Integrating Task and Data Parallelism with the Group 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 using 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 in 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.

*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.

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 [11] and PVM [17]). 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 *group 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. Such compositions 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 group communication archetype organizes processes into groups and defines group communication between and 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.

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 group communication archetype.

2.1 Climate simulation

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

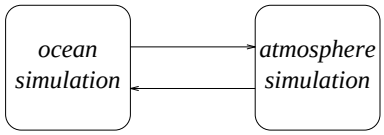

Figure 1: Climate simulation.

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.

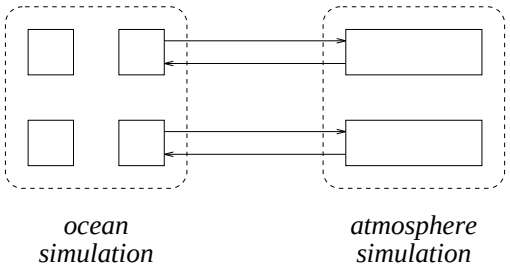

Figure 2: Climate simulation (refinement).

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.

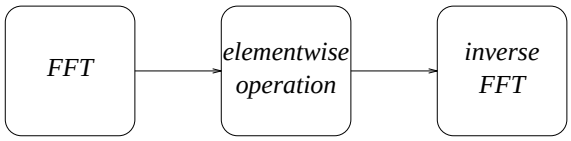

Figure 3: FFT pipeline.

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

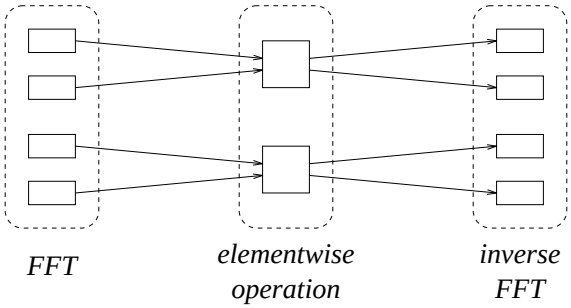

Figure 4: FFT pipeline (refinement).

3 Process structures

The group 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 fo-

cus 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 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.)

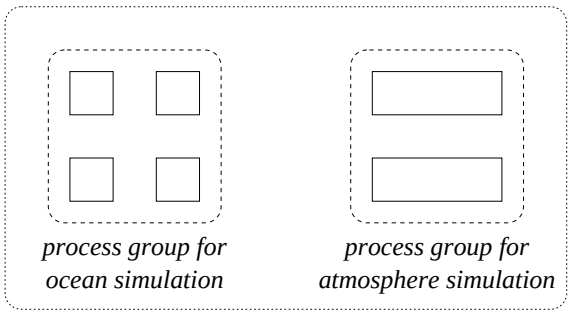

Figure 5: Two-level nesting for climate simulation.

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 by calling a procedure like the following:

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

The input parameters specify the program to be executed by each process (`executable_file_name`); the number of processes in the group (`number_of_processes`); the processors to be used (`number_of_processors` and `array_of_processors`); how processes are to be mapped to processors—e.g., by blocks or cyclically (`processes_to_processors_map`); and how this process group communicates with other process groups (`number_of_sending_ports`, `array_of_sending_ports`, `number_of_receiving_ports`, and `array_of_receiving_ports`, described in §5).

After execution of the procedure, the output parameter (`group_handle`) uniquely identifies this process group.

We combine process groups into networks thus:

```
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, ocean_processors, BLOCK, ....) ;
    create_group(atmosphere, "atmosim", 2,
        2, atmo_processors, BLOCK, ....) ;
    ....
end_network() ;
```

4 Distributed data structures

In the group 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 [12] and HPF [13].

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

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.

The model is flexible in that it allows transmitting all or part of a data structure. 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.

The 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.

We define communication between groups in terms of group sends and group receives. In a group send, the sending group collectively sends all or part 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 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.

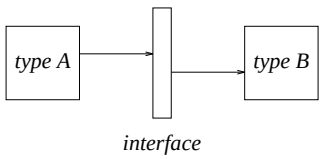

Figure 6: The interoperability problem.

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 simulation example of §2.1.

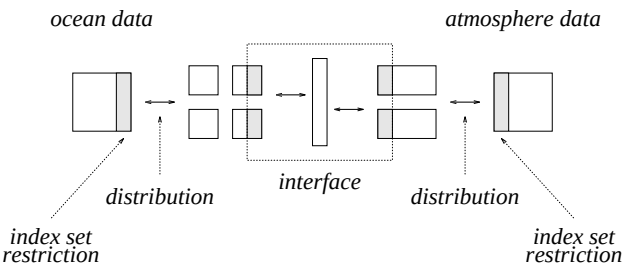

Figure 7: Interoperability for the climate simulation.

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 [11]. 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.

Group-to-group channels are declared at the level of the enclosing nesting; each path establishes two ports, a sending port and a receiving port, which are passed to the process groups. We can create a group-to-group channel by calling a procedure like the following:

```
channel(
    /*in*/  sending_group_handle,
    /*in*/  receiving_group_handle,
    /*in*/  sending_group_distribution,
    /*in*/  sending_group_restriction,
    /*in*/  receiving_group_distribution,
    /*in*/  receiving_group_restriction,
    /*in*/  interface_map,
    /*out*/ sending_port,
    /*out*/ receiving_port
    ) ;
```

The input parameters specify the groups connected by the channel (`sending_group_handle` and `receiving_group_handle`, defined by `create_group` operations), which can be identical; how the data structure to be sent is distributed (`sending_group_distribution`) and what part is to be sent (`sending_group_restriction`); how the data structure into which data is to be received is distributed (`receiving_group_distribution`) and what part is to receive data (`receiving_group_restriction`); and how the data sent is to be mapped into the data received—e.g., normal or transposed (`interface_map`).

After execution of the procedure, the output parameters (`sending_port` and `receiving_port`) represent the ends of the defined channel and can be included in the 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 thus defined, via procedure calls like the following:

```
group_send(
    /*in*/  sending_port,
    /*in*/  data_location
    )
group_receive(
    /*in*/  receiving_port,
    /*out*/ data_location
    )
```

The input parameter (`sending_port` or `receiving_port`) indicates the port to be used—and, implicitly, what parts of the data are to be sent or received. The input/output parameter (`data_location`) references the local section of the distributed data structure to be used for sending or receiving.

The exchange of boundary information in the climate simulation example of §2.1 requires two channels, one in each direction, so the network would be defined thus:

```
begin_network() ;
    create_group(ocean, "oceansim", 4,
        4, ocean_processors, BLOCK,
        1, ocean_send_ports,
        1, ocean_receive_ports) ;
    create_group(atmosphere, "atmosim", 2,
        2, atmo_processors, BLOCK,
        1, atmo_send_ports,
        1, atmo_receive_ports) ;
    channel(ocean, atmosphere,
        BLOCK(2,2), EAST_BOUNDARY(1),
        ROW(2), WEST_GHOST_BOUNDARY(1),
        NORMAL,
        ocean_send_ports[0],
        atmo_receive_ports[0]) ;
    channel(atmosphere, ocean,
        ROW(2), WEST_BOUNDARY(1),
        BLOCK(2,2), EAST_GHOST_BOUNDARY(1),
        NORMAL,
        atmo_send_ports[0],
        ocean_receive_ports[0]) ;
end_network() ;
```

Because of the circular data dependencies (between `create_group` and `channel` operations), procedure calls in a process-network definition are assumed to execute concurrently.

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, there are operations that make sense when applied to arrays but not when applied to records.) Within a group, therefore, the group 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 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 group/collective communication routines that encapsulate the details of both simulated and real parallelism.

Simulating parallelism: A key feature of our approach to the stepwise development of parallel programs is the simulated-parallel version. The simulated-parallel version of the program being developed 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 can be proved to preserve 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 too difficult to be practical. However, for programs based on the group communication archetype, we can simplify the problem by exploiting the common computation/communication structure represented by the archetype.

Because interaction among processes is restricted to group/collective communication operations, each process can be expressed as a sequence of blocks, each representing a local-computation section or a communication operation. Parallel execution can thus be simulated by representing each process by a simulated processes and then, for each block, looping sequentially over all simulated processes. Distributed memory can be simulated by giving each simulated process its own distinct set of program variables.

The archetype provides support for many aspects of this simulation, including code to support looping over simulated processes and simulated-parallel versions of the archetype communication routines. The application-specific parts of the simulated-parallel version are essentially identical to their counterparts in the real-parallel version, and conversion from one to the other is straightforward and preserves correctness.

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.

We can abstract from such computations a specialized version of the group communication archetype, the *spectral-methods archetype*, in which the computation consists of a single process group, and the communication operations are the two redistribution operations—group-to-group communication operations in which the sending and receiving groups are the same.

To use this archetype to develop a parallel program, we can begin with a sequential version of the program, as illustrated by figure 8.

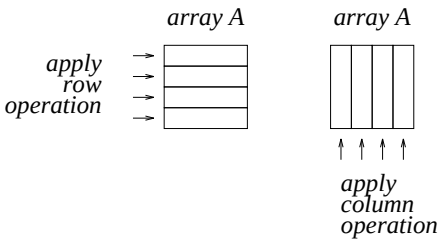

Figure 8: Stepwise development, step 1.

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.

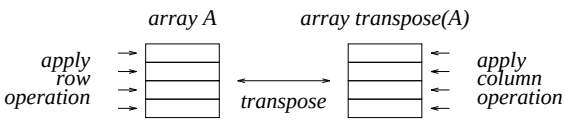

Figure 9: Stepwise development, step 2.

Since row operations operate on all rows independently, they can be performed in parallel; similarly for column operations. Thus, as shown in figure 10, 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.

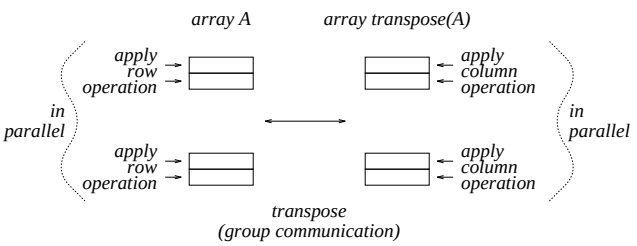

Figure 10: Stepwise development, step 3.

Figure 10 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
    call simulated row-to-column transpose
    for all simulated processes P
        for all rows J in local section P
            perform column operation on row J
        endfor
    endfor
    call simulated column-to-row 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
        call row-to-column transpose
        for all rows J in local section P
            perform column operation on row J
        endfor
        call column-to-row 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. To a large extent, application performance is dependent on the performance of the archetype's communication routines and is thus the responsibility of the archetype developer rather than the application developer. Just as archetype developers can provide implementations of these routines based on different parallel languages and libraries, they can provide implementations tuned for different architectures.

However, the application developer may be responsible for other aspects of performance tuning—for example, choosing the appropriate granularity. If the archetype parameterizes the relevant quantities (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 be used to guide this process.

8 Implementation and applications

In this section, we present several implementation experiments. Each experiment is based on a specialized version of the group communication archetype and consists of developing an archetype implementation (code library and program skeleton) plus one or more applications that make use of it. The programs described here are written in either Fortran M [11] or Fortran with NX [15], but we have also developed archetype-based programs in Fortran with p4 [4], Fortran with PVM [17], and CC++ [6].

8.1 Spectral-methods archetype

In this archetype, described in the example in §7.1, data is a two-dimensional array, and the computation consists of a sequence of row operations and column operations. As described in §7.1, an effective way of parallelizing such computations uses a single process group, data distributed alternately by rows and columns, and two redistribution communication operations.

This archetype has been used to parallelize a spectral code that provides a numerical solution of the three-dimensional Euler equations for incompressible flow with axisymmetry. Periodicity is assumed in the axial direction; the numerical scheme [5] uses a Fourier spectral method in the periodic direction and a fourth-order finite difference method in the radial direction.

Figures 11 and 12 show execution times for a Fortran M implementation of this application on the IBM SP1 and a network of Sun workstations, respectively. As anticipated, performance scales moderately well with the number of processors, up to the point at which granularity is too small to produce an effective ratio of computation to communication.

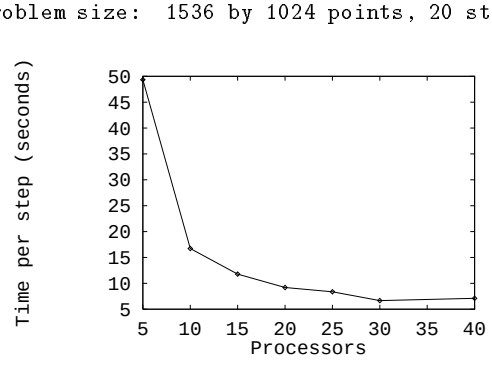

Figure 11: Spectral code execution time (SP1).

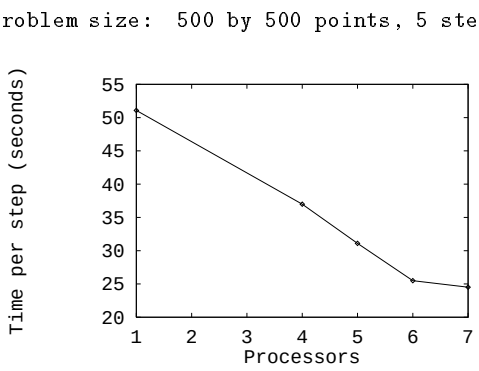

Figure 12: Spectral code execution time (Suns).

8.2 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 using a single process group; the mesh is distributed among processes, and the required communication operations are intragroup (collective) operations—exchange of boundary values with neighboring processes and (for some computations) global reduction.

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 [16]. 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].

In addition, this archetype has been implemented both in Fortran M (for development) and in Fortran with NX (for use on the Intel Delta). The application-specific parts of programs based on the archetype are almost completely independent of which implementation is used, making them very portable.

Further, a simulated-parallel version of this archetype has been implemented (in Fortran 77); this version was used in the parallelization of the electromagnetics application, allowing all development and debugging to be done in the sequential domain.

Figure 13 shows execution times for a Fortran-with-NX implementation of the CFD application running on the Intel Delta. As anticipated, performance scales with the number of processors, up to the point at which granularity is too small to produce an effective ratio of computation to communication.

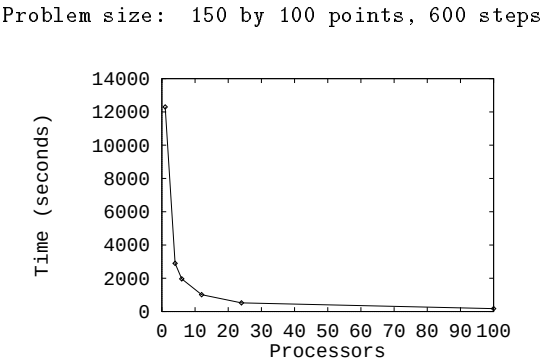

Figure 13: Mesh code execution time (Delta).

8.3 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 independently to each row); performing a column operation (an operation applied independently to each column); performing local calculations on each grid cell; and writing data to a (sequential) output file.

One approach to parallelizing this computation uses four process groups: a group consisting of one host process 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].

Figure 14 shows execution times for a Fortran-plus-NX implementation of this application on the Intel Delta. As anticipated, performance scales with the number of row and/or column processes, up to the point at which granularity is too small to produce an effective ratio of computation to communication.

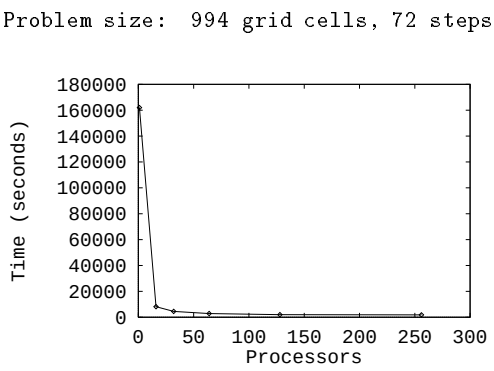

Figure 14: Airshed code execution time (Delta).

8.4 General group communication archetype

We have also implemented a preliminary version of the general group communication archetype described in §3; 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 has not yet been implemented.

9 Related work

A great deal of research has been done on software reuse and templates [3, 9]. 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 [18].

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.

We differ from some earlier work on process groups, such as [8], in our view of process groups as SPMD computations, with collective group-to-group communication operations. We differ from more recent work involving process groups, such as MPI [14], in our emphasis on intergroup rather than intragroup communication. Finally, we differ from the many libraries of collective-communication operations in our emphasis on stepwise refinement, abstraction, and the integration of task and data parallelism.

A more complete discussion of related work appears in an earlier version of this paper [7].

10 Summary

The group 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, and in fact can make the task of converting an application from one parallel language to another almost trivial, enhancing portability. While our focus is on ease of development rather than on optimal performance, properly tuned archetype implementations should make it possible to achieve very good performance.

Much work, however, remains to be done: extending and unifying the implementation experiments described in this paper into a library of archetypes and associated tools; generalizing the ideas presented here about process structures from simple one- and two-level nestings to arbitrary compositions of process groups and networks; and extending the definitions of distributed data structures and group communication to apply to these arbitrary compositions.

Acknowledgments

Many of the ideas here were motivated by the work on compositionality and 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; Peter Carlin, Greg Davis, Sharif Rahman, and Tom Zavisca for their help with implementations; and Adam Rifkin and John Thornley for their comments on 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 Pennsylvania State University, July 1992.

[3] P. Brinch Hansen. Model programs for computational science: A programming methodology for multicomputers. *Concurrency: Practice and Experience*, 5(5):407–423, 1993.

[4] R. M. Butler and E. L. Lusk. Monitors, messages, and clusters — the p4 parallel programming system. *Parallel Computing*, 20(4):547–564, 1994.

[5] C. Canuto. *Spectral Methods in Fluid Dynamics*. Springer Series in Computational Physics. Springer-Verlag, 1988.

[6] 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.

[7] K. M. Chandy, R. Manohar, B. L. Massingill, and D. I. Meiron. Integrating task and data parallelism with the collective communication archetype. Technical Report CS-TR-94-08, California Institute of Technology, 1994.

[8] D. R. Cheriton and W. Zwaenepoel. Distributed process groups in the V kernel. *ACM Transactions on Computer Systems*, 3(2):77–107, 1985.

[9] M. I. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[10] D. Dabdub and J. H. Seinfeld. Air quality modeling on massively parallel computers. *Atmospheric Environment*, 28(9):1679–1687, 1994.

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

[12] 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.

[13] 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).

[14] Message Passing Interface Forum. MPI: A message-passing interface standard, May 1994. Available via anonymous FTP from `ftp.mcs.anl.gov`.

[15] P. Pierce. The NX message-passing interface. *Parallel Computing*, 20(4):463–480, 1994.

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

[17] V. S. Sunderam, G. A. Geist, J. Dongarra, and R. Manchek. The PVM concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–545, 1994.

[18] E. F. Van de Velde. *Concurrent Scientific Computing*. Springer-Verlag, 1994.