

Making a Dataparallel Language Portable for Massively Parallel Array Computers

Martin C. Herbordt*

Department of Electrical and Computer Engineering
University of Houston
Houston, TX 77204-4793

James H. Burrill[†] and Charles C. Weems[‡]

Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Abstract: A key goal in language design is to simultaneously achieve portability and efficiency. Achieving a general solution to this problem is quite difficult: virtually all attempts have emphasized one or the other requirement by restricting either the architecture domain, the application domain, or both. In this study we present i) a framework that explains why meeting these requirements simultaneously is so difficult, and ii) our approach, which, though it may not be the final word on this subject, implements a new set of trade-offs that may come closer to a balanced solution than has been previously achieved. Our solution includes an easy to use language based on the dataparallel programmer's model, a compiler that hides as many machine variations as possible, a library with emulations of constructs that map directly to hardware on some but not all machines, and a library with different versions of those critical application functions for which a single algorithm is not optimal across all hardware configurations. We have found the programmer cost for the application and architecture domains considered here to be quite reasonable.

1 Introduction

It has been apparent since the late 1950's that SIMD arrays are well suited for image-to-image computing [37]. By the early 1960's it had been discovered that they were also excellent for extracting features from images: for example, the Illiac III was built specifically to extract tracks in bubble chamber images [26]. Several decades of further study have found SIMD arrays useful in an increasingly wide variety of computer

vision applications including region and line segmentation, line finding through perceptual organization, hough transforms, graph matching, extraction of structure from motion, and in solving many problems from computational geometry. For a sample of this work see [31, 25, 28, 18] and the references in [14]. Even so, SIMD designs have never converged to a particular set of attributes, at least not nearly to the same extent as have serial computers or even shared memory multiprocessors. Many fundamental design questions including granularity, datapath design, memory hierarchy, and, especially, routing network remain open.

The languages supporting these SIMD arrays have been equally varied, usually being directly related to the particular machine for which they were written. At first, high-level languages were a luxury so all coding was done in native assembly language. The next step was machine-specific high-level languages such as Glypnir for the Illiac IV [23]. Later languages were more general but still retained machine dependent restrictions, typically limits on array size related to physical machine size [30]. More recently, languages have been written for families of machines having a variable number of PEs; for example, codes written in C* were required to be able to run on all members of the CM2/200 family whether they have 1K or 256K processors [36]. Even with this progress, however, portability has received relatively little attention in SIMD languages in comparison to languages intended for serial and even MIMD machines: efficiency, then ease of coding, have always been the primary concerns.

What we consider here are the issues involved with creating a single language for the class of SIMD arrays, especially for coding computer vision applications, without giving up efficiency. We will see that as a bonus we will have created a language that is also useful for running these applications on multiprocessor and multicomputer systems. The requirements of such a language are as follows: It must

*NetAd: herbordt@uh.edu. Supported in part by an IBM Fellowship and by the NSF through CAREER award #9702483.

[†]NetAd: burrill@cs.umass.edu. Supported in part by DARPA Project#DAAL02-91-C-0051.

[‡]NetAd: weems@cs.umass.edu. Supported in part by DARPA under contract DACA76-89-C-0016, monitored by the U.S. Army Engineer Topographic Laboratory; under contract DAAL02-91-K-0047, monitored by the U.S. Army Harry Diamond Laboratory

1. be easy to use and lend itself to a natural programming style,
2. hide architectural features that are not essential to the programmer's model,
3. allow access to architectural features on particular machines that are essential in using those machines to their fullest potential,
4. discourage the programmer from using constructs that will result in poor performance, and
5. be source-code portable.

Although careful language design and a good compiler can get us a ways towards achieving these goals, it is also quite clear that as long as there are major variations in SIMD architectures, especially with respect to local indexing and communication network, these requirements are fundamentally incompatible. The essential issue is the inherent difficulty of achieving both portability and consistent efficiency across a variety of parallel architectures.

Achieving a general solution to this problem is quite difficult: virtually all attempts have emphasized one or another of the requirements by restricting either the architecture domain, the application domain, or both. For example, Apply achieves portability at the cost of restricting the applications to be image-image [13], while C*, though quite general, is still not suitable for running on arrays with reconfigurable broadcast buses. In fact, just the premise of a language for computer vision tasks on SIMD arrays restricts both architecture and application domains.

In this study we do not present a complete solution to these problems; what we *do* present i) a framework that explains why meeting these requirements simultaneously is so difficult (and in the process explains why previous attempts fall short), and ii) our approach, which, though it is not a complete success, implements a new set of trade-offs that may come closer to a balanced solution than has been previously achieved.

Our approach contains the following components:

- Careful language design that presents the programmer with an easy to use dataparallel programmer's model,
- Use of compilers to hide as many machine variations as possible (e.g. array size) while retaining functionality such as automatic code optimization,
- Including constructs in the language to support all significant and unique hardware features in the class of SIMD arrays plus the capability to add others relatively easily,

- Providing a library with emulations of constructs that map directly to hardware features on some but not all machines, and
- Providing a library with different versions of those critical software functions for which a single algorithm is not optimal across all hardware configurations.

We discuss these components in turn, but begin by describing the architecture domain.

2 Architectural Assumptions

In this section we specify what we mean by SIMD array architecture. The **required** features are as follows:

- Controller and array of PEs
- Controller is a general purpose computer
- Controller broadcasts instructions and scalars to the array
- Array feeds back OR to controller
- I/O system to transfer data to/from array
- PEs have at least enough register file to hold three 32 bit operands or can efficiently emulate that capability
- PEs have sufficient ALUs and, if necessary, scratch space, to perform standard arithmetic and logical operations including floating point
- PEs know their own unique ID and whether they are on an edge or not
- PEs have activity control to conditionally execute instructions
- The array has communication capability sufficient to implement, or emulate efficiently, neighbor communication
- PEs have access to enough memory (on and off chip) to be able to store data required by significant computer vision codes and a mechanism to transfer data on and off chip.

The **optional** features, that is, the features that may or may not be in the machine and still have it be a part of the class, are as follows:

- Local indexing within each PE
- Array feeds back COUNT to controller
- Permutation routing network
- Permutation routing network with combining for reductions

- Segmented scan network
- Reconfigurable broadcast network

These lists were created to match the architecture space spanned by machines such as the MasPar MP1/2 [3], the Thinking Machines CM2/200 [35], the Cambridge Systems DAP [27], the MCNC Blitzen [4], the UMass/Hughes CAAPP [39], the Current Technology MM32k [11], the MIT Abacus [5], and numerous other similar machines. Considering first the list of required hardware features, there are three items where we are somewhat stricter than the common definition of SIMD arrays: the register file size, the memory size, and the mesh routing capability. The exact criteria for their inclusion is discussed in a later section, for now we simply state that we are attempting to define a class of computers that are no more different from one another than the class of serial machines; i.e. a class of machines for whose members compilers could be expected to generate similarly efficient object code from any appropriate input program.

The minimum register file size of 96 bits is derived from the register transfer language model: we assume a load-store machine where the register instructions have the form (OPCODE,out,in1,in2) and operands in1, in2, and out can have up to 32 bits. These tuples can be efficiently emulated on machines with only 64 bits of register file such as the Abacus, or on machines such as the CM2 and DAP with no register file at all, but which are designed to run as memory-memory machines.

The memory size and access capability is necessary, again, because of the load-store model. Machines where the data flows primarily from PE to PE, (e.g. the SPLASH [2]) rather than primarily between memory and datapath of each PE are also applicable here.

The mesh routing capability is required due to the prevalence of window-based algorithms in low-level vision. However, due to granularity and chip I/O considerations, many SIMD processors have only 1-D connections [1, 9, 10, 20, 21, 22]. Where these machines can efficiently emulate mesh processors, they are included, where they have primarily a systolic or pipelined programmer's model they are not.

The list of optional features is not as critical and only the capability of executing multiple threads simultaneously is explicitly disallowed. We will include in the discussion section with the implications of adding features to either list.

Another note: the features listed above are not sufficient to specify all architectures on which the codes discussed here can run efficiently. If a machine *does* have additional features such as local control (i.e. it is a MIMD machine) the codes are still likely to run

efficiently. However, there is no guarantee that features on such machines can be used to their potential by codes written in our language. In the terminology of this article we say that requirement 3 has not been met for that architecture/application combination.

3 First Order Solution: Supporting Required Features

We discuss creating a portable, efficient, high-level language for the class of machines specified by the required features presented above. We do this by presenting a the appropriate subset of the language ICL as an example [6]. The operators and methods are summarized in Table 1.

ICL uses C++ as a base, adding a class called Plane to represent data stored in a slice across PE memories. There are Plane types corresponding to all the standard C types except double and long. Planes can be operated upon using the same arithmetic and logical operators as are available for the scalar data types (int, float, etc.). Arithmetic and logical Plane instructions denote element-element computations. No restriction is placed on the size of the Planes, although they must be two dimensional (this is currently being generalized). Plane sizes do not need to be specified until run time, although more efficient code is generated if they are known at compile time. Plane elements can be permuted with NEWS nearest neighbor shifts and summarized with the ANY (global OR) method. Writing to a subset of elements can be restricted with the Select Active method. Various ID and Edge methods return Planes wherein each element has a value corresponding to its ID or whether it is on a Plane edge, respectively. See Figure 1 for an example of an ICL procedure that uses some of these constructs.

Mapping these ICL language constructs to the required set of SIMD array hardware features is straightforward. ICL is a dataparallel language and as such presents a single thread of control: this is executed on the host or controller. Instructions involving Planes are broadcast to the PE array. All scalar instructions are executed by the host while Plane instructions are executed by the PE array. In expressions with mixed scalar/Plane operands, scalars are broadcast by the controller to all PEs and used as immediates in PE instructions. Other language constructs, such as OR, NEWS, Select Active, etc. map as expected to the corresponding hardware features. Since the Plane size is not restricted, mapping Planes to physical PE arrays can involve storing several Plane elements in each PE's memory. In that case, instructions involving Planes do

Language Construct	Comment
<u>Dyadic Operators</u>	
+, -, *, /, %, &, , ^, <<, >>	
<u>Monadic Operators</u>	
-, ~, ++, --	
<u>Assignment Operators</u>	
=, +=, -=, *=, /=, %=, &=, =, <<=, >>=, ^=	
<u>Assignment Methods</u>	
Convert	cast PlaneType to another
Resize	change size of Plane
<u>Conditional Operators</u>	
==, !=, <, <=, >, >=	
<u>Type Cast Methods</u>	
BitPlane, CharPlane, UCharPlane, ShortPlane, UShortPlane, IntPlane, UIntPlane, FloatPlane	
<u>Bit Manipulation Methods</u>	
Insert	move <i>i</i> BitPlanes starting at <i>j</i> in input to the <i>i</i> BitPlanes starting at <i>k</i> in output
Bit	return the <i>i</i> th BitPlane
<u>Read or Alter PE Status Methods</u>	
Edge, NEdge, SEdge, WEdge, EEdge	return edge status in BitPlane
RowIndex, ColIndex	return row/col index in ShortPlane
Index	return index in IntPlane
Activity	set activity to BitPlane input
<u>Inter-PE Communication Methods</u>	
North, South, East, West	2D mesh interPE communication
<u>Array/Controller Interaction Methods</u>	
Any Access	return scalar OR of input BitPlane
ToArray	converts a Plane to a scalar array
FromArray	converts a scalar array to a Plane
<u>Plane I/O Methods</u>	
Read, Write	input or output a Plane
Display	send plane to display device

Table 1: Shown is the part of the ICL virtual machine that corresponds to the required hardware feature set.

```

gradient(IntPlane Image, FloatPlane Magnitude,
         FloatPlane Orient)
{
  IntPlane Xsobel(Image.Size()), Ysobel(Image.Size());
  IntPlane Temp(Image.Size());
  IntPlane Square(IntPlane input);
  FloatPlane Sqrt(FloatPlane input);
  FloatPlane Arctan(FloatPlane Y, FloatPlane X);

  Temp = Image + (Image.East() + Image.West())/2;
  Ysobel = Temp.North() - Temp.South();
  Temp = Image + (Image.North() + Image.South())/2;
  Xsobel = Temp.East() - Temp.West();
  Magnitude = Sqrt(FloatPlane(Square(Xsobel)
                               + Square(Ysobel)));
  Orient = Arctan(FloatPlane(Ysobel),FloatPlane(Xsobel));
}

```

Figure 1: Shown is an ICL function that uses constructs mapping to the required feature set to compute the gradient magnitude and orientation of an image.

not map 1-1 to PE array instructions, but rather a level of emulation must be added; this is discussed below. Extending ICL to implement new language constructs simply requires writing additional C++ methods.

Using a high level language such as ICL which hides much of the physical processor from the programmer satisfies requirements 1 and 2 stated in the introduction. How it does so without giving up a great deal of efficiency is discussed in the next two sections.

4 Second Order Solution: Supporting Optional Features

The third requirement of a portable/efficient language is that it allow access to architectural features that are essential in using a machine to its potential. In order to meet this requirement, ICL has a number of additional methods that operate on Planes: these perform permutations, reductions, scans, and broadcasts. These methods map directly to: permutation networks such as those in the Connection Machine [35] and MasPar MP1/2 [3], combining/scanning networks as in the Connection Machine, and broadcast networks as in the Polymorphic Torus [24], Clip-4 [8], DAP [19], and CAAPP [39]. Local indexing, as in the MasPar and Blitzen [4], is supported by indexing an array of Planes with an index Plane to return a Plane. Extending ICL to support more hardware features simply requires adding the appropriate C++ methods. See Table 2 for a summary of the language constructs that support the optional hardware feature set.

Language Construct	Comment
<hr/>	
<u>Index Operator</u> [Plane]	local address autonomy
<u>Feedback Method</u>	
Count	return scalar count of input BitPlane
<u>Read or Alter PE Status Method</u>	
Coterie	set reconfigurable broadcast bus switches according to CharPlane input
<u>InterPE Communication Methods</u>	
Route	one-to-one interPE communication
RouteOP	many-to-one combining interPE communication
RegionBroadcast	one-to-many transfers, only within regions
RegionSelectOP	return BitPlane of all min or max values in each region
RegRouteOP	same as RouteOP but destination must be within region
ScanOP, ScanRowOP,ScanColOP	scans and segments scans

Table 2: Shown is the part of the ICL virtual machine that corresponds to the optional hardware feature set.

The fifth requirement is that programs be source code portable. In other words a program written, say, for the Connection Machine, should also run on the DAP. We achieve this ‘second order’ portability through emulation: programs for a particular target machine that include language constructs intended to support an optional hardware feature not present on that target machine have those constructs replaced by the appropriate emulation routines. These emulation routines use constructs that map to features that *are* present on the target machine. For communication operations, the language constructs which require emulation on various architectures are shown in Table 3.

The emulations mentioned here have been written for ICL: we call them, collectively, the *Operator Emulation Library*. For local indexing, the emulation routine is trivial, for the communication functions they are more complex. For example, permutation routing and reduction on meshes and meshes with broadcast are implemented with the neighbor connections and by having each PE emulate a switch [15]. Region operations on machines with reduction networks are implemented with repeated segmented scans [25].

The operator emulation library addresses the basic portability problem stated in requirement 5: programs written in ICL will run on any machine in the domain. Programs written specifically for a particu-

Network	MPA Communication Instruction			
	Permutation Route	RegRoute (non-unif. reduction)	Scan (Segmented)	Region Broadcast
Mesh	need emul.	need emul.	need emul.	need emul.
Mesh + Broadcast	need emul.	need emul.	need emul.	direct support
Mesh + Circuit Switched	direct support	need emul.	need emul.	need emul.
Mesh + Packet Switched	direct support	direct support	direct support	need emul.

Table 3: Table shows which communication instruction/network combinations are supported entirely by the hardware architecture and which must be emulated through a combination of software and hardware.

lar processor will do so very efficiently, within the skill of the programmer and the normal coding efficiency of a standard compiler. The problem that is *not* solved is requirement 4, which is that the programmer is discouraged from using constructs that result in poor performance. This issue is discussed in later sections.

If new optional features appear within our domain, e.g. a novel routing network, further emulation routines will need to be written. In the past, however, writing these routines has proved to be far less time consuming than what is typically spent writing a compiler back-end.

5 ICL Compiler Implementation

There were several major issues in implementing an ICL compiler for the CAAPP; compilers for other machines in the class are analogous:

- integration of the two threads of execution, those of the controller and the array,
- code optimization,
- virtual PE emulation,
- conformability, i.e. checking Plane variables for matching dimensions.

Three ICL compilers have been implemented for the CAAPP. One version is a straight-forward implementation using C++ methods to generate instructions to be sent to the CAAPP. The second version augments the first with a run-time CAAPP register allocation procedure to minimize loads and stores. The third version uses a modified Gnu compiler to perform

code optimization and compile time register allocation. This last version is now briefly described; see [7] for details and a comparison of run-time versus compile time register allocation.

The first two issues are addressed by using a Gnu C++ compiler as a base and modifying it so that it treats Plane variables just as it does scalars such as ints and floats. This was done by i) adding logic so that Plane types are treated analogously to those standard in C, ii) defining the necessary operations on those types, and iii) modifying the code generator to send the appropriate PE instructions to the PE array when those Plane operations are encountered. The result is that ICL takes advantage of much of the optimization code provided by the Gnu C++ compiler. One example is that the compiler allocates Planes to PE registers using the same optimizing register allocation procedure used for scalar variables. Another example is that common subexpression elimination is extended to include expressions involving Planes.

Since the Gnu compiler has no concept of virtualization, support was added for that as well. Planes larger than the physical array are partitioned into *tiles*. Most Plane operations require that a physical instruction be executed for each tile, although feedback and communication instructions require physical instructions across tiles. The compiler maximizes performance by scheduling as many physical instructions per tile as possible before swapping in the next one. This is done by detecting the multi-tile instructions and using them as ‘barriers.’

Perhaps the biggest problem to be overcome was *conformability*: two Planes are conformable if they have the same dimensions. Operations may only be performed on conformable Planes. The first two ICL compilers generate code which checks conformability at run-time resulting in significant controller overhead. The third compiler checks conformability at compile time through pattern matching on the abstract syntax tree. This does not completely eliminate all run-time checking, however. Since ICL does not require the programmer to hardwire Plane size, the information is sometimes simply not available to the compiler.

The third ICL compiler has allowed us to create a language based on C++, without non-standard extensions, and without sacrificing the optimizations and high-quality code generation expected of a native compiler.

6 Portability, Efficiency, and Type Architectures

This section follows from arguments presented by Larry Snyder in his classic paper [33].

Creating an executable image from an application specification requires at least three transformations: i) the specification is formulated into a single or a series of algorithms, ii) the algorithms are encoded in a programming language, and iii) the programming language is compiled into executable code. For serial machines the steps are usually independent: when thinking about how to do a sort, a programmer considers quick-sort or merge-sort independently of whether to code it in Lisp or Fortran and, usually, independently of whether the target machine is a Sun or an HP.

We borrow terminology from Snyder by calling any set of machines where we can largely ignore differences in hardware during the algorithm development phase a *type architecture*. Putting it another way, a type architecture encompasses a set of architectures that are related closely enough so that their compilers can do a similarly good job producing executable code for a particular application, given no other information about the application besides the code itself. For example, machines that vary only in number of registers or pipeline depth are probably members of the same type architecture. In fact, one could make a good case that most serial architectures are members of the same type architecture: this is why the SPEC benchmark can be based on program codes, rather than task specifications as is common in vision benchmarks [29, 31, 40]. Similarly, for SIMD arrays, machines that differ only in PE ALU complexity or array size are also probably in the same type architecture.

On the other hand, massively parallel computers with significantly different routing networks are very likely to be members of *different* type architectures. For example, different topologies have different optimal data movement algorithms (see e.g. [34]). While this alone is not enough to place machines into different type architectures—data movement constructs are likely to be supplied by the compiler through routines such as those in our operator emulation library—this principle extends to algorithms for innumerable *problems* involving communication. One need only to consider the myriad papers with titles such as “Optimal Algorithms for \mathcal{X} on the \mathcal{Y} Network” to verify this statement.

Accordingly, SIMD machines with our required feature set are in the same type architecture. Also, a set of SIMD machines is part of the same type architecture if they contain the required feature set, have

similar router networks, and all either have or do not have local indexing capability. If any one of these factors changes, however, then membership in the same type architecture is no longer assured.

As a concrete example, assume that the low level parts of the IU benchmark are to be coded as efficiently as possible in ICL for the CM2 and the CAAPP. The pertinent information here is that the IU benchmark specification contains a convex hull and connected components labeling. To produce the efficient codes, the programmer does not need to know about the number of PEs on each machine, the type of floating point support, or the amount of on-chip storage. Granted this information might be helpful, but no more so than it would be to a programmer of a serial computer trying to maximize performance by bypassing a compiler. The difference in router network, however, is critical: while the CM2 has a routing-combining network, the CAAPP has a reconfigurable broadcast mesh. Both of these networks are directly accessible through ICL language constructs. The difference in networks, however, results in fundamentally different codes being optimal: while the convex hull for the CM2 code uses a Jarvis March, the that for the CAAPP is based on the Graham Scan. The connected components labeling similarly requires different algorithms. Because the algorithm selection differed due to architectural differences, we say that the CAAPP and the CM2 are not members of the same type architecture.

From the previous example it is also apparent that it makes sense for membership in a type architecture to depend on the application set. If the tasks for which two target architectures are to be used do not require different algorithms, then those machines are effectively in the same type architecture for that set of tasks.

In the introduction, we stated as a goal a language that was both portable and efficient for all machines in the class of SIMD arrays, including those with both required and a mix of optional features. We now consider several language design alternatives with respect to the requirements in the introduction and the discussion on type architectures. For all of these alternatives, we assume that emulation routines of the kind discussed in the previous section are available so that requirement 5 is not an issue.

1) *Create a compiler which can spot a task in the application code that is using a suboptimal algorithm and create or select the appropriate algorithm for the target architecture.*

This is the ideal solution but, unfortunately, it is still far beyond current compiler technology. Research con-

tinues in this area, however. Here all machines belong to the same type architecture.

2) *Create a language which includes only constructs supporting a single type architecture where that type architecture is the intersection of the features in the set of target architectures.*

Given the set of target architectures in this study, this choice results in a language that supports only the required features. Programs will still run on machines with optional features—they just will not be able to use them. Although not intended specifically for this set of processors, an example of such a language is Apply which does not contain general communication primitives such as permutations. This choice violates requirement 3 that all important hardware features be accessible. Here a language supporting one type architecture is being compiled to machines in another.

3) *Create a language that only includes constructs supporting a single type architecture where that type architecture includes some subset of optional features.*

This is what virtually all current SIMD languages do. Since some hardware features on some machines are not accessible, this choice violates requirement 3 as above. It also violates requirement 4 that the programmer be discouraged from using constructs not well supported on the target machine. Again, a language supporting one type architecture is being compiled to machines in another.

4) *Create a language that includes constructs supporting all the optional hardware features.*

This is what ICL does. This choice also violates requirement 4. Here the language supports multiple type architectures.

5) *Create a language that includes constructs supporting features that are not in the set of either required or optional features.*

Examples of this would be control parallel languages, which assume local PE control, and also Fortran 90 which has some constructs that do not map well onto any existing communication network. This choice violates requirement 4. Again, the language supports multiple type architectures, but this time some are not even physically realizable.

6) *Restrict the set of applications to those that run optimally on machines with the minimal set of features.*

This is what some image processing languages do. The result is similar to option 2). For example, if the only operations to be done are convolutions, it is possible to be both portable and efficient: machines with, say, complex router networks do not need them and so it

does not matter whether they are supported by constructs in the language or not. Restricting the set of applications, in and of itself, is not bad if that is what is intended: in fact it is something to be taken advantage of. It does mean, however, that machines where features are going unused are perhaps not being used to their fullest potential. Also, computer vision tasks often *can* take advantage of complex features such as sophisticated router networks so this restriction is artificial. Here all machines are effectively in the same type architecture with respect to the application set.

As we stated earlier, the problem lies with simultaneously achieving requirements 3 and 4. Violating either 3 or 4 can cause a substantial slowdown over code written in a language for the appropriate type architecture as we will now show.

7 Dealing with Type Architecture Mismatches

The solutions in the previous section are all less than ideal because of type architecture mismatches. When precisely does a type architecture mismatch cause a problem? Whenever a task has different optimal algorithms for different type architectures. It does not take a long search through the literature to find dozens if not hundreds of examples: just comparing some articles about computational geometry on meshes and RMeshes will yield many such examples.

What is the consequence of running the wrong algorithm? We have run some experiments and found that they can be dramatic. One example is with the task of labeling connected components. The algorithm used by the CM-2 uses either pointer jumping or segmented-grid-scan based algorithms [25], while that on the CAAPP uses multi-associative leader election via region broadcast [18]. We coded these algorithms in ICL and ran them each on both CM-2 and CAAPP machine models using ENPASSANT [16]. In the ‘mismatch’ cases, networks needed to be emulated (the router network on the one hand and the broadcast network on the other). The slowdown resulting from using the incorrect algorithm on the CM2 model was a factor of 13.5. The slowdown of using the incorrect algorithm on the CAAPP model was a factor of 81.

Clearly such mismatches are unacceptable. The way to deal with them, in lieu of a compiler smart enough to do the work for us, is to create separate versions of these offending tasks for each type architecture that requires one. If the number of tasks and type architectures is large, then the creation of such an application function library (AFL) would be prohibitive,

leaving the near-term possibility achieving simultaneous portability and efficiency in doubt.

But before abandoning hope, we must first determine whether such dramatic cases are common, and if so, whether they are always the same task repeated over and over, or many different tasks. To do this we examined a set of SIMD codes produced by the UMass VISIONS and IUA labs over several years and which has been used in conjunction with ENPASSANT for architecture evaluation [17]. We found very few such instances: besides the connected components algorithm just described, the only others were a Hough transform, a convex hull, and right-angle detector. The first of these was in a structure from motion code, the latter two were in the IU benchmark. What we did find was that the majority of cycles were spent executing relatively simple though extremely compute intensive functions. This conclusion has been reinforced by our architecture studies: although we have often found very large differences in performance depending on presence or absence of particular features, this has generally not been because the slower machine needed to do a task differently. Rather, it was because it just could not do it any faster.

We find these last results promising. What they signify is that there is reason to hope that the number of tasks that must definitely be recoded, at least for computer vision applications, will be small. The current contents of our AFL is three versions each of exactly the four tasks mentioned.

8 Discussion

Changing the Hardware Feature Sets

Expanding the set of required features reduces the size of the OEL and AFL since there are fewer variations and therefore fewer type architectures. Contracting the set of required features increases the size of the libraries since there are more optional variations. For example, assume that NEWS connections are optional, but that 1D connections are still required. This would increase the size of the OEL as now NEWS communication must be emulated for 1D machines. It would also increase the size of the AFL in so far as, say, different algorithms are required for window-based operations on 1D arrays as for 2D.

Adding optional features may or may not increase the size of the libraries. For example, adding support for 8-way connectivity was accomplished in a few hours since the 4-way emulation of 8-way communication is trivial. Adding an entirely new communication scheme **S**, however, would require that i) communication constructs be added to ICL to support **S**, ii) all the other

communication operations in ICL be emulated, if necessary, for **S**, iii) emulations be added for *other* communication schemes to emulate the constructs added to support **S**, and iv) new application functions be added for tasks where **S** calls for use of a different algorithm.

We do not believe that emergence of a new scheme **S** is likely, however. We have found that beyond nearest neighbor connections, the only crucial differences in networks are whether or not they efficiently support the standard routing functions of permutation, reduction, scan, and broadcast.

Summary

The approach we have taken includes language, operator emulation, and application function libraries. Together, they meet all five of the requirements stated in the introduction.

The cost, besides a certain inelegance, is the programming involved to add operator emulations and application functions to the libraries as new machines are added to the class of SIMD arrays. Ideally the operator emulation libraries would come with the compilers supplied with each machine, much the way math and graphics libraries are supplied now. Depending on the universality of the application in question, the vendor may or may not be obligated to supply an AFL as well.

In the ENPASSANT project where the goal is to model many machines in this class, we have written our own OEL and AFL (as described above). So far, this programming cost has been on the order of months, so a rough estimate for the cost of adding an essential new feature is likely to be in the neighborhood of weeks. This is much less than the time it takes to create a compiler back end.

Other Approaches

The problem described here is well known. Often solutions fall into one of two diametrically opposed camps: those which imply that parallel processing will converge to a single type architecture and those which just want to get the best performance possible out of a machine. Those in the first camp include the initial versions of the type architecture work described here and the BSP model proposed by Valiant [38]. More recent versions of each have recognized that a solution based entirely on one model with no regard to architecture may not always be ideal and might require, e.g., optimized libraries on different parallel machines [12]. Supporters in the latter camp include many of the actual programmers of parallel processors who are used to acquiring, and using, detailed knowledge of the target architecture to maximize performance.

Currently, it appears that parallel programming has converged to three basic models: dataparallel, using languages such as Fortran 90 (similar to the over-

all approach discussed here); explicit message passing, sometimes using PVM or MPI; and multi-threading using, perhaps, Posix. Vendors of multiprocessors are expected to support all three. The trend, however, is for fewer cycles to be executed from user codes and more from third party applications. In effect these are application function libraries.

Interestingly, an early discussion of this issue [32] proposes that type architectures (idealized machines) will emerge corresponding to a small set of parallel computation approaches. These approaches include processors with local memory and reconfigurable topology, synchronous and asynchronous shared memory, as well as a dataflow machine, and an associative processor. It was also stated that these approaches overlap, are not necessarily complete, but also that if the total number was not small, then the correct salient feature sets probably had not been identified. We believe that while for large-grained machines this may certainly be true, for fine-grained machine the interPE communication network currently cannot be ignored. Naturally, if a clear network choice emerges, this situation will change.

Status and Future Work

We have implemented this approach for three different environments:

- A compiler for the CAAPP
- A compiler that produces C code
- A compiler that creates virtual machine code for use by ENPASSANT in architecture evaluation experiments.

The next task is to create a multithreaded version to run dataparallel applications on multiprocessors.

References

- [1] Adaptive Solutions, Inc. *CNAPS Data Book*. Beaverton, OR 97006, 1995.
- [2] Arnold, J. M., Buell, D. A., and Davis, E. G. Splash 2. In *Proc. of the 4th ACM Symposium on Parallel Algorithms and Architectures* (1992), pp. 316–322.
- [3] Blank, T. The MasPar MP-1 architecture. In *Proc. 35th IEEE Comp. Conf.* (1990), pp. 20–24.
- [4] Blevins, D. W., Davis, E. W., Heaton, R. A., and Reif, J. H. Blitzen: A highly integrated massively parallel machine. *JPDC 8* (1990), 150–160.
- [5] Bolotski, M., Simon, T., Vieri, C., Amirtharajah, R., and Knight, Jr., T. F. A 1024 processor 8 ns SIMD array. In *Proc. 16th Conf. on Adv. Res. in VLSI* (1995).
- [6] Burrill, J. H. *The Class Library for the IUA: Tutorial*. AAI, Inc., Amherst, MA 01003, 1992.

- [7] Burrill, J. H. *Using the GNU C++ Compiler to Generate Code for a Massively Parallel SIMD Processor*. AAI, Inc., Amherst, MA 01003, 1993.
- [8] Fountain, T. J. CLIP4: a progress report. In *Languages and Architectures for Image Processing*, S. L. M. J. B. Duff, Ed. Academic Press, Boston, MA, 1981.
- [9] Fountain, T. J., Matthews, K. N., and Duff, M. J. B. The CLIP7A image processor. *IEEE Transactions on PAMI PAMI-10*, 3 (1988), 310–319.
- [10] Fujita, Y., Yamashita, N., and Okazaki, S. A 64 parallel integrated memory array processor and a 30 gips real-time vision system. In *Proc. of CAMP* (1995), pp. 242–249.
- [11] Glover, M. A., and Miller III, W. T. *A SIMD Parallel Processor Based on DRAM*. Current Technology, Inc., Durham, NH, 1996.
- [12] Goudreau, M., Lang, K., Rao, S., Suel, T., and Tsantilas, T. Towards efficiency and portability: Programming with the bsp model. In *Proc. of the 8th ACM Symposium on Parallel Algorithms and Architectures* (1996), pp. 1–13.
- [13] Hamey, L. G. C., Webb, J. A., and Wu, I. An architecture independent programming language for low-level vision. *Computer Vision, Graphics, and Image Processing* 48 (1989), 246–264.
- [14] Herbordt, M. C. *The Evaluation of Massively Parallel Array Architectures*. PhD thesis, Dept. of Comp. Sci., U. of Mass. (also TR95-07), 1994.
- [15] Herbordt, M. C., Corbett, J. C., Spalding, J., and Weems, C. C. Practical algorithms for online routing on fixed and reconfigurable meshes. *J. Par. Dist. Comp.* 20, 3 (1994), 341–356.
- [16] Herbordt, M. C., and Weems, C. C. An environment for evaluating architectures for spatially mapped computation: System architecture and initial results. In *Proc. of CAMP* (1993).
- [17] Herbordt, M. C., and Weems, C. C. Towards the empirical design of massively parallel arrays for spatially mapped applications. In *Proc. of CAMP* (1995), pp. 59–66.
- [18] Herbordt, M. C., Weems, C. C., and Scudder, M. J. Non-uniform region processing on SIMD arrays using the coterie network. *Machine Vision and Applications* 5, 2 (1992), 105–125.
- [19] Hunt, D. J. The ICL DAP and its application to image processing. In *Languages and Architectures for Image Processing*, S. L. M. J. B. Duff, Ed. Academic Press, London, 1981.
- [20] Johannesson, M. *SIMD Architectures for Range and Radar Imaging*. PhD thesis, Department of Electrical Engineering; Linköping University; Linköping, Sweden, 1995.
- [21] Jonker, P. P. *Morphological Image Processing: Architecture and VLSI Design*. Kluwer, The Netherlands, 1992.
- [22] Krikelis, A. Computer vision applications with the Associative String Processor. *J. of Parallel and Distributed Computing* 13 (1991), 170–184.
- [23] Lawrie, D. H., Layman, T., Baer, D., and Randal, J. M. Glypnir—a programming language for the Illiac IV. *CACM* 18, 5 (1975), 157–164.
- [24] Li, H., and Maresca, M. The Polymorphic-Torus Architecture for computer vision. *IEEE Trans. on PAMI PAMI-11*, 3 (1989), 233–243.
- [25] Little, J. J., Bletloch, G. E., and Cass, T. A. Algorithmic techniques for computer vision on a fine-grained parallel machine. *IEEE Trans. on PAMI PAMI-11*, 3 (1989), 244–257.
- [26] McCormick, B. T. The Illinois Pattern Recognition Computer – ILLIAC III. *IEEE Trans. on Electronic Computers C-12*, 12 (1963), 791–813.
- [27] Parkinson, D., and Jesshope, C. R. The AMT DAP 500. In *Proc. IEEE Computer Society Conf.* (1988).
- [28] Prasanna Kumar, V. K., and Reisis, D. Image computations on meshes with multiple broadcast. *IEEE Trans. on PAMI PAMI-11*, 11 (1989), 1194–1202.
- [29] Preston, K. The Abington Cross benchmark survey. *IEEE Computer* 22, 7 (1989), 9–18.
- [30] Reeves, A. P. Parallel Pascal. In *The Massively Parallel Processor*, J. L. Potter, Ed. MIT Press, Cambridge, MA, 1985.
- [31] Rosenfeld, A. A report on the DARPA Image Understanding Architectures Workshop. In *Proc. Image Understanding Workshop* (1987), pp. 298–301.
- [32] Segall, Z., and Snyder, L., Editors. Report of the NSF/CMU workshop on performance efficient parallel programming. *Performance Evaluation Review* 15, 2 (1987), 16–31.
- [33] Snyder, L. Type architectures, shared memory, and the corollary of modest potential. *Annual Review of Computer Science* 1 (1986), 289–317.
- [34] Stout, Q. Supporting divide-and-conquer algorithms for image processing. *JPDC* 4 (1987), 95–115.
- [35] Thinking Machines Corporation. *Connection Machine Model: CM-2 Technical Summary*. Cambridge, MA, 1987.
- [36] Thinking Machines Corporation. *C* Programming Guide*. Cambridge, MA, 1990.
- [37] Unger, S. H. A computer oriented toward spatial problems. *Proc. of the IRE* 47 (1958), 1744–1750.
- [38] Valiant, L. G. A bridging model for parallel computation. *CACM* 33, 8 (1990), 103–111.
- [39] Weems, C. C., Levitan, S. P., Hanson, A. R., Riseman, E. M., Nash, J. G., and Shu, D. B. The Image Understanding Architecture. *IJCV* 2, 3 (1989).
- [40] Weems, C. C., Riseman, E. M., Hanson, A. R., and Rosenfeld, A. The DARPA image understanding benchmark for parallel computers. *JPDC* 11 (1991).