Developing Heterogeneous Applications Using Zoom and HeNCE *

Richard Wolski[1], Cosimo Anglano[2], Jennifer Schopf[1] and Francine Berman[1]

[1] Department of Computer Science and Engineering, University of California, San Diego
[2] Dipartimento di Informatica, Universitá di Torino

Abstract

Heterogeneous network computing enables the development of a single complex application using a distributed network of possibly dissimilar machines. While heterogeneous networks promise cost-effective compute cycles, almost no software is available to aid in the design and implementation of their applications. In this paper, we couple the Zoom representation, designed to facilitate development of heterogeneous applications, with the HeNCE graphical language and tool, designed as a representation for an execution model of heterogeneous programs targeted to PVM. The combination of Zoom and HeNCE provides a hierarchical representation which exposes performance issues, and provides a means of automatically translating that representation into code executable on a heterogeneous network of computers.

1 Introduction

Heterogeneous network computing enables the development of a single complex application using a distributed network of possibly dissimilar machines. It is an important and emerging field, drawing concepts from both computer and computational science. While networks of computers can provide large computational capacities, the efficient use of such computing ensembles is not yet well understood. In particular, heterogeneous applications have computation and communication behaviors distinct from those designed for individual parallel or sequential computers.

Scientists and researchers from a variety of disciplines typically participate in the development of heterogeneous applications. Currently, there are few tools available for heterogeneous programming that assist in the design and implementation of these complex applications. In this paper, we outline a methodology through which heterogeneous applications can be designed, specified, and implemented. Applications are first described using the Zoom representation [1]. Zoom captures application structure hierarchically, serving both to facilitate human communication and as an interface to programming tools. We then describe how the Zoom representation of an application can be translated to HeNCE [2], a graphical language for heterogeneous computing. HeNCE provides a rich set of programming primitives and an execution model designed to assist the programmer in developing programs for execution on a networked group of heterogeneous machines [3].

Together, Zoom and HeNCE provide a powerful tool for heterogeneous program development. The Zoom representation of an application is intended to allow scientists and programmers working at various levels of the development hierarchy to communicate and reason about their application. Using Zoom as an interface to the HeNCE graphical language provides a way for such a description to be automatically translated into executable code targeted to a heterogeneous system. That is, Zoom brings hierarchical structure and adds data conversion representations to the execution and control mechanisms of HeNCE.

In the next section we will briefly describe Zoom and HeNCE. Section 3 describes the translation process, Section 4 illustrates some of the differences between the two representations, and in Section 5 we draw conclusions and point to future work.

2 Zoom and HeNCE Descriptions

The Zoom representation is hierarchical, with each level providing more detailed information. Level 1 provides a simple structural description of the application

*This work was supported in part by NSF grant ASC-9301788, NSF grant ASC-9308900, ESPRIT-BRA project No. 7269 "QMIPS", and by the Italian CNR project "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo", grant 91.00879.PF68. E-mail addresses of authors are: {rich, jenny, berman}@cs.ucsd.edu, mino@di.unito.it. Accepted by the Heterogeneous Workshop, IPPS '95.

while more details of the algorithm-machine mapping are given in Level 2. Level 3 adds data conversions and more detail about communication for the purpose of assessing performance trade-offs. While [1] includes a detailed description of the Zoom representation, we review the basic components in the following subsections for completeness.

2.1 Zoom Level 1 Components

The structure level (Level 1) of the Zoom representation depicts an *application* as a linear sequence of *phases*[1]. Each phase circumscribes a graph whose rectangular nodes are *coupling units* and whose edges are dashed lines representing communication between coupling units. Coupling units represent logical components (collections of program tasks) of the application that can potentially communicate across machine boundaries. Communication between program tasks that does not span machine boundaries is not made explicit at Level 1.

A Level 1 representation consists of *phase boundaries* that enclose a portion of the application, *boxes* representing coupling units labeled with their logical component, and *dashed arcs* representing communication between coupling units and to and from phase boundaries. Note that dashed arcs indicate that although communication takes place, the form of the communication is not specified, i.e. they do **not** dictate a precedence relationship. Dashed arcs simply show that at some point in time communication takes place. For example, computation and communication may be overlapped.

Phases may be executed once or repeatedly. Repeated phases of the application are enclosed by sets of double lines, one at the beginning of the repeated phase and one at the end. Phases that are executed once are enclosed by a set of single lines.

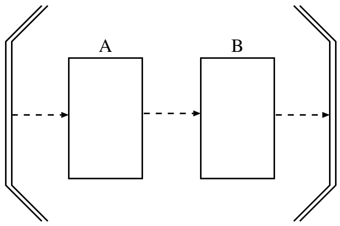

Figure 1: Level 1 Example

Figure 1 depicts the Level 1 Zoom representation for a fictitious application. It consists of two coupling units, A and B, which are linked sequentially and are executed multiple times as a series of iterations.

2.2 Zoom Level 2 Components

The tasks of a heterogeneous application can typically be implemented on multiple machines. Each implementation may use a different algorithm, a different programming paradigm, etc. Under the Zoom model, it is the coupling unit that defines those application components that can have multiple instances. Zoom also provides a way to represent different implementations of the same coupling unit for a single architecture. Two different physics modules targeted to the same machine, for example, may both compute ocean temperature with different degrees of accuracy.

Furthermore, the Level 2 representation differentiates between coupling units. *Rectangular boxes* represent coupling units in which exactly one implementation may execute; *octagonal boxes* represent coupling units in which zero or more implementations may execute. Rectangular or octagonal coupling units contain *ovals* which represent distinct algorithm-machine pairings. Each oval represents a different combination of algorithm, language, programming paradigm, machine, etc.

With this additional information about the implementation of the coupling units, it is also possible to include more information about the communication between them. Level 1 dashed arcs are replaced by *wires* indicating strict communication between all possible implementations of the source and destination coupling units, or *tubes* indicating that some pair of implementations will communicate using an overlapping or pipelined communication.

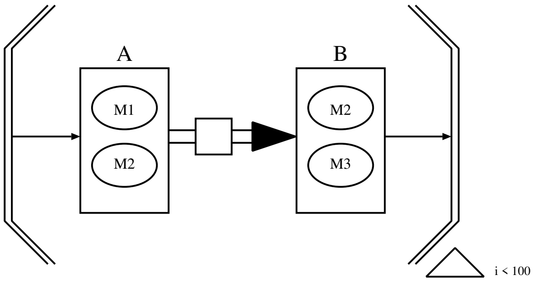

Figure 2: Level 2 Example

In addition, Level 2 includes data conversion information. A *square* intersecting a tube or wire indicates that the type or structure of the data communicated along that edge must be converted for some pair of

[1] The decomposition of heterogeneous applications into phases represents the structure of typical heterogeneous programs and follows [8].

implementations (ovals) in the source and destination coupling units. Finally, the location and criteria for termination of repeated phases is indicated by a *triangle*.

Figure 2 depicts the Level 2 Zoom representation components for the same application as shown in Figure 1. In the figure, coupling unit A has two implementations, one for machine M1 and one for machine M2, and coupling unit B has two implementations for machines M2 and M3. The computation of A and B may be overlapped in a pipeline. Both coupling units are executed iteratively until index variable i is incremented to be greater than or equal to 100. The square shown between the coupling units indicates that some pair of implementations in A and B require a structure conversion.

2.3 Zoom Level 3

At Level 3, more specific information about the application's implementation and resource requirements is depicted. In particular, the communication paradigms, structure conversion requirements, and conversion requirements mandated by different machine data formats (format conversions) are shown for each legal pairing of implementations (Level 2 ovals). Level 3 components contribute to a cost model for assessing performance trade-offs.

In a Level 3 representation, each Level 2 tube or wire is augmented with a set of three matrices showing *connectivity*, *format conversion*, and *structure conversion*. Recall that a Level 2 tube or wire identifies a pair of communicating coupling units (rectangular or octagonal boxes), each of which has one or more implementations (ovals). Every oval in the source box corresponds to a row in each matrix, and every oval in the destination box is associated with a column.

An element (i, j) of the *connectivity matrix* (denoted by a "c" located next to its lower right hand corner) is a pair of integers (x, y), where x is the size of the data item sent across the communication link from oval i in the source coupling unit to oval j in the destination coupling unit, and y is the total amount of data. The fraction x/y represents the granularity of the communication and is less than 1 if the communication is pipelined or overlapped and equal to 1 if the communication is strict. If there is no communication between two ovals, we place $(0, 0)$ as its connectivity matrix entry. Note that using this representation, wires can be thought of as a special case of tubes in which $x = y$.

The *format conversion matrix* is denoted by an "f" near its lower righthand corner. If a pair of ovals require a format conversion, then their corresponding element is marked with S if the conversion is mapped with the source. D indicates that the conversion belongs with the destination, and B signifies that the conversion is done on both ends (as when an intermediate representation is used). If the ovals do not communicate or if there is no format conversion, N is used.

Structure conversions between ovals are denoted S if they are computed with the source oval, and D if they are computed with the destination oval. Non-communicating ovals or ovals with no structure conversion are denoted with N. The structure conversion matrix is marked with "s" next to its lower righthand corner.

Figure 3 shows the connectivity, format conversion, and structure conversion matrices for the example in Figures 1 and 2. The connectivity matrix shows that

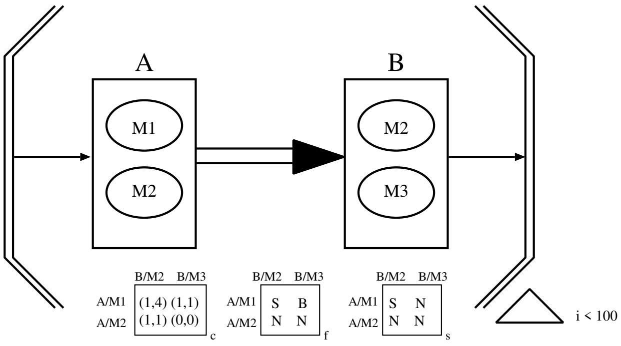

Figure 3: Level 3 Example. Only matrices for the tube are shown.

the implementation for machine M1 of coupling unit A (denoted A/M1) is linked to the implementation of B for machine M2 (B/M2) by a tube. A/M1 communicates with B/M3 via a wire, as does A/M2 when it is coupled with B/M2. Note that the Level 2 tube indicates that some legal pairing may overlap execution, but the connectivity matrix makes the communication relationships between specific implementations explicit.

The format conversion matrix indicates that a conversion is required when A/M1 communicates with B/M2, and that conversion is scheduled with the source on machine M1. A/M1 also requires a format conversion when it communicates with B/M3, but the conversion is split between the source machine and the destination. A/M2 does not require a format conversion when it communicates with B/M2, and no conversion is specified for A/M2 and B/M3 since they do not communicate.

Only a single structure conversion is shown in the example between A/M1 and B/M2. The S indicates that the conversion should be computed by the source machine M1.

2.4 HeNCE

HeNCE was developed by Beguelin, Dongarra, Geist, Manchek, Plank, and Sunderam [2]. Under the HeNCE paradigm, the programmer explicitly specifies parallelism by depicting data and control dependencies in the form of a graph. The HeNCE programming tool provides an editor so that these graphs may be entered directly using a point and click interface. Alternatively, HeNCE programs may be entered textually to allow the use of conventional program editors. Once written and compiled, the HeNCE system maps a program to a user-defined collection of machines. Both the programmer and the system can control the mapping of each program component. PVM (Parallel Virtual Machine) [10] is used to implement communication between program components, and to control execution, and the HeNCE system automatically inserts the necessary PVM primitives into the program at compile time. The user only need supply a valid HeNCE program graph.

We take the following description and Figure 4 from [3] where a more complete discussion of the HeNCE representation and programming tool can be found. Denoted by a circle, each node within a HeNCE program represents a subroutine written in some external programming language, such as Fortran or C. All subroutines must be functional, computing their results based solely on their inputs. A subroutine name, a strongly typed list of parameters, and a collection of source files (one per architecture) are associated with each node.

Data dependencies between nodes and control flow are represented by directed arcs. The source node must execute to completion before the destination node. All data available at the source node is assumed to be available to every descendent node along a connected path from the source to the end of the graph.

2.5 HeNCE Rewrite Rules

HeNCE includes a set of graph rewrite rules that define its execution model. Conditional execution is represented by a bracketed and labeled subgraph. Each conditional subgraph contains a single source and a single sink. The label, located next to the subgraph source, specifies a boolean expression. If the expression evaluates to true, the nodes within the subgraph are executed.

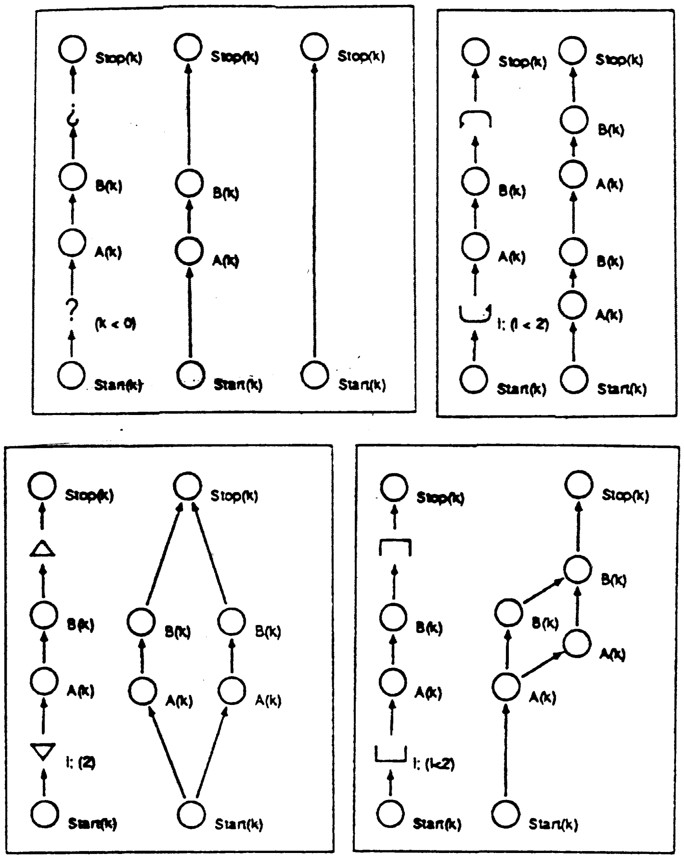

Figure 4: HeNCE Rewrite Rules

Similarly, a HeNCE loop is also represented by a labeled subgraph having a single source and sink. The source is labeled with an index variable, and a boolean expression guarding loop termination.

Parallel sections may be specified within a HeNCE program using a fan subgraph. The subgraph's source is labeled with an integer expression. At runtime, the expression is evaluated to determine how many times the subgraph (having a single sink) should be replicated. HeNCE supports pipelined execution through the pipe subgraph construct. A pipe subgraph (which again has a single source and a single sink) is divided into levels based on data dependencies. Each level is assumed to constitute a stage in the pipe. A boolean expression similar to that used in the loop construct controls how many times the entire pipe will be executed. During iteration i of the pipe, stage k's completion enables both stage $k+1$'s execution, and stage k's execution at iteration $i+1$ (see Figure 4).

2.6 HeNCE Cost Matrix

The programmer specifies mapping information for the entire program through a two-dimensional cost matrix. The rows of the matrix correspond to the different machines available from the heterogeneous system. The columns correspond to the various subroutines (HeNCE nodes) that make up the program. Matrix element *(m,n)* represents the relative non-negative, integer cost of executing subroutine n on machine m. A larger integer value indicates a greater execution cost, but a zero value in element *(m,n)* means that subroutine n will not be executed on machine m. Figure 5b in the next section shows an example HeNCE cost matrix.

3 Translating Zoom to HeNCE

In this section, we describe how a Zoom representation may be translated into a HeNCE program for execution. To do so, we use an enhancement of Calcrust, a heterogeneous application currently under development at JPL [4]. The purpose of Calcrust is to combine USGS survey data, NASA satellite data, and sounding data taken from oil company archives to produce a 3-D rendering of the earth's surface and crust. Two coupling units, each targeted to a single architectural type, make up the actual application. An fft-like data filtering stage, executing on a Cray C90, feeds data to a parallel rendering package executing on an Intel Delta. The filter stage is not overlapped with the rendering stage, and the application does not iterate.

To Calcrust, we add a third explicit smoothing stage between the filter and the renderer, we increase the set of possible target machines, and we hypothesize several possible implementations. The filter may be executed either on an RS6000 workstation or a Cray C90; the smoothing function can be mapped to a CM-5, a C90, or a Paragon; and the renderer may be either executed on a CM-5 or a C90.

3.1 Rectangular Boxes

A Zoom box (and the Level 2 components it contains) corresponds roughly to a HeNCE node. Each subroutine source file from HeNCE corresponds to a Zoom Level 2 oval.

Before we discuss some of the details associated with translating rectangular boxes to their HeNCE equivalents, we need to augment Zoom slightly. Note, that within a rectangular Zoom box, exactly one enclosed oval may be selected for execution. Zoom, however, does not include a rule for making that selection. The simplest (and most prevalent) solution to this problem is to put the burden on the programmer. That is, for most currently existing heterogeneous applications, the programmer *a priori* specifies which algorithm-machine matches are to be executed. In this work, we add to Zoom a programmer-defined set of algorithm-machine matches (Level 2 ovals) that are intended to make up an execution of the application. Only one oval may be selected in each rectangular box, however multiple ovals may be selected from octagonal boxes (see next section). Selected ovals will be distinguished throughout our examples in boldface.

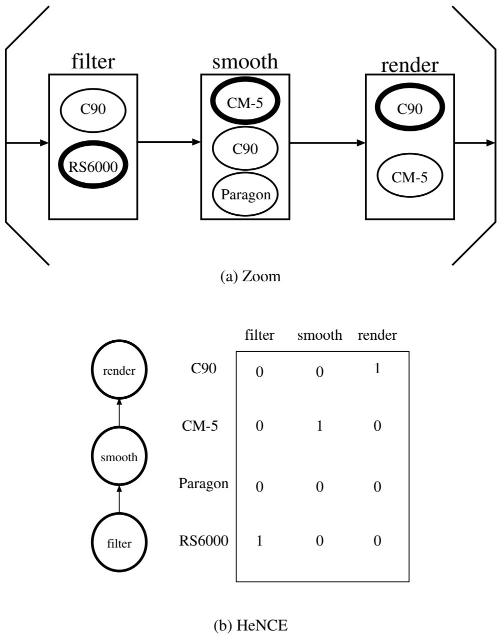

Figure 5: Zoom Level 2 Representation and HeNCE Graph and Cost Matrix.

The selection of a Zoom oval within a box results in the definition of a column in the HeNCE cost matrix attached to that subroutine. For example, consider the example Zoom and HeNCE representations shown in Figure 5. This figure shows the Zoom representation for the application. The boldfaced ovals show that the filter has been mapped to an RS6000, the smoothing function has been assigned to a CM-5, and the renderer to a C90. Equivalently, under HeNCE (Figure 5b) there is a one-to-one correspondence between each Zoom box and a HeNCE graph node. Since exactly one implementation of each Zoom box can be selected, each corresponding column in the HeNCE cost matrix contains a single non-zero element.

3.2 Octagonal Boxes

The translation of a Zoom octagonal box is slightly different. Consider the example program shown in Figure 6. The smoothing function has been assigned

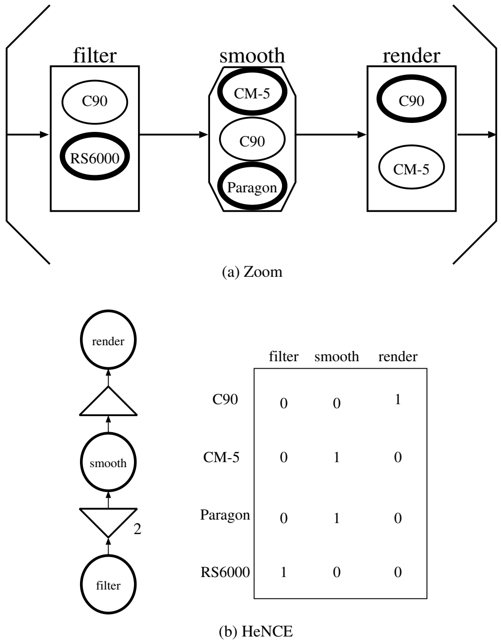

Figure 6: Level 2 Zoom Octagonal Box and HeNCE Graph with Fan construct and Cost Matrix.

to both the CM-5 and the Paragon, as depicted by the octagonal Zoom box and the two highlighted ovals in Figure 6a. The corresponding HeNCE representation and cost matrix are shown in Figure 6b. HeNCE will create two copies of the smooth node as a result of the fan, and then will consult the cost matrix. Since the Paragon and the CM-5 are weighted the same, HeNCE will assign one copy to each.

Care must be taken when performing this translation, however, as HeNCE keeps track of how much load it has assigned to any given machine. If, because of other previously assigned application components, either the CM-5 or the Paragon were heavily loaded, HeNCE might assign both smoothing subroutines to the other more lightly loaded machine. In this example, where there is no parallelism and no overlap, the fan construct is appropriate. For the general case, however, the translation depicted in Figure 7 is applicable.

The second translation is independent of the load assigned to any of the machines in the system. It has the disadvantage, however, that the code implementing the selected ovals must be replicated under

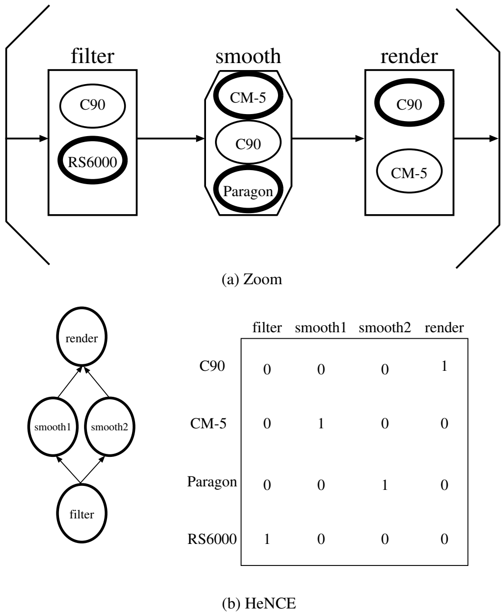

Figure 7: Level 2 Zoom Octagonal Box and HeNCE Graph with Cost Matrix (general case).

two entry points (smooth1 and smooth2 in the figure). Similarly, the cost matrix expands by one column per implementation that is selected.

3.3 Tubes and Pipes

Under Zoom, two dependent coupling units that may have their execution overlapped are connected by a tube. HeNCE associates overlapped computations using a pipe construct, so for simple cases, the translation is one-to-one. Figure 8 depicts an overlapped version of our example application.

In Figure 8, the Zoom boxes joined by tubes translate to HeNCE nodes within a pipe construct. Note that the Zoom tube does not include the notion of a termination condition as does the HeNCE pipe. The reason for this is that the Zoom tube is intended to represent several different forms of overlapped computation. In one form, the producing box generates a complete data structure, call it A and passes to the consuming box. While the consuming box computes using the first A structure, the producing box generates another, hence the producer's and consumer's executions overlap. It is this form of "pipelining" that HeNCE represents and is able to execute. The termination condition dictates when no more A's will be generated and the computations within the pipe will cease to execute.

Frequently, however, the producing computation is

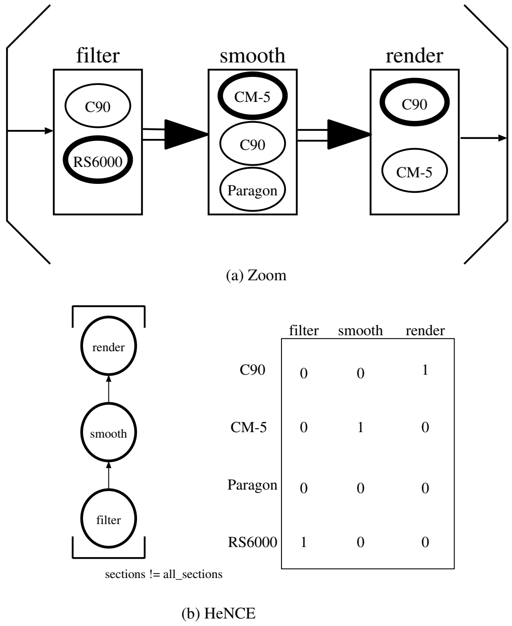

Figure 8: Level 2 Zoom Tube and HeNCE Pipe

generating pieces of a larger data structure and sending them to the consumer one-at-a-time where they are used to compute another aggregate data structure. For example, in the GCM code developed at UCLA [7], the atmospheric model passes "bands" of a grid representing the earth's atmosphere to the ocean model. The ocean code then uses each band to compute its effect on a corresponding region of the ocean. While each band can be thought of as an individual data structure, the logical model is one in which pieces of the atmosphere are sent to compute pieces of the ocean. That is, the goal of each computation is to generate a complete atmosphere grid and a complete ocean grid respectively. Once a complete grid is present in both models, the codes do not terminate, but they do synchronize. Hence the HeNCE notion of termination condition is difficult to apply to the GCM code directly.

We intend the Zoom example shown in Figure 8 to represent this kind of piecewise overlap. The *filter* stage produces part of an image that is smoothed while another part is being filtered. Therefore, in the figure, we depict the termination condition to be "section != all_sections" indicating that the pipe will terminate when all pieces of the image have been processed.

The primary difference between Zoom and HeNCE regarding pipelining is that HeNCE views overlap as a computational characteristic, whereas in Zoom, it is represented as a communication paradigm. Figure 9 more concretely illustrates this difference using another simple example. The box marked *edge find*

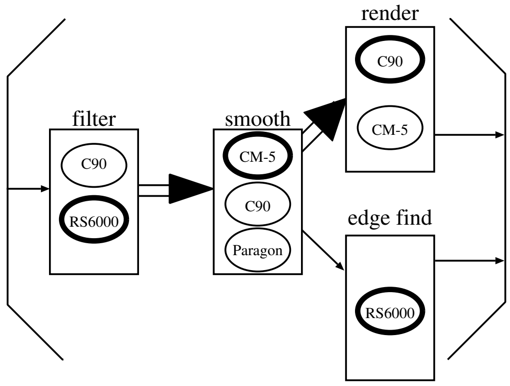

Figure 9: Level 2 Zoom Tube Not Easily Translated to HeNCE Pipe

in the figure represents an edge detection routine only implemented for the RS6000. Note, however that the arc connecting it to the *smooth* box is a wire and not a tube. Since the execution of the *edge find* coupling unit does not overlap that of the *smooth* coupling unit, it must wait for the entire output from *smooth* to begin executing. The renderer, however, can compute using partial results from *smooth*. That is, *smooth* is both part of a pipeline with *render* and part of a strict communication with *edge find* at the same time. Since HeNCE views overlap as a computational rather than a communication characteristic, a subroutine is either part of a pipeline, or it is not – never both. To represent the application depicted in Figure 9 using HeNCE, a collector node would need to be added within the pipe construct to coalesce all of the *smooth* subroutine's output for *edge find*. We don't anticipate that this sort of transformation can be done automatically for the general case, however.

3.4 Structure Conversions

An important execution cost incurred by some heterogeneous applications comes from the need to convert data as it is passed from one machine to another [6]. Floating point numbers are stored differently on Cray computers than on most workstations, for example. Many communication libraries such as PVM [10], Express [5], and XDR [9] will translate one machine's data format to another as the data is moved between machines. These format conversion are represented under Zoom at Level 3 in the f-matrix associated with each wire or edge.

Since HeNCE uses PVM, however, format conversions are automatically performed as part of any communication. Therefore, there is no corresponding Zoom-to-HeNCE translation. However, different routines within an application may also require conversion of higher-level data structures. For example, in the heterogeneous GCM implementation discussed in [7], the atmospheric models use a different grid density than does the ocean model.

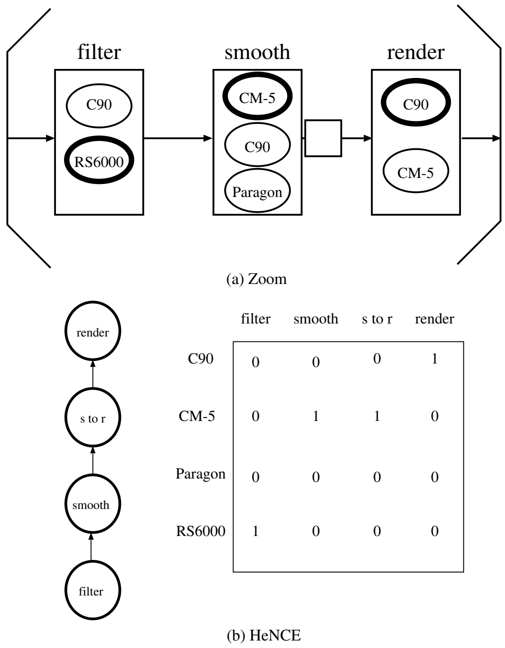

Figure 10: Level 2 Zoom Squares and HeNCE Nodes

In Zoom, the structure conversion computation is represented as a square. The rationale behind representing conversions explicitly is that they contribute to execution cost and, as such, the programmer or the application may wish to assign them individually to different machines. In the GCM example referenced earlier, the interpolation routine may be scheduled either on the same machine as the atmospheric model or on the same machine as the ocean model. The conversion routines, then, can be thought of as a separate (though not completely independent) computations of an application. As such, Zoom squares translate to individual HeNCE nodes, as shown in Figure 10.

The square in the Zoom representation shown in Figure 10a is drawn between the smooth coupling unit and the renderer. Notice that it is situated immediately next to the smooth box indicating that the conversion is intended to be performed on the same machine as the smoothing function. In the HeNCE representation depicted in Figure 10b, a node labeled "s to r" is included in the program graph between the smooth node and the render node. The cost matrix column corresponding to node "s to r" is the same as the column corresponding to smooth to ensure that they are scheduled to the same machine.

3.5 Iterations

Loops, under Zoom, can be classified into two types: those that overlap successive iterations and those that do not. If wires are used to link the interior graph features with the iteration boundaries, then the iterations do not overlap, and the translation from Zoom to a HeNCE loop is relatively straightforward. There is no direct translation for the other case.

There are also two types of termination conditions that must be translated from Zoom to HeNCE. For an indexed loop, the termination condition label from the Zoom triangle transfers directly to the HeNCE termination expression (for example, in Figure 6). Loops governed by an internal termination condition, however, are a bit more complex to translate. Zoom uses a triangle to denote which Level 2 oval is responsible for generating the loop termination conditions. Ultimately, however, the routine responsible for determining loop termination must answer "yes" or "no" at the end of each iteration. To translate such a loop into the HeNCE representation potentially requires that an artificial variable be set to the truth value of this test. HeNCE would then query the variable as part of its execution cycle.

4 General Differences between Zoom and HeNCE

While the functionalities of Zoom and HeNCE are largely complementary, there are several differences that bear further study. In particular, while most applications currently expressible by Zoom can be translated to HeNCE, some cannot. Zoom associates the separable and important features of a heterogeneous application with individual graph elements whenever possible. The goal of the HeNCE representation, on the other hand, is the automatic generation of executable code for the application that is described. Labels attached to HeNCE graph features specify semantic meaning necessary for implementation rather than for other purposes such as execution cost estimation, resource requirements, etc. A general difference, then, is that as a basis for tool design, and as a representation useful to the application scientists operating

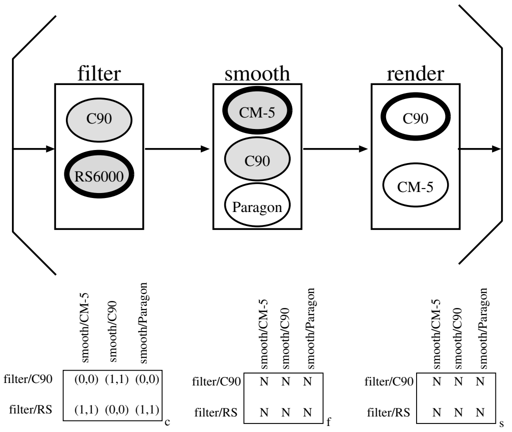

Figure 11: Zoom Compatibility Sets. Only matrices for Filter to Smooth arc are shown.

at various levels of detail, Zoom captures information that does not necessarily translate directly into executable code. Conversely, the HeNCE program representation constitutes a powerful high-level graphical language in which each symbol has a well-defined mapping to a set of executable statements.

4.1 Parameters

One important way in which HeNCE and Zoom differ concerns the association between the various coupled routines in a heterogeneous application and their parameters. Specifically, HeNCE associates a strongly typed parameter list with every node. The analogous association in Zoom would be to assign a parameter list with every rectangular or octagonal box. However, doing so requires that all implementations represented by ovals within a box take exactly the same set of parameters, which may not be the case. Consider the Zoom representation shown in Figure 11 in which the Level 3 matrices are shown for the wire linking the *filter* box with the *smooth* box.

It may be that the RS6000 implementation of the filter and the CM-5 implementation of the smoothing function (shown shaded in the figure) communicate via one set of parameters, while two C90 versions (crosshatched in the figure) share different parameters. The C90 implementations cannot be coupled with either the RS6000 or the CM-5 implementations without a structure conversion. If no such conversion is included as part of the application, the C90 implementations are incompatible with those targeted to the CM-5 and the RS6000 in the figure.

Under HeNCE, all implementations of a given subroutine (each represented by a single HeNCE node) must take the same set of parameters. It might be possible (using duplicated nodes and conditionals) to circumvent this restriction, but in general, the HeNCE representation contains no mapping information. There is no way to change the parameter sets used between subroutines as a function of their mapping.

4.2 Multiple Implementations for a Single Architecture

It may be that the application scientist wishes to include different implementations for the same coupling unit on a single architecture. In our example program, there may be two different rendering packages available for the CM-5, each useful for a different visualization. HeNCE associates a single source file per architecture with each graph node so different implementations for an architecture are specified by different nodes. In addition, the choice between those nodes can only be made at runtime using the HeNCE conditional construct. In contrast, Zoom integrates mapping information into the representation. Each algorithm-machine matching is represented by a separate Level 2 oval. Since the various mapping choices are visible at Level 2 in Zoom, the inclusion of multiple implementations is natural. The HeNCE representation, because it is primarily concerned with execution semantics, does not expose potential mappings in a program graph.

4.3 Mapping

The difference in mapping representation also affects how Zoom and HeNCE might be used in an environment where dynamic scheduling is supported. Currently, HeNCE programs are statically mapped according to the cost matrix. The mapping occurs just before the application is executed, but no dynamic system load information is consulted, nor does the mapping change once execution has been started. In [3], the authors describe the incorporation of dynamic system information as part of future HeNCE development. Similarly, dynamic mapping information currently is not included as part of the Zoom representation. While we plan to investigate Zoom's utility as an interface to a dynamic mapping tool, it too is part of our future work. The difference between Zoom and HeNCE in this regard, however, is that the HeNCE mapping interface is cost-based. As we note in the preceding section, Zoom exposes part of the mapping process to the programmer in the form of application choices in the representation. We believe that both cost-based and user-oriented mapping strategies will be required to implement effective dynamic scheduling techniques.

5 Conclusions and Future Work

In this work, we seek to combine the complementary features of Zoom and HeNCE into an effective specification and development tool for heterogeneous applications. Zoom's hierarchical structure and mapping-oriented representational features make it a natural interface for a variety of heterogeneous software tools. By coupling the execution model implemented as part of the HeNCE representation with a Zoom application description, we provide an application-oriented interface to a heterogeneous execution model. Differences between the design goals of Zoom, and those of HeNCE lead to questions that must be resolved before all applications can be translated. However, we believe these differences to be surmountable.

As part of our future work, we plan to investigate how Zoom and HeNCE can be extended and integrated. In particular, memory and I/O bandwidth constraints frequently impact the design and implementation of heterogeneous applications. Both Zoom and HeNCE should be augmented to further consider resource constraints.

Additionally, we plan to develop other tools using Zoom as an interface. With a more complete cost model, and dynamic system information, Zoom can be used as part of performance estimation and scheduling tools for heterogeneous applications.

While heterogeneous networks promise cost effective compute cycles, little useful software is available to aid in the design and implementation of targeted applications. The combination of Zoom and HeNCE provides a hierarchical representation which exposes performance issues, and a means of automatically translating that representation into code executable on heterogeneous networks of computers.

Acknowledgements

We would like to thank the applications researchers with whom we spoke for sharing their experiences in developing heterogeneous programs. In addition, we would like to thank the Heterogeneous Reading Group at UCSD for their helpful suggestions.

References

[1] Anglano, C., Schopf, J., Wolski, R., and Berman, F. Zoom: A hierarchical representation for heterogeneous applications. submitted to the Journal of Parallel and Distributed Computing.

[2] Beguelin, A., Dongarra, J., Geist, G., Manchek, R., Plank, J., and Sunderam, V. Hence: A user's guide version 1.2. Tech. Rep. CS-92-157, University of Tennessee, February 1992.

[3] Beguelin, A., Dongarra, J., Geist, G., Manchek, R., and Sunderam, V. Graphical development tools for network-based concurrent supercomputing. In *Proceedings of Supercomputing '91* (1991), IEEE Press, pp. 435–44.

[4] Bergman, L., Braun, H.-W., Chinoy, B., Kolawa, A., Kuppermann, A., Lyster, P., Mechoso, C. R., Messina, P., Morrison, J., Stanfill, D., St.John, W., and Tenbrick, S. Casa gigabit testbed : 1993 annual report; a testbed for distributed computing. Tech. Rep. CCSF-33, Caltech Concurrent Supercomputing Facilities, May 1993.

[5] Flower, J., and Kowala, A. Express is not just a message passing system: Current and furure directions in express. *Parallel Computing 20* (1994), 597–614.

[6] Khokhar, A., Prasanna, V. K., Shaaban, M., and Wang, C.-L. Heterogeneous Supercomputing: Problems and Issues. In *Proceedings of the 1992 Heterogeneous Workshop* (1992), IEEE CS Press.

[7] Mechoso, C. R., Ma, C.-C., Farrara, J. D., Spahr, J. A., and Moore, R. W. Parallelization and distribution of a coupled atmosphere-ocean general circulation model. *Monthly Weather Review 121*, 7 (July 1993), 2062–76.

[8] Snyder, L. *Phase Abstractions for Portable and Scalable Parallel Programming.* MIT Press, 1990.

[9] Sun Microsystems Inc. *Network Programming Guide – External Data Representation Standard: Protocol Specification*, 1990.

[10] Sunderam, V. S., Geist, G. A., Dongarra, J., and Manchek, R. The pvm concurrent computing system: evolution, experiences, and trends. *Parallel Computing 20*, 4 (April 1994), 531–45.