model encourages locality of reference and parameterized control over granularity and communication. Previous studies have presented evidence that this model supports portability across diverse parallel machines [4]. Finally, an important feature of ZPL is the flexibility to specify at runtime key parameters that can affect communication granularity and data mapping.

3 Implementation of the Direct Solver

The simplest but most expensive strategy for computing vortex interactions computes all pairwise interactions individually. Therefore, each vortex must accumulate the effect of every other vortex. A general data structure for such an algorithm is a 1D ensemble array of cells, where each cell is a list of particles. This can be expressed in ZPL as:

```
var     vortices:  [R] array [1..M] of particle;
        potential: [R] array [1..M] of double;
region  R = [1..P];
```

The size, `P`, of the ensemble array and the number of particles to place in each cell of the array can be specified at run-time. This scheme provides flexibility in choosing an appropriate granularity of parallelism. One would select `P` to suit the target architecture, for example to match the number of processors or the machine's ideal granularity of communication. By default, the ZPL compiler maps 2D ensemble arrays to processors in a 2D blocked fashion.

The algorithm consists of `P` iterations (shown below). Each iteration shifts the contents of a cell to its neighbor using a torus topology. In ZPL the At operator shifts an entire cell's contents, in this case `M` particles, and the cyclic shift is completed using the `wrap` operator, which connects ends of an array as in a torus.

The computation portion of an iteration involves computing the pairwise interactions of just two cells. The procedure `add_carrier_effects()` is a scalar procedure that is defined on arrays. The parallelism comes implicitly from the ensemble array data structure. The code itself is identical to what would be written for a scalar computation. It is a nested loop to compute the effects of the particles in a visitor cell on the particles in the cell. ZPL allows this function to be promoted to operate on ensemble arrays. This promotion of scalar functions to array functions greatly simplifies the programming process.

```
[R] begin
      read_input("data", vortices);
      potential := 0.0;              -- initialize all particles of all cells

      carrier := vortices
      for i := 1 to P do
[left of R]  wrap carrier;                          -- send right cell to left
             carrier := carrier@left;               -- shift cells left
             add_carrier_effects(vortices, carrier, potential);
      end;
      write(phi);
    end;
```

The ZPL compiler produces ANSI C code that can execute on any number of processors. This output code is identical for all target machines but is linked with a small machine-dependent library that defines operations such as message sends and receives. For shared memory machines such as the KSR-2, message passing is implemented as shared queues. The compiler currently performs no machine-specific optimizations.

Figure 1 shows that the direct solver achieves good speedup relative to the hand-coded C version on the Intel Paragon. The Paragon is a mesh-connected distributed memory computer with Intel i860 processors and 16MB of memory per node. The ZPL program running on one processor is 7.9% slower than the C version. With 16 processors the speedup is 13.76, and the relative speedup (based on the ZPL program running on one processor) is 14.85.

The KSR-2 is a shared memory multiprocessor with a ring of rings interconnection structure. Each processor runs at 40Mhz and has 32MB of local memory. On the KSR-2 the ZPL program is 15% slower than C. For 16 processors the speedup relative to C is 11.47, and the relative speedup is 13.23. There are two reasons for the lower speedup on the KSR. The processor is faster, making the cost of communication relatively more expensive, and our current message passing implementation copies data more times than is necessary. With larger problem sizes the speedup will naturally improve.

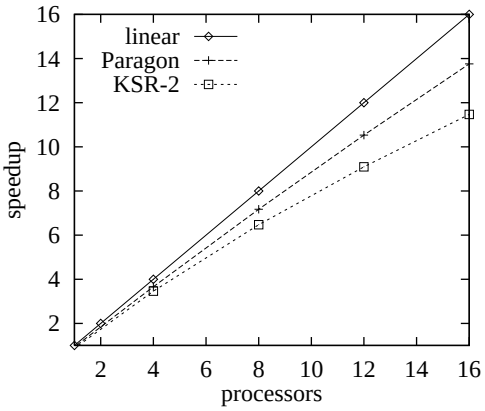

FIG. 1. *Speedup for the Direct Code (6000 particles)*

4 Implementation of a Fast Solver

A direct method that evaluates all interactions is prohibitively expensive for realistic simulations. Fast solvers exploit the fact that the effect of a neighboring particle decreases as its distance increases. This section describes the implementation of a fast vortex method that requires in principle a linear amount of work. Computational savings in the fast solver are obtained by combining large numbers of particles into a small set of discrete values (a ring) whose effect approximates the effect of the cluster of particles. There are two kinds of cluster approximations: "outer-rings" that represent the effect of a cluster in the far field and "inner rings" that represent the effect of far-away particles in the near field. Fast solvers construct and evaluate these cluster approximations in a hierarchical fashion, much like multigrid solvers. The approximations we use are based on Poisson's formula [2].

The data structure consists of a hierarchy of distributed grids that store these ring approximations and transfer information between adjacent levels. The particles are stored at the finest level as an ensemble array of lists of particles, as shown below.

```
region   R3 = [1..N/2, 1..M/2];            -- N and M can be set at runtime
         R4 = [1..N, 1..M];

type      part_list = record
                        count: integer;
                        list : array [1..Max] of particle;
                      end;

var       vortices : [R4] part_list;
          ring4    : [R4] ring;
          ring3    : [R3] ring;
```

```
direction sw2 = [2, -2];    north = [-1, 0];
```

The implementation of this solver is similar to that of one V-cycle of a multigrid method. As shown below, a first sweep starts from the finest level and builds outer approximations of the velocity vector at all levels. A second sweep builds inner approximations from the coarsest level down to the finest. These sweeps require inter-level communication. In the descent phase, intra-level communication is needed to compute ring-ring interactions between a cell and its well-separated neighbors whose parents are not well-separated from the cell's parent. Such neighbors are at most three cells away. The effects of particles in neighboring cells are computed directly at the finest level.

```
procedure fast();
 [R4] begin
              initialize();
              ORAfinest(vortices, ring4);
              go_up_43(ring3, ring4);           -- inter-level communication
 [R3]         go_up_32(ring2, ring3);
 [Expand2(R2)] visit_2_neighbors(ring2);        -- intra-level communication
 [Expand3(R3)] visit_3_neighbors(ring2);        -- intra-level communication
 [R3]         go_down_23(ring2, ring3);
               . . .
              neighbor_contributions(vortices); -- nearest neighbor communication
      end;
```

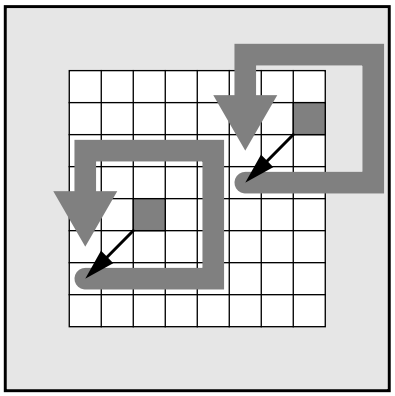

FIG. 2. *Intra-level Communication with 2-Neighbors.*

Figure 2 shows the intra-level communication with 2-neighbors. The corresponding ZPL code is shown below. First, for each cell, the southwest neighboring cell is copied to the local cell. Then, a sweep is made around the box, accumulating the effects of ring-ring interactions. To specify the data motion corresponding to the thick arrow in Figure 2, the `visit_2_neighbors()` function is logically invoked in the shaded region shown in Figure 2 (this is expressed above using a macro called Expand2 that expands to a list of regions). Note that the same code applies to all cells, regardless of boundary conditions. The compiler does not generate communication for "neighbors" that lie outside the data space (*i.e.*, when the thick arrow enters the shaded region). The 3-neighbor communication is analogous, although not all 3-neighbors are needed.

```
procedure visit_2_neighbors(var ring: [2] Box); -- ring is a 2D ensemble array
 var tmp : [R] Box;
     i   : integer;
 begin
```

```
    tmp := ring;

    -- add the contributions of the 2-neighbors
    tmp := tmp@sw2;                              -- translate to southwest
    for i := 1 to 4 do
        tmp := tmp@north;
        add_contributions_OR_lm(ring, tmp);
    end;
    . . .                                -- repeat for north, east, south and west
end;
```

5 Conclusions

ZPL allows the elegant expression—even when compared against sequential programs—of both the direct and fast N-body solvers. The direct solver achieved good performance on two radically different architectures. We also expect good speedup for the fast solver.

In a complete wind simulation the fast solver must be invoked at each time step. Each time step introduces new particles to the simulation to satisfy appropriate boundary conditions. Furthermore, as particles already present in the simulation domain move, they may cross cell boundaries of the finest grid level or may leave the domain altogether. Particles must therefore be redistributed for the next time step of the N-body solver. Our ZPL implementation specifies the movement of these particles across the simulation domain, but leaves the details of inter-processor communication to the compiler. These will be shown in an upcoming report.

A general issue with all multi-level computations is the mapping of the different levels of the hierarchy to processors. For example, grids can be aligned eccentrically to minimize communication (as shown above in our region definitions) or aligned concentrically to maximize parallelism. The best choice will likely depend on the architecture. ZPL allows these mappings to be specified *at runtime*, and we intend to explore these issues on various machines.

References

[1] Gail Alverson, William Griswold, David Notkin, and Lawrence Snyder. A flexible communication abstraction for nonshared memory parallel computing. In *Proceedings of Supercomputing '90*, November 1990.

[2] Christopher R. Anderson. An implementation of the fast multipole method without multipoles. *SIAM Journal of Sci. Stat. Computing*, 13(4):923–947, July 1992.

[3] William Griswold, Gail Harrison, David Notkin, and Lawrence Snyder. Scalable abstractions for parallel programming. In *Proceedings of the Fifth Distributed Memory Computing Conference*, 1990. Charleston, South Carolina.

[4] Calvin Lin and Lawrence Snyder. A portable implementation of SIMPLE. *International Journal of Parallel Programming*, 20(5):363–401, 1991.

[5] Calvin Lin and Lawrence Snyder. ZPL: An array sublanguage. In Uptal Banerjee, David Gelernter, Alexandru Nicolau, and David Padua, editors, *Languages and Compilers for Parallel Computing*, pages 96–114. Springer-Verlag, 1993.

[6] Lawrence Snyder. Foundations of practical parallel programming languages. In *Proceedings of the Second International Conference of the Austrian Center for Parallel Computation.* Springer-Verlag, 1993.

[7] George Turkiyyah, Dorothy Reed, and Jiyao Yang. Fast vortex methods for predicting wind-induced pressures on building systems. *submitted for publication*, 1993.