Integrating Task and Data Parallelism with the Collective Communication Archetype*

K. Mani Chandy, Rajit Manohar, Berna L. Massingill, Daniel I. Meiron
California Institute of Technology
Pasadena, CA 91125

Abstract

A parallel program archetype aids in the development of reliable, efficient parallel applications with common computation/communication structures by providing stepwise refinement methods and code libraries specific to the structure. The methods and libraries help in transforming a sequential program into a parallel program via a sequence of refinement steps that help maintain correctness while refining the program to obtain the appropriate level of granularity for a target machine. The specific archetype discussed here deals with the integration of task and data parallelism by using collective (or group) communication. This archetype has been used to develop several applications.

1 Introduction

Archetypes. Many parallel applications share common features in design, testing, debugging, performance tuning, and program structuring. A parallel program *archetype* is an abstraction that embodies common features shared by parallel applications within a domain. An archetype aids the development of reliable, efficient applications within a domain by providing design methods and code libraries appropriate for the domain, with specific emphasis on stepwise refinement and selection of the appropriate degree of granularity. Archetypes are language-independent; they allow application developers to continue to use the sequential languages and program development environments with which they are familiar.

Reuse of design methods. The design methods and refinement techniques provided by a parallel program archetype can be reused for many applications with a common computation/communication structure, improving productivity.

Use of familiar tools. The methods and techniques provided by an archetype are also language-independent; the code libraries associated with an archetype can be implemented using different parallel languages and libraries (for example, Fortran M [12] and PVM [3]). Stepwise refinement allows much of the work of developing a parallel program to be done in a sequential environment, using familiar tools and techniques; the choice of a particular parallel language or library can be deferred until the last step of the refinement process. This contributes both to productivity, by allowing developers to do most of their work in a familiar environment, and to portability, by minimizing the work required to convert an application to use a different parallel language or library.

Integration of task and data parallelism. The archetype presented in this paper is the *collective communication archetype*, which we use to integrate task and data parallelism and which is appropriate for a variety of applications including mesh, spectral, and splitting computations. The archetype is helpful in developing programs structured as task-parallel compositions of SPMD (single program, multiple data) programs. Task-parallel compositions of SPMD programs are appropriate in a number of situations. Total system (e.g., global climate) simulations are composed of different kinds of subsystems (e.g., ocean, land, and atmosphere simulations). Similarly, computer-aided design tools used in the design of total systems require the integration of programs dealing with different disciplines—for example, design of a satellite system involves thermal vibration, optimum trajectory, and costing calculations.

Compositionality. The collective communication archetype focuses on organizing processes into groups and on defining collective communication between and

*This work was supported in part by the AFOSR under grant number AFOSR-91-0070, and in part by the NSF under Cooperative Agreement No. CCR-9120008. The government has certain rights in this material.

within these groups. A key feature of archetypes is an emphasis on *compositionality*: defining ways to combine units into larger units such that the internal details of each unit are hidden. Compositionality facilitates code reuse.

Organization of this paper. The remainder of this paper is organized as follows: §2 presents two motivating examples of integrating task and data parallelism. §3 through §6 describe the collective communication archetype: §3 describes how the archetype organizes collections of processes into process groups and process networks; §4 describes the archetype's model of distributed data; §5 discusses communication between process groups; and §6 discusses communication within process groups. §7 describes a method for program development based on the archetype. §8 describes implementation experiments. §9 discusses related work. §10 summarizes results so far and suggests directions for future work.

2 Examples of integrating task and data parallelism

In this section, we present two examples of problems that can be structured as task-parallel compositions of SPMD programs; these examples illustrate some computation and communication structures encompassed by the collective communication archetype.

2.1 Climate simulation

Consider a climate-model simulation consisting of two coupled simulations, an ocean simulation and an atmosphere simulation, that exchange boundary data periodically, as shown in figure 1.

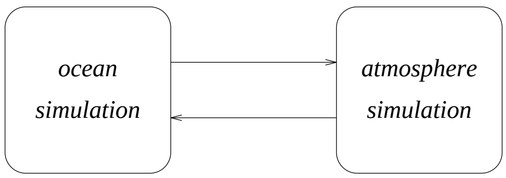

Figure 1: Climate simulation.

Such a simulation can be refined into a task-parallel composition of two SPMD calculations, as shown in figure 2. Observe that exchanging boundary data between the two simulations requires communication paths between boundary processes in the ocean simulation and corresponding boundary processes in the atmosphere simulation.

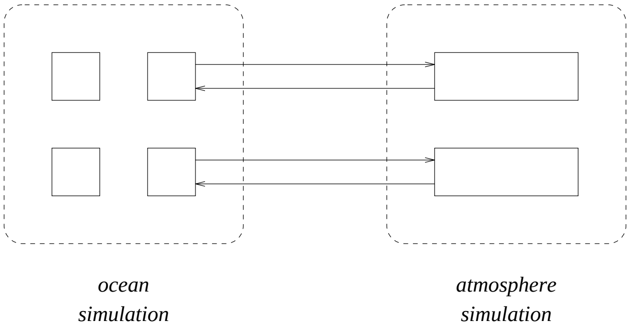

Figure 2: Climate simulation (refinement).

2.2 Pipelined computation using FFT

A number of computations (such as convolution, correlation, and filtering) can be represented by the following sequence of steps: (1) Perform a fast Fourier transform (FFT) on the data (assumed to be one-dimensional); (2) manipulate the result of the transform elementwise; and (3) perform an inverse FFT on the result of the manipulation.

To perform this sequence of steps on many sets of data, we can use a pipeline, with one stage for each of the above steps, as shown in figure 3.

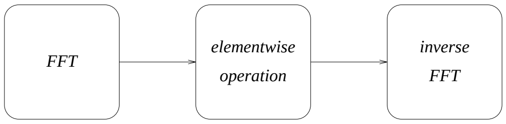

Figure 3: FFT pipeline.

This computation can be refined into a composition of three SPMD computations, one for each stage of the pipeline, as shown in figure 4.

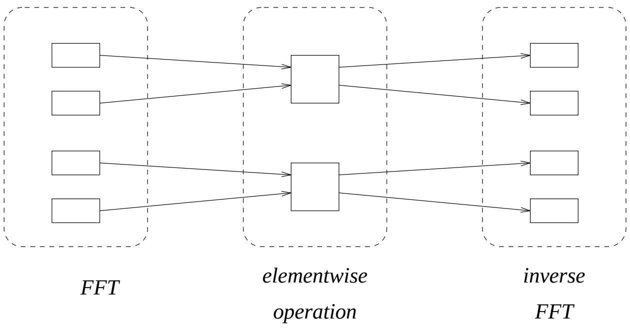

Figure 4: FFT pipeline (refinement).

3 Process structures

3.1 Overview

The collective communication archetype organizes collections of processes using two constructs: *process groups*, whose members are identical, and *process networks*, whose members need not be identical. These constructs are analogous to the array and record constructs used in defining composite data types.

The process group is analogous to the array construct: An array is a composition of data items of the same type, in which each data item can be referenced by an index. Similarly, a process group is a composition of identical processes, in which each process can be referenced by an index (relative to the process group).

The process network is analogous to the record construct: A record is a composition of data items of different types, in which each data item can be referenced by a name or other non-numeric handle. Similarly, a process network is a composition of non-identical processes or process groups, in which each process or group can be referenced by a name or other handle.

Like the array and record constructs, process groups and networks can be used to define nestings of arbitrary complexity. However, in this paper we focus on simple one- and two-level nestings: single process groups (effective for representing SPMD computations) and networks of process groups (effective for representing task-parallel compositions of SPMD computations).

For example, the climate-model simulation of §2.1 is organized as a network of two process groups, as shown in figure 5. (We defer consideration of communication until §5.)

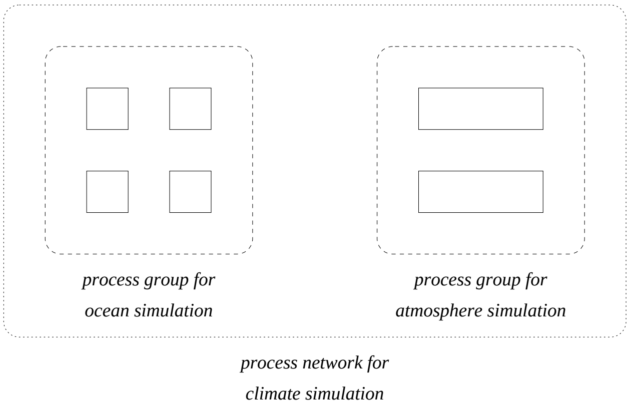

Figure 5: Two-level nesting for climate simulation.

3.2 Details

In a one- or two-level nesting, a process group represents an SPMD computation—a collection of identical processes executing the same program. We can create a process group thus:

```
create_group(
    group_handle,
    executable_file_name,
    number_of_processes,
    number_of_processors,
    array_of_processors,
    processes_to_processors_map,
    number_of_sending_ports,
    array_of_sending_ports,
    number_of_receiving_ports,
    array_of_receiving_ports
    ) ;
```

where:

- `group_handle` (output) uniquely identifies this process group.
- `executable_file_name` (input) is the name of the program to be executed by each process.
- `number_of_processes` (input) is the number of processes in the group.
- `number_of_processors` (input) is the number of processors to be used.
- `array_of_processors` (input) identifies the processors to be used for the group.
- `processes_to_processors_map` (input) indicates how processes are mapped to processors—e.g., by blocks or cyclically.
- `number_of_sending_ports` (input), `array_of_sending_ports` (input), `number_of_receiving_ports` (input), and `array_of_receiving_ports` (input) define how this process group communicates with other process groups, as described in §5.2.

We combine process groups into networks as follows:

```
begin_network() ;
    create_group(group1, ....) ;
    create_group(group2, ....) ;
    ....
end_network() ;
```

For example, the network for the climate simulation of figure 5 would be defined thus:

```
begin_network() ;
    create_group(ocean,
        "oceansim", 4,
        4, processor_array_ocean, BLOCK,
        ....) ;
    create_group(atmosphere,
        "atmosim", 2,
        2, processor_array_atmosphere, BLOCK,
        ....) ;
    ....
end_network() ;
```

4 Distributed data structures

In the collective communication archetype, data structures can be distributed over process structures (networks and groups), with the members of a particular data structure distributed among the members of a particular process structure. In this paper, we restrict attention to distributing arrays over process groups, in much the same manner as arrays are distributed over SPMD computations in Fortran D [13] and HPF [14].

In our model, a distribution of a data structure over a process structure can be completely specified by a one-to-one map from global indices onto {process-number, local-indices} pairs. Thus, each element of a distributed array corresponds to exactly one element of one process's local section.

5 Communication between groups

5.1 Overview

Communication between groups is collective and based on transmitting all or part of a distributed data structure from one process group to another. Our model of group-to-group communication is intended to meet two goals: flexibility and compositionality.

Our model of group-to-group communication is flexible in that it allows transmitting all or part of a data structure from one group to another. It supports both the transmission of whole data structures needed for the FFT pipeline example of §2.2 and the transmission of partial data structures needed for the climate simulation example of §2.1.

Our model is compositional in that it allows two process groups to communicate without knowing the details of each other's internal structure and data distribution. If process groups A and B are composed in a process network, the distribution of data structures in process group A can be varied without requiring code changes in process group B.

For simplicity, we define communication between groups in terms of group sends and group receives. In a group send, the sending group collectively sends part or all of one of its distributed data structures; in a group receive, the receiving group receives into all or part of one of its distributed data structures. The sending group and the receiving group can be different groups, or they can be the same group (in which case group communication is equivalent to intragroup collective communication). The data being sent need not be distributed in the same way in the sender as in the receiver, nor need the sending group and the receiving group have the same structure or number of processes, though the total amount of data to be sent must be the same as the total amount of data to be received.

This intergroup communication problem is a special case of the interoperability problem of interfacing different types, as shown in figure 6.

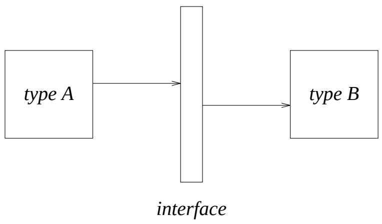

Figure 6: The interoperability problem.

In our case, types A and B are each defined by two functions—a distribution function from global to local indices, and a restriction (or subsetting) function on the global index set—and the interface is defined as a mapping from one (restricted) index set to the other. Figure 7 illustrates this for the climate-model simulation example of §2.1.

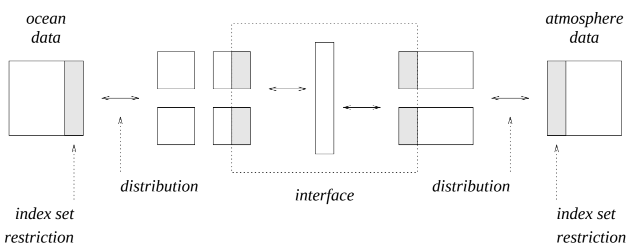

Figure 7: Interoperability for the climate simulation.

While support for the general case (any combination of distribution, restriction, and interface mappings) is possible in principle, it may be cumbersome to implement, so we restrict attention to a few common types that suffice to cover many applications.

As an additional simplification, we require that all group-to-group communication take place over explicitly-defined typed communication paths or *channels*, analogous to the typed process-to-process channels of Fortran M [12]. A group-to-group channel is typed with the distribution and restriction functions of both the sender and the receiver as well as the interface mapping. This information makes it possible to translate each group-to-group channel into a set of process-to-process communication paths, as illustrated in the refinement steps in §2.1 and §2.2.

5.2 Details

Group-to-group channels are declared at the level of the enclosing nesting; each path establishes two ports, a sender and a receiver, which are passed to the process groups. A group-to-group channel is defined thus:

```
channel(
    sending_group_handle,
    receiving_group_handle,
    sending_group_distribution,
    sending_group_restriction,
    receiving_group_distribution,
    receiving_group_restriction,
    interface_map,
    sending_port,
    receiving_port
    ) ;
```

where:

- `sending_group_handle` (input) and `receiving_group_handle` (input) are group handles defined by `create_group` operations, and can be identical.
- `sending_group_distribution` (input) defines the distribution of the data structure from which data to be sent is extracted. `sending_group_restriction` (input) defines what part of the data structure is to be sent. Similarly, `receiving_group_distribution` (input) and `receiving_group_restriction` (input) define the distribution of the data structure into which data is to be received and indicate what part is to be received into.
- `interface_map` indicates how the subset of data sent is to be mapped into the subset of data to be received—e.g., normal or transposed.
- `sending_port` (output) and `receiving_port` (output) can then be included in sending and receiving groups' `array_of_sending_ports` and `array_of_receiving_ports`.

Group sends and group receives are then performed SPMD-style by all processes in the process group using the sending and receiving ports defined by this `channel()` operation, thus:

```
group_send(
    sending_port,
    data_location,
    )
group_receive(
    receiving_port,
    data_location,
    )
```

where:

- `sending_port` (input) and `receiving_port` (input) are the result of a `channel()` operation.
- `data_location` (input for `group_send` and output for `group_receive`) is the array from which data is to be sent or into which data is to be received. Observe that the channel, and thus the port, specifies what parts of the data are to be sent or received.

To exchange boundary information between the coupled simulations of §2.1 requires two channels, one in each direction. We expand the process-network definition of §3.1 thus:

```
begin_network() ;
    create_group(ocean,
        "oceansim", 4,
        4, processor_list_ocean, BLOCK,
        1, (ocean_send_port),
        1, (ocean_receive_port)) ;
    create_group(atmosphere,
        "atmosim", 2,
        2, processor_list_atmosphere, BLOCK,
        1, (atmosphere_send_port),
        1, (atmosphere_receive_port)) ;
    channel(ocean, atmosphere,
        BLOCK(2,2),
        EAST_BOUNDARY(1),
        ROW(2),
        WEST_GHOST_BOUNDARY(1),
        NORMAL,
        ocean_send_port,
        atmosphere_receive_port) ;
    channel(atmosphere, ocean,
        ROW(2),
        WEST_BOUNDARY(1),
        BLOCK(2,2),
        EAST_GHOST_BOUNDARY(1),
```

```
        NORMAL,
        atmosphere_send_port,
        ocean_receive_port) ;
end_network() ;
```

6 Communication within groups

Some collective communication operations within a process group can be defined as intergroup communication actions in which the sending and receiving groups are identical. However, there are also collective-communication operations that make sense when applied within a process group but not when applied to two process groups. (Extending the data types analogy of §3.1, there are operations that make sense when applied to arrays but not when applied to records.) Within a group, therefore, the collective communication archetype permits intragroup collective communication operations based on point-to-point communication with other members of the group. Such operations include internal boundary exchange (the exchange of local section boundaries between neighboring processes) and global reduction operations (e.g., global maximum or sum).

7 Stepwise refinement

7.1 Stepwise refinement for correctness

Parallel program archetypes are particularly useful in guiding both stepwise development of new parallel programs and stepwise parallelization of existing sequential programs. Starting with a sequential algorithm or program, the application developer can apply a sequence of small semantics-preserving transformations (refinements) to produce a parallel version of the original algorithm or program. All intermediate stages of this process are sequential programs and can thus be developed, tested, and debugged using familiar tools and techniques. The last intermediate stage is a sequential *simulated-parallel* program that can be transformed into an equivalent real-parallel program in a simple, mechanical way.

The role of an archetype in this process is twofold. First, it guides the process by providing a framework for the simulated-parallel and real-parallel versions. Second, it provides a program skeleton and collective communication routines that encapsulate the details of both simulated and real parallelism.

7.1.1 Simulating parallelism

A key feature of our approach to the stepwise development of parallel programs is the simulated-parallel version of the program. The simulated-parallel version of the program under development addresses all of the difficulties of the real parallel program (interleaved execution, distributed memory, and interprocess communication and synchronization) in the context of a sequential program that can be developed and debugged with familiar tools. Transforming this simulated-parallel program into an equivalent "real-parallel" program is straightforward and preserves correctness.

In the general case, simulating the operation of a parallel program is a problem in discrete-event simulation, and producing and debugging such a simulation can be difficult. However, the collective communication archetype restricts interprocess interaction to collective communication operations, which simplifies the simulation, since the computation of each process can be expressed as a sequence of local-computation sections and collective communication operations.

In the simulated-parallel version of the program, processes are represented by simulated processes, and parallel execution is simulated by looping sequentially over all simulated processes, for each local-computation section or collective communication operation. Distributed memory is simulated by giving each simulated process a unique copy of program variables; each simulated process has access only to its own copy of the variables. Communication operations are simulated using simulated channels (implemented as queues), in which an attempt to receive from an empty channel is an error. (This restriction allows us to claim that if the simulated-parallel version is correct, its real-parallel equivalent is also correct.)

The details of this simulation—looping over simulated processes and simulating communication—are provided as part of the archetype implementation.

7.1.2 Example

As an example, consider a spectral-methods computation, in which data is organized as a two-dimensional array, and the required calculations are composed of row operations (in which an operation is applied independently to each row of the array) and column operations (in which an operation is applied independently to each column of the array). Such computations arise in some computational fluid problems, in which the row and column operations are fast Fourier transforms or vectorized matrix solves. An effective way of parallelizing such computations is based on alternately distributing the data by rows and by columns; for row operations, the data is distributed by rows, and all rows can be operated on in parallel, while for column operations, the data is distributed by columns, and all columns can be operated on in parallel. The required communication operations consist of row-to-column and column-to-row redistributions—group-to-group communications in which the sending and receiving groups are the same.

We can abstract from such computations a specialized version of the collective communication archetype, the *spectral-methods archetype*. We now describe how this archetype can be used to guide development of a parallel program.

We begin with a sequential version of the program, as illustrated by figure 8.

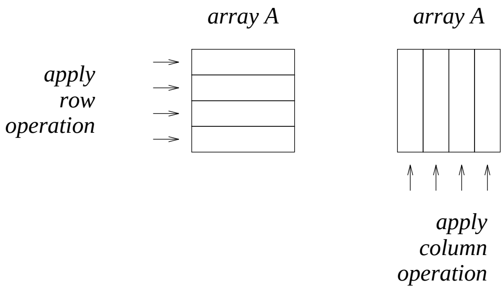

Figure 8: Spectral-methods archetype, step 1.

We next observe that column operations on the original matrix are equivalent to row operations on its transpose, so an equivalent sequential program is the one illustrated by figure 9.

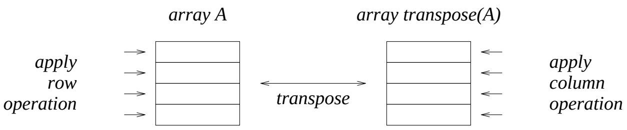

Figure 9: Spectral-methods archetype, step 2.

Since row operations operate on all rows independently, they can be performed in parallel; similarly for column operations. Thus, we can partition the rows of A among processes and perform row operations in parallel; similarly, we can partition the rows of A^T (columns of A) among processes and perform column operations in parallel, as shown in figure 10.

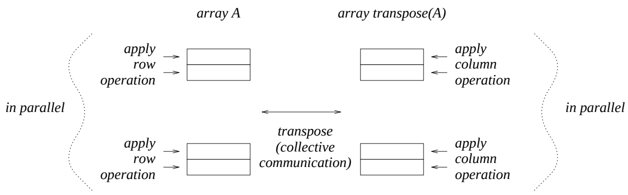

Figure 10: Spectral-methods archetype, step 3.

This figure illustrates the operation of both the simulated-parallel version and the real-parallel version. The difference is that the simulated-parallel version is a sequential composition of simulated-parallel compositions:

```
repeat
    for all simulated processes P
        for all rows J in local section P
            perform row operation on row J
        endfor
    endfor
    perform simulated transpose
    for all simulated processes P
        for all rows J in local section P
            perform column operation on row J
        endfor
    endfor
    perform simulated transpose
endrepeat
```

while the real-parallel version is a parallel composition of sequential compositions:

```
repeat
    parallelfor all processes P
        for all rows J in local section P
            perform row operation on row J
        endfor
        perform transpose
        for all rows J in local section P
            perform column operation on row J
        endfor
        perform simulated transpose
    endparallelfor
endrepeat
```

7.2 Stepwise refinement for efficiency

Parallel program archetypes can also be helpful in refining an initial parallelization to improve its efficiency. For example, if the archetype parameterizes the quantities that determine granularity (e.g., number of processes and size of messages), the application developer can easily experiment with using different values for these quantities to improve performance.

Archetype-based performance models can also help application developers in tuning for performance.

8 Implementation experiments

In this section, we describe a number of implementation experiments, in which work on one or more real applications was used to guide development of an archetype implementation (code library and program skeleton). The archetypes discussed here are specialized versions of the collective communication archetype; the implementations are based on a variety of parallel languages and libraries — Fortran M [12], p4 [5] with Fortran, PVM [3] with Fortran, and PVM with C.

8.1 Single-group implementation experiments

8.1.1 Mesh-computation archetype

In this archetype, data is distributed over a two- or three-dimensional mesh. The basic structure of the computation is a time-step loop, at each step of which new values are computed for each point in the mesh, with a point's new values being computed as a function of the old values at the point and at its neighbors.

This computation parallelizes readily; the mesh is distributed among processes, and the required communication operations consist of an exchange of boundary values with neighboring processes and (for some computations) a global reduction operation.

This archetype has been used with two applications. A two-dimensional version was used to parallelize a computational fluid dynamics code that simulates high Mach number compressible flow using a conservative and monotonicity-preserving finite difference scheme [18]. In addition, a three-dimensional version was used to parallelize a computational electromagnetics code that performs numerical simulation of electromagnetic scattering, radiation, and coupling problems [2].

8.1.2 Spectral-methods archetype

In this archetype (described also in §7.1.2), data is a two-dimensional array, and the computation consists of a sequence of alternating row operations (operations applied to each row) and column operations (operations applied to each column).

As described in §7.1.2, an effective way of parallelizing such computations is based on alternately distributing the data by rows and by columns; for row operations, the data is distributed by rows, and all rows can be operated in parallel, while for column operations, the data is distributed by columns, and all columns can be operated on in parallel. The required communication operations consist of row-to-column and column-to-row redistributions—group-to-group communications in which the sending and receiving groups are the same.

Program skeletons and communication routines for this archetype have been implemented, but have not yet been used to develop application programs. Examples of spectral-methods computations are found in computational fluid dynamics, where the row and column operations are fast Fourier transformations or vectorized matrix solves.

8.1.3 Linear-algebra archetype

This archetype is based on the parallel algorithms for matrix and vector operations described in [21]. Data consists of one-dimensional arrays (vectors) and two-dimensional arrays (matrices), and the computation consists of vector and matrix operations.

Parallelization is based on distributing arrays across a grid of processes; required communication operations are for the most part based on recursive doubling.

This archetype has been implemented and used to develop code for some of the vector and matrix algorithms in [21].

8.2 Multiple-group implementation experiments

8.2.1 Splitting archetype

In this archetype, data is based on a two-dimensional grid, and computation is a sequence of operations of the following types: reading data from a (sequential) input file; performing a row operation (an operation applied to each row); performing a column operation (an operation applied to each column); performing local calculations on each cell; and writing data to a (sequential) output file.

One approach to parallelizing this computation involves creating four process groups: a single host-process group for reading and writing sequential files; a group of row processes for performing row operations; a group of column processes for performing column operations; and a group of local processes for performing local calculations on grid cells. Row and column operations and local calculations can then be done in parallel; the required communication consists of redistribution operations (e.g., from the row distribution of the row-process group to the two-dimensional block distribution of the local-process group).

This archetype has been used to parallelize the CIT airshed model, which models smog in the Los Angeles basin [10].

8.2.2 General collective communication archetype

We have also implemented a preliminary version of the general collective communication archetype described in §3.2; this implementation allows the user to define networks of process groups and perform point-to-point communication between processes in different groups. Channel-based group-to-group communication as described in §5.2 has not yet been implemented.

9 Related work

A great deal of research has been done on software reuse [15] and templates [22]. A difference between our work and much of this previous work is that our archetype includes stepwise-refinement design methods, and our focus is on *design* reuse as much as code reuse, particularly for parallel computing targeted to different architectures. Much of our research deals with capturing design steps in a concrete form so that the design process can be reused, whereas other work on templates (e.g., [1]) is more concerned with reuse of programming skeletons. In this emphasis on design methods based on stepwise refinement, we follow the work of Eric Van de Velde [21].

Another difference between our work and much of the earlier work is that we deal with the integration of task and data parallelism in a language-independent and environment-independent manner. Programmers can use our archetypes with languages and tools with which they are familiar. Many of the ideas on compositionality described here are based on ideas in Fortran M [12]. The integration of Fortran M and Fortran D [7, 11] is particularly relevant. Likewise, the integration of pC++ [4] with CC++ [8, 6] provides an object-oriented language that allows for integration of task and data parallelism. Merlin [9] also takes an abstraction-based approach to integrating task and data parallelism, but it uses monitors rather than channel-based composition.

We differ from collective communication definitions like MPI [16], Zipcode [20], and PARTI [17] and from libraries of collective operations like pC++ [4] and the Multicomputer Toolbox [19] in our emphasis on stepwise refinement, abstraction, and the integration of task and data parallelism.

10 Summary

The collective communication archetype aids in the development of reliable, efficient parallel programs for many applications by providing design methods based on stepwise refinement and code libraries; it is particularly effective for integrating task and data parallelism. The archetype is language-independent; archetype implementations based on different parallel languages and libraries allow application developers to continue to use the languages and program development environments with which they are familiar.

Much work, however, remains to be done. The implementation experiments described in this paper will be extended and unified into a library of archetypes and associated tools. The ideas about process structure presented here will be generalized from simple one- and two-level nestings to arbitrary compositions of process groups and networks, with definitions of distributed data structures and collective communication extended to apply to these arbitrary compositions.

Acknowledgments

Many of the ideas here were motivated by the work on paradigm integration by Ian Foster, particularly in the context of Fortran M. The ideas about stepwise refinement are based in large part on the work of Eric Van de Velde.

In addition, we thank John Beggs and Donald Dabdub for their help in providing and explaining applications, Sharif Rahman for his help with implementation experiments, and Adam Rifkin for his constructive criticism of this paper.

References

[1] R. Barrett, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods.* SIAM, 1993.

[2] J. H. Beggs, R. J. Luebbers, D. Steich, H. S. Langdon, and K. S. Kunz. User's manual for three-dimensional FDTD version C code for scattering from frequency-independent dielectric and magnetic materials. Technical report, The Pennsylvanie State University, July 1992.

[3] A. Beguelin, J. Dongarra, A. Geist, and B. Manchek. A user's guide to PVM parallel virtual machine. Technical Report ORNL TM-1126, Oak Ridge National Laboratory, 1991.

[4] F. Bodin, P. Beckman, D. Gannon, S. Narayana, and S. X. Yang. Distributed pC++: Basic ideas for an object parallel language. *Scientific Programming*, 2(3):7–22, 1993.

[5] R. Butler and E. Lusk. User's guide to the p4 parallel programming system. Technical Report ANL-92/17, Argonne National Laboratory, 1992.

[6] P. Carlin, K. M. Chandy, and C. Kesselman. The Compositional C++ language definition. Technical Report CS-TR-92-02, California Institute of Technology, 1992.

[7] K. M. Chandy, I. T. Foster, K. Kennedy, C. Koelbel, and C.-W. Tseng. Integrated support for task and data parallelism. *International Journal of Supercomputer Applications*, 8(2), 1994. In press.

[8] K. M. Chandy and C. Kesselman. CC++: A declarative concurrent object oriented programming language. Technical Report CS-TR-92-01, California Institute of Technology, 1992.

[9] B. Chapman, P. Mehrotra, J. Van Rosendale, and H. Zima. Language extensions for multidisciplinary applications. Unpublished document.

[10] D. Dabdub and J. H. Seinfeld. Air quality modeling on massively parallel computers. *Atmospheric Environment*, 1994. To appear.

[11] I. T. Foster, B. Avalani, A. Choudhary, and M. Xu. A compilation system that integrates High Performance Fortran and Fortran M. In *Proceedings: 1994 Scalable High Performance Computing Conference*. IEEE, 1994. To appear.

[12] I. T. Foster and K. M. Chandy. FORTRAN M: A language for modular parallel programming. *Journal of Parallel and Distributed Computing*, 1994. To appear.

[13] G. C. Fox, S. Hiranandani, K. Kennedy, C. Koelbel, U. Kremer, C.-W. Tseng, and M.-Y. Wu. Fortran D language specification. Technical Report CRPC-TR90079, Center for Research on Parallel Computation, December 1990.

[14] High Performance Fortran Forum. High Performance Fortran language specification, version 1.0. Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, 1992 (revised Jan. 1993).

[15] C. W. Krueger. Software reuse. *ACM Computing Surveys*, 24(2):131–273, 1992.

[16] Message Passing Interface Forum. Document for a standard message-passing interface (draft), February 1994.

[17] R. Mirchandaney, J. H. Saltz, R. M. Smith, D. M. Nicol, and K. Crowley. Principles of runtime support for parallel processors. In *Proceedings of the 1988 ACM International Conference on Supercomputing*, pages 140–152, 1988.

[18] D. I. Pullin. Direct simulation methods for compressible ideal gas flow. *Journal of Computational Physics*, 34:231, 1980.

[19] A. Skjellum. The Multicomputer Toolbox: Current and future directions. In *Proceedings: Scalable Parallel Libraries Conference*, 1993.

[20] A. Skjellum, S. G. Smith, C. H. Still, A. P. Leung, and M. Morari. The Zipcode message-passing system. In G. C. Fox, editor, *Parallel Computing Works!* Morgan Kaufmann, 1992.

[21] E. F. Van de Velde. Concurrent scientific computing. Draft, California Institute of Technology, 1993. To be published by Springer-Verlag.

[22] D. M. Volpano and R. B. Kieburtz. The templates approach to software reuse. In T. J. Biggerstaff and A. J. Perlis, editors, *Software Reusability*, chapter 9, pages 247–255. ACM Press, Addison Wesley, 1989.