Achieving Portable Task and Data Parallelism on Parallel Signal Processing Architectures

Hank Hoffmann, Eddie Rutledge, Jim Daly, Glenn Schrader, Jan Matlis, and Patrick Richardson
MIT Lincoln Laboratory

Abstract

The demands of today's signal processing community place a number of difficult requirements on the software engineers who develop signal processing applications. Two of the foremost of these requirements are portability and scalability. MIT Lincoln Laboratory has been working on a software library to facilitate parallel signal processing which specifically addresses these issues.

To meet these requirements, the library has been designed to separate the job of implementing a signal-processing algorithm from the job of mapping that algorithm to a parallel architecture. The person who is responsible for coding the signal processing algorithm is referred to as the "application developer," while the person who is responsible for determining the best way to map the application onto the hardware is referred to as the "mapper."

This decoupling of the mapping concerns from the tasks and data structures which are being mapped is the essential ingredient of a portable and scalable library. Using this development scheme, application developers only need to do their jobs once. When the application needs to be ported to a new machine, or new hardware becomes available, the mapper simply needs to change the file which stores the maps for all mappable objects in the application. Then the new file can be read into the existing application as before, and no changes need to be made to the application code.

Because mapping is so essential, one of the fundamental objects in our library's design space is a *Map*. *Maps* describe where and how different tasks and data structures are distributed. In order to ensure that the library can handle the variety of COTS signal processing hardware that exists in today's market, maps are built to handle distributing both tasks and data over heterogeneous and/or hierarchical hardware architectures.

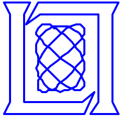
There are several classes in our library that can be mapped. Collectively, these data types are referred to as *Mappable Types*, and they include both tasks and data structures. Application developers write their signal processing code in terms of these mappable types. Separately, the mapper is responsible for developing the maps that will be used by the application developer's mappable types. These maps are stored in a file and read into the application at run-time. When each mappable type is instantiated, it simply looks at its map and builds itself accordingly.

Thus, this library allows applications to be ported to any hardware architecture that can be described by a map (provided that architecture supports the library code), and it allows applications to be scaled easily from a single processor to a number of different processors, all without changing the application code.

In this paper we present an overview of the library we have been developing here at MIT Lincoln Laboratory, which goes into depth on what we consider a *map* to be, how *maps* can handle heterogeneous, hierarchical signal-processing architectures, and specifically how we map tasks and data objects to these architectures.

To illustrate these principles further, we present an example of a small application that implements a digital filtering algorithm, which has both task and data parallel components. We show this algorithm as implemented using our library and demonstrate that the only changes that need to be made to port this implementation from a network of Solaris workstations to an embedded platform involve changes to a text file which contains information on the maps, but do not involve changes to the code which implements the algorithm. To further illustrate the power of our library, we show that the portability and scalability of the

application written using our library is significantly enhanced compared to a similar implementation of the same algorithm written using existing standards like VSIPL and MPI.



Achieving Portable Task and Data Parallelism on Parallel Signal Processing Architectures

Hank Hoffmann

Eddie Rutledge

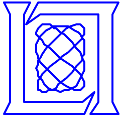
Jim Daly

Glenn Schrader

Jan Matlis

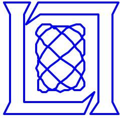
Patrick Richardson

*This work is sponsored by the US Navy, under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and not necessarily endorsed by the United States Air Force.



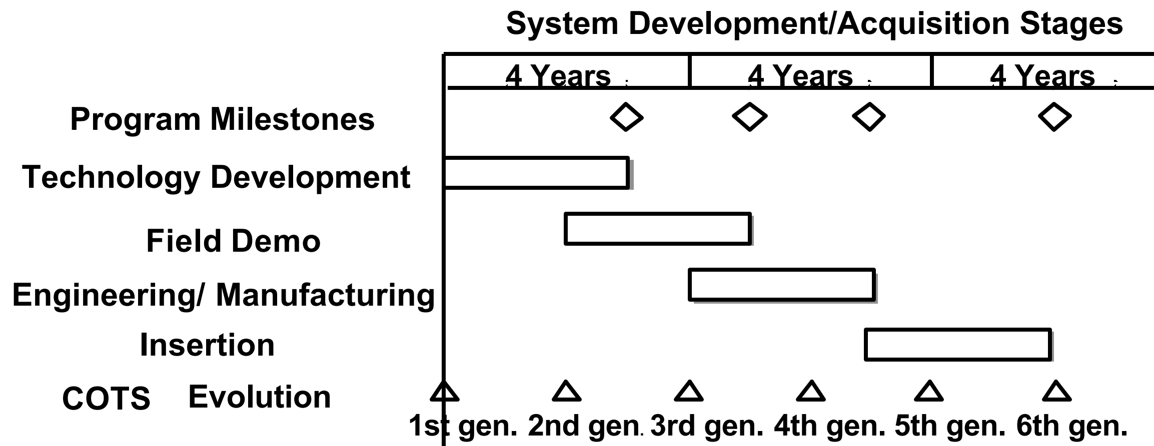
Overview

- Motivation - why write portable software?
- Philosophy
  - how to achieve portability
  - how to measure portability
- Overview of Software Library
- Example signal processing application
- Conclusion

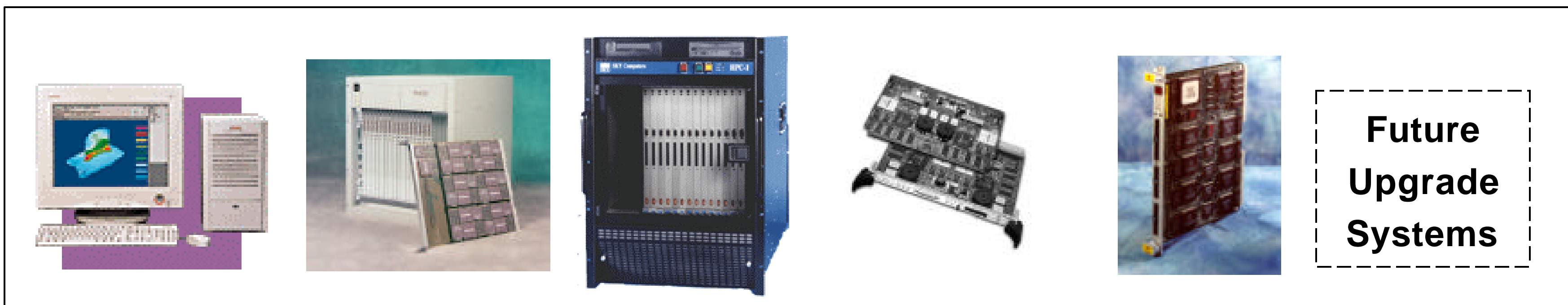


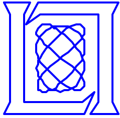
Motivation

- **Take Advantage of New Processor Technology**
  - Portable software enables rapid COTS insertion and technology refresh



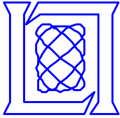
- **Interoperability**
  - larger choice of platforms available





Current Standards for Parallel Coding

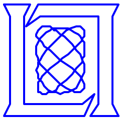
- **Industry standards (e.g. VSIPL, MPI) represent a significant improvement over coding with vendor-specific libraries**
  - **None of the work detailed in this presentation would be possible without the groundwork laid by standards such as VSIPL and MPI**
- **However, current industry standards still do not provide enough support to write truly portable parallel applications**
- **How can we build even more portable systems that work in parallel?**



Characteristics of Portable Software

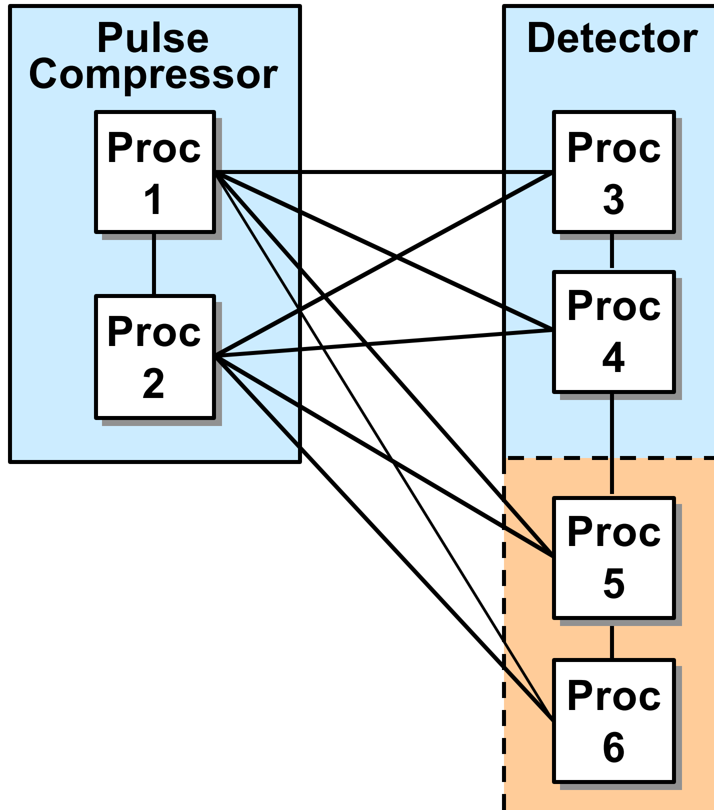
Portable software maintains functionality and performance with minimal code changes

	Single Processor	Parallel Processor
Functionality	• Compile and run on new platform	• Compile and run on new platform • scale to new processor set • handle new communication network
Performance	• Preserve performance (e.g. FFTW) • Take advantage of processor specific traits (e.g. L1/L2/L3 cache vector processing, etc.)	• Handle everything for single processor case • Load balancing across processors • Exploit algorithm parallelism



Writing Parallel Code Using Current Standards

Algorithm & Mapping

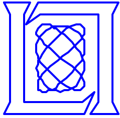


Code

```
while(!done)
{
  if ( rank()==1 || rank()==2 )
        pulse compress ();
  else if ( rank()==3 || rank()==4 )
        detect();
}
```

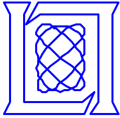
```
while(!done)
{
    if ( rank()==1 || rank()==2 )
        pulse compress();
   else if ( rank()==3 || rank()==4) ||
            rank()==5 || rank==6 )
        detect();
}
```

• We need the ability to abstract parallelism away from the code, and to treat distributed objects as a single unit



Overview

- Motivation - why write portable software?
- Philosophy
  - how to achieve portability
  - how to measure portability
- Overview of Software Library
- Example signal processing application
- Conclusion



Philosophy

Separate the job of writing a parallel application from the job of assigning hardware to that application

"Application Developer"

- Converts algorithm into code

```
while( !done )
{
    pulseCompress();
    detect();
}
```

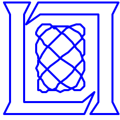
- Writes code once
- Easier to code, because only concerned with mathematics, not distribution

"Mapper"

- Maps code to hardware
- Creates new mappings when code is scaled or ported

PC Map: Proc 0, Proc 1

DET Map: Proc 2, Proc 3

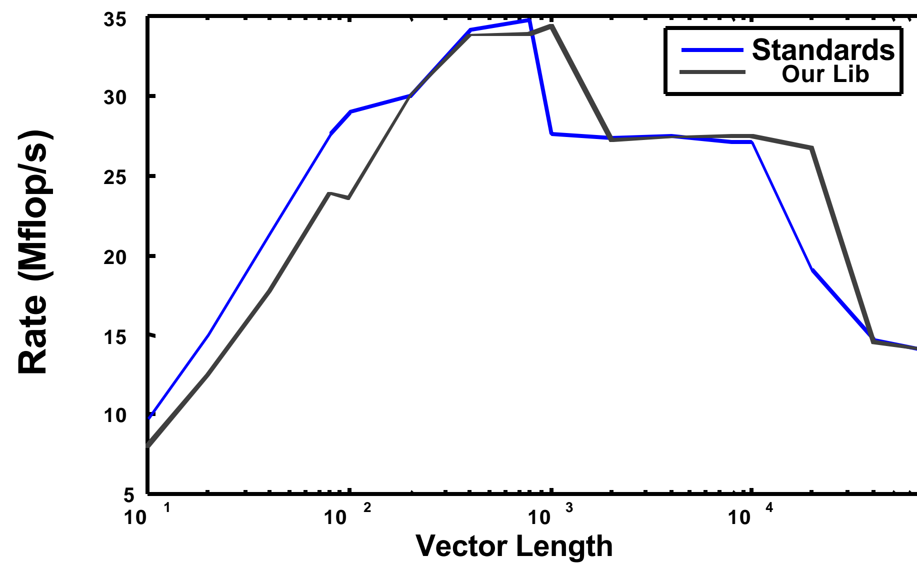


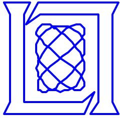
Measuring Success

- **Code Complexity:**
  Number of lines of application code that have to be changed to port or scale

```
if( rank() == 0 )
{
        // ...
}
```

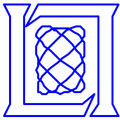
- **Performance:**
  Must preserve the performance of a similar application built on lower-level libraries





Overview

- Motivation - why write portable software?
- Philosophy
  - how to achieve portability
  - how to measure portability
- Overview of Software Library
- Example signal processing application
- Conclusion



A New Parallel Signal Processing Library

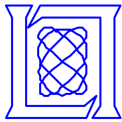
Combining the best of existing standards and STAPL into a new library

- STAPL: Space-Time Adaptive Processing Library

Standards → Interface
C++ → Abstraction
STAPL → Parallel Constructs
→ A New Library for PSP

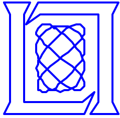
Benefits
- Portability
- Performance
- Fully Parallel
- Familiar Interface
- Increased Productivity

000921-hch-11
HCH 4/27/01



Overview of Principal Library Constructs

	Class	Description	Parallelism
Signal Processing & Control	Matrix/Vector	Used to perform matrix/vector algebra, providing VSIPL functionality except that data and operations may be distributed	Data
	Computation	Used to perform more complicated signal/image processing functions (as per VSIPL) on matrices/vectors (E.G. FFT)	Both
	Task	Supports algorithm decomposition into potentially distributed signal processing tasks	Task/control
	Conduit	Provides distributed data flow between tasks (transports matrices/vectors)	Task/control
Mapping	Map	Specifies how Tasks, Matrices/Vectors, and Computations are distributed on processor	Both
	Atlas	Container class for Maps	



PVL Concepts

- **Tasks** provide "control scope" for computation
- **Conduits** transport data

A=B+C;

E=FFT(D);

C B A D FFT E

Task I

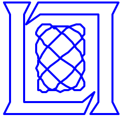
Conduit

Task II

- **Grid** (part of a Map) provides machine abstraction

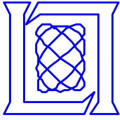
Processor

- Each distributed object has a **MAP** consisting of:
  - **Grid** (binding to physical machine)
  - **Distribution** (of object over Grid)
- Maps provide <u>portability</u> and <u>performance</u>



Overview

- Motivation - why write portable software?
- Philosophy
  - how to achieve portability
  - how to measure portability
- Overview of Software Library
- Example signal processing application
- Conclusion



Example of a Task and Data Parallel Application

Signal Processing algorithm with 3 steps

Digital Input

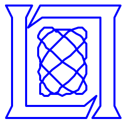
- generates a 52 channel by 768 range matrix

Low Pass Filter

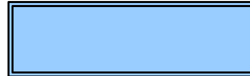
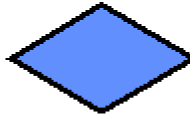
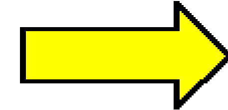
- receive 52 x 768 matrix
- Apply coarse filter
- 2:1 decimation
- Apply fine filter

Beamformer and Detector

- receive 52 x 384 matrix
- form beams
- apply detection template
- store results



Mapping Parallelism in the Algorithm to Library Constructs

- Tasks - 
- Computations - 
- Matrices -
- Conduits - 

Digital Input

52 x 768

Low Pass Filtering

52 x 768

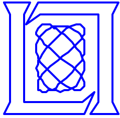
FFT

52 x 384

Beamforming and Detection

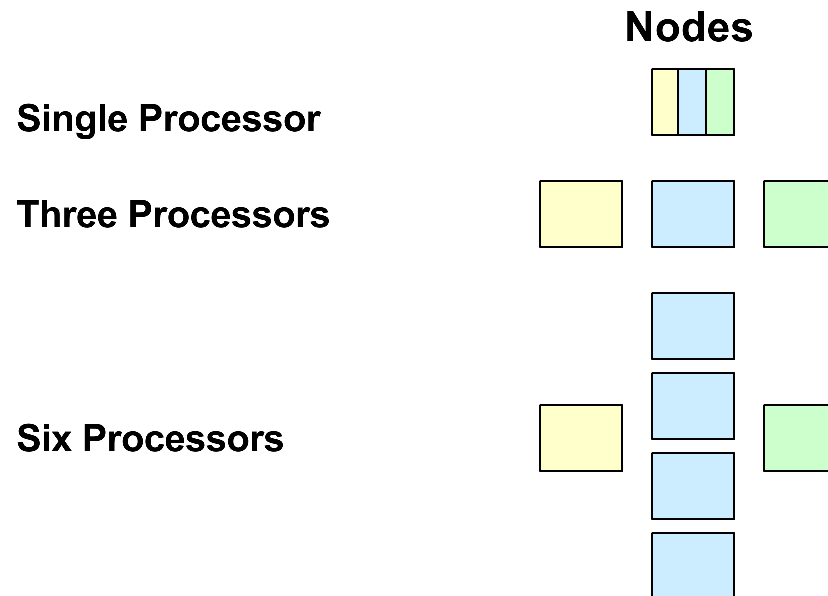
Matrix Mult.

52 x 384

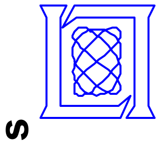


Implementing the Algorithm

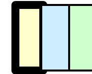
- Examine Implementations of the algorithm using our library and VSIP/MPI
- Distributions:



- Compare Lines of Code for the two different implementations on each mapping

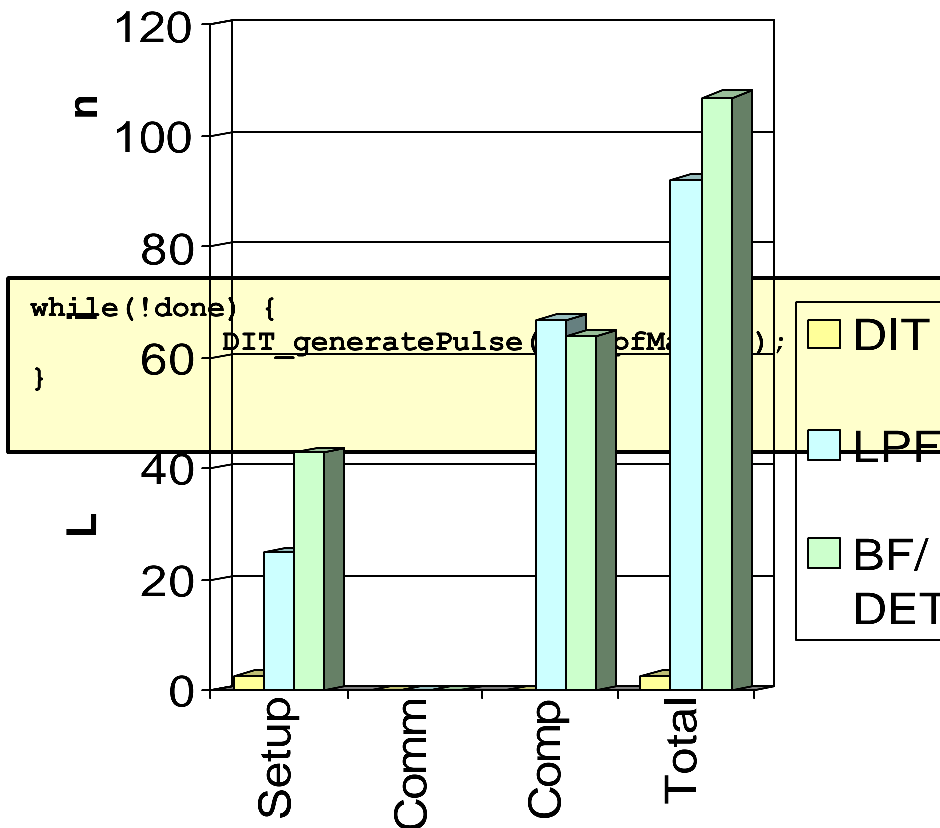


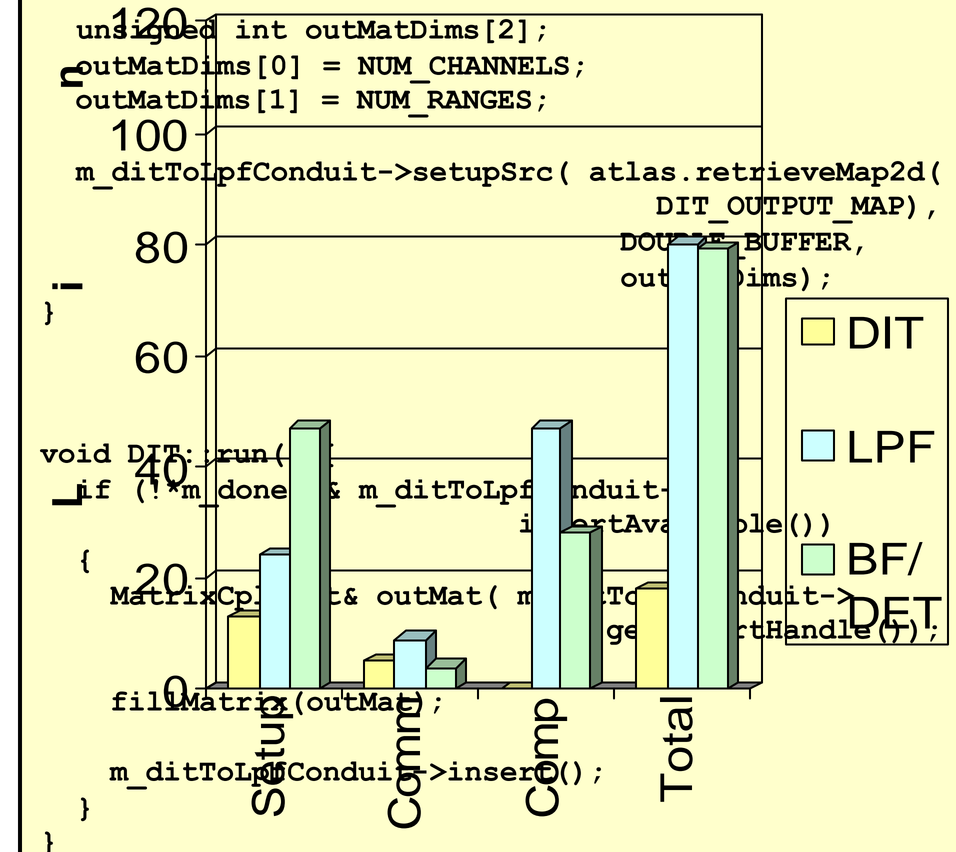
Single Processor Mapping

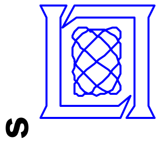


VSIPL | **PVL**

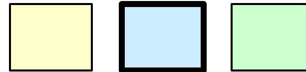
- **VSIPL code has no communication overhead**
- **Some code compaction because PVL uses C++ instead of C**





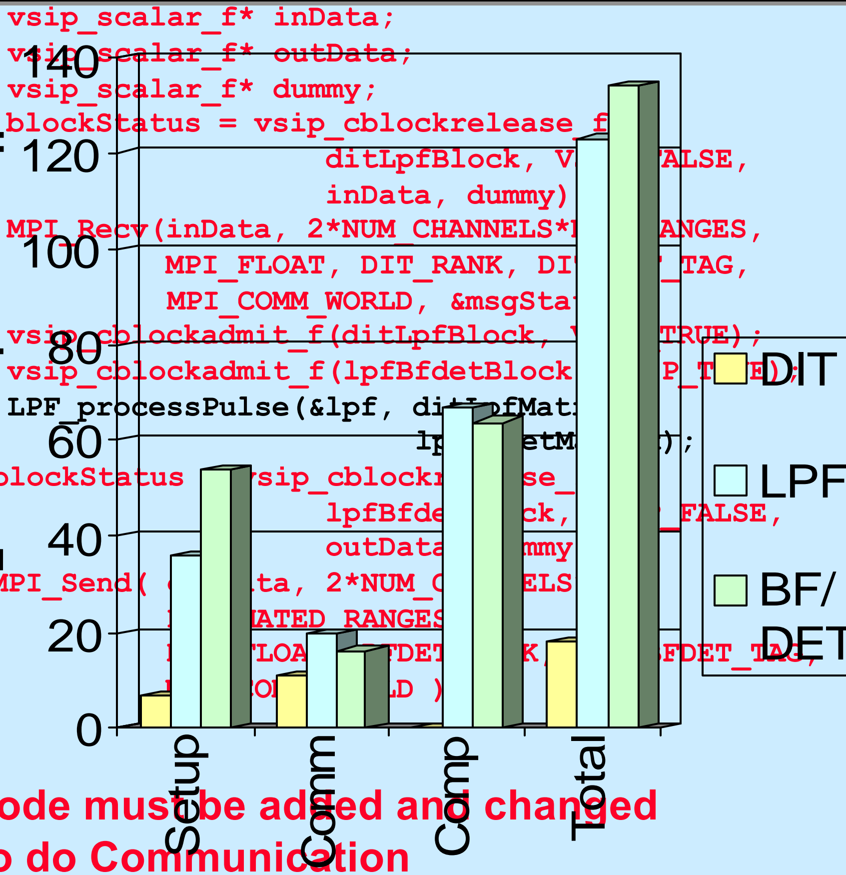


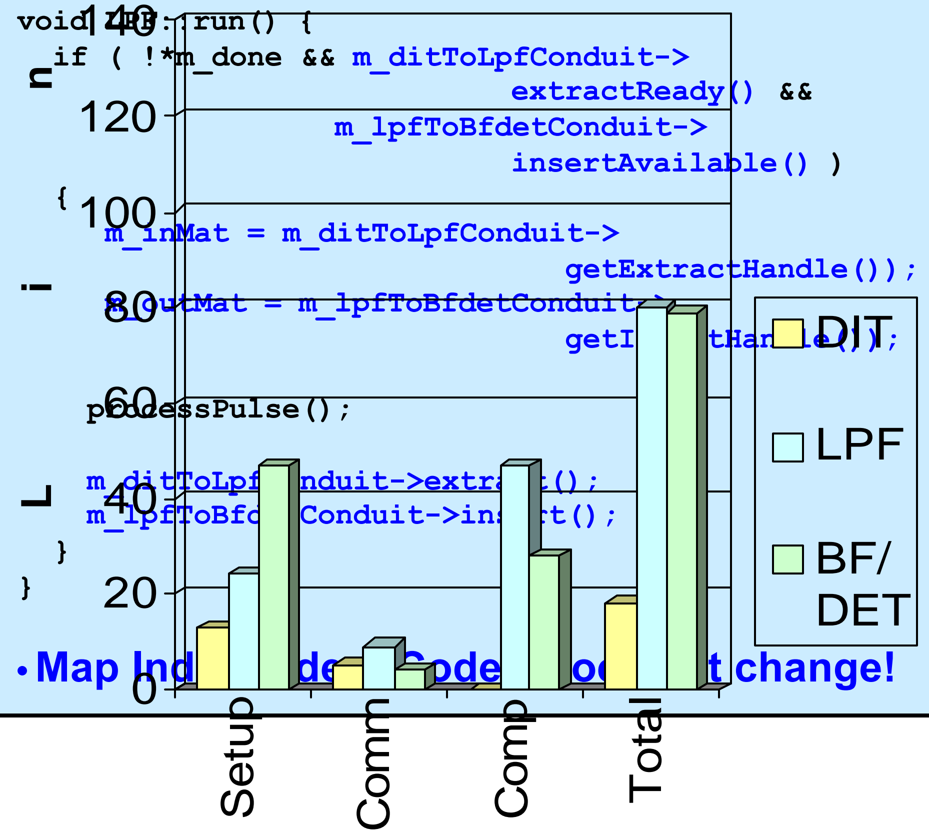
Three Processor Mapping

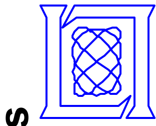


VSIPL + MPI | **PVL**

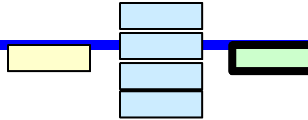
- **Code must be added to the VSIPL implementation for communication**
- **PVL code does not need to change - it is parallel ready**







Six Processor Mapping



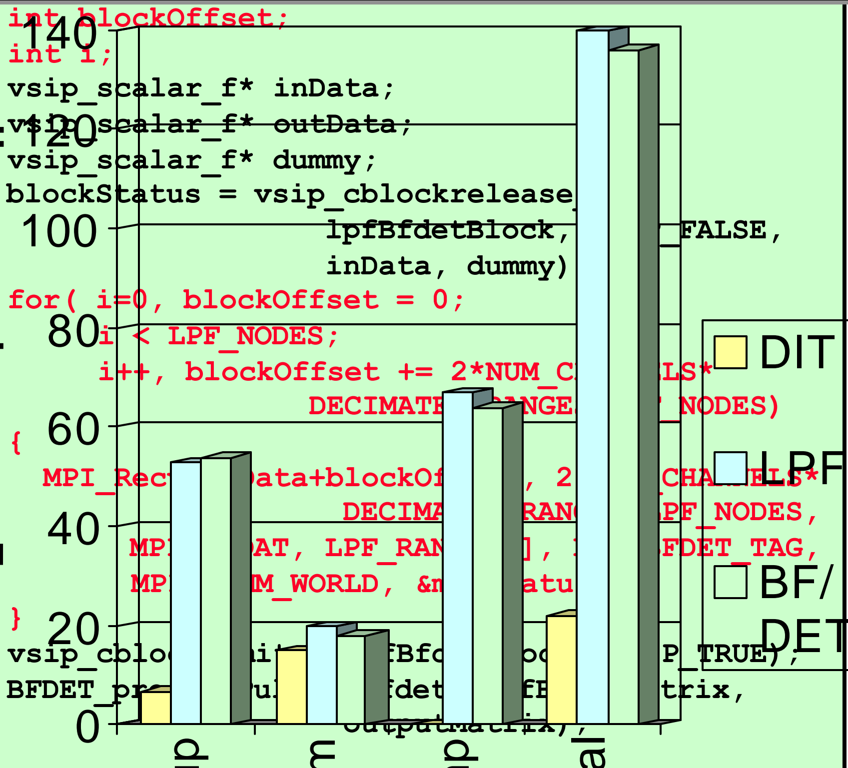
VSIPL + MPI

PVL

- **Small but tedious changes to the VSIPL/MPI implementation**
- **PVL code does not need to change**

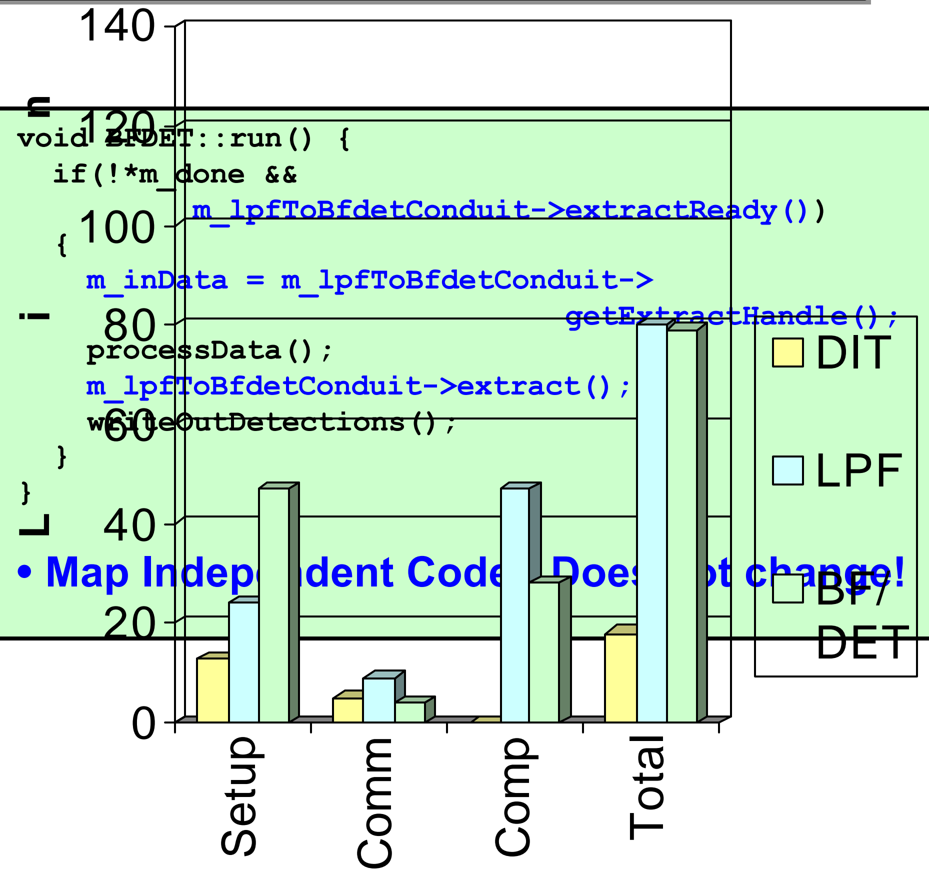
```
int blockOffset;
int i;
vsip_scalar_f* inData;
vsip_scalar_f* outData;
vsip_scalar_f* dummy;
blockStatus = vsip_cblockrelease_
                lpfBfdetBlock,        FALSE,
                inData, dummy)
for( i=0, blockOffset = 0;
     i < LPF_NODES;
     i++, blockOffset += 2*NUM_C     LS*
              DECIMATE        _NODES)
{
  MPI_Rec    ata+blockOf      ,  2      CHANNELS*
               DECIMA      RAN      PF_NODES,
  MP      AT, LPF_RAN      ],       FDET_TAG,
  MP      M_WORLD, &m      atu
}
vsip_cbloc      i      fBfd      oc       P_TRUE);
BFDET_pr      u      fdet      fF       trix,
                outputMatrix);
}
```

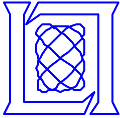
- **Code must be added and changed to do Communication**




```
void BFDET::run() {
  if(!*m_done &&
     m_lpfToBfdetConduit->extractReady())
  {
    m_inData = m_lpfToBfdetConduit->
                    getExtractHandle();
    processData();
    m_lpfToBfdetConduit->extract();
    writeOutDetections();
  }
}
```

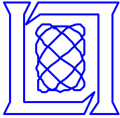
- **Map Independent Code Does not change!**



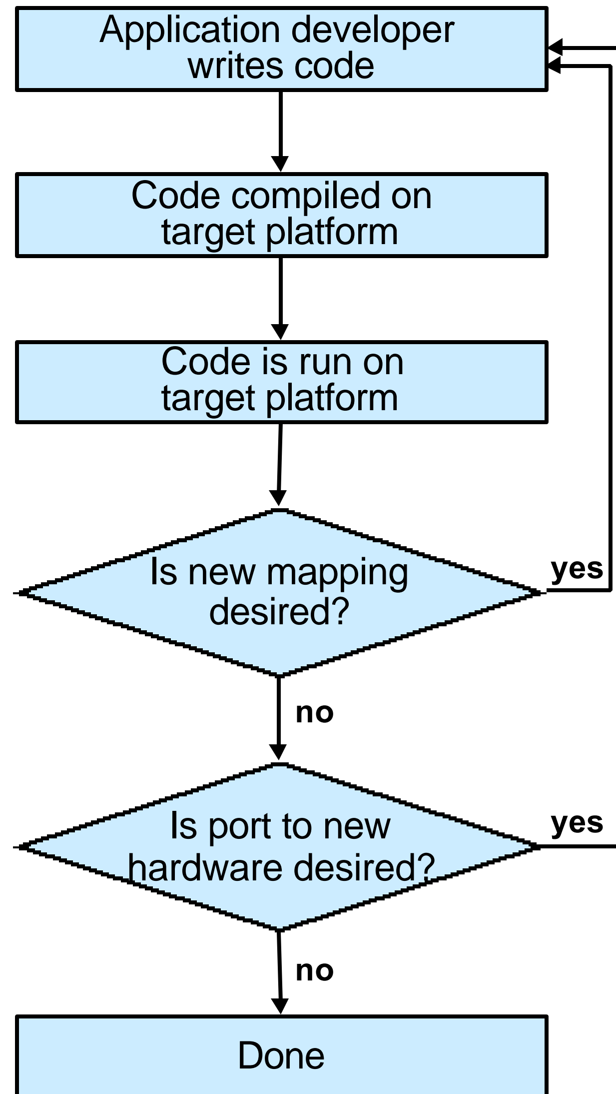


Overview

- Motivation - why write portable software?
- Philosophy
  - how to achieve portability
  - how to measure portability
- Overview of Software Library
- Example signal processing application
- Conclusion

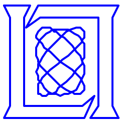


System Development Using Current Software Technology



Traditional Code is:

- **Map Dependent**
- **Inflexible**
- **Non-scalable**



System Development Using Our Library and Philosophy

Application developer writes code

Code debugged on workstation

Code compiled on target platform

Mapper edits map file for target platform

Code is run on target platform

Is new mapping desired? (yes → Mapper edits map file for target platform; no → next)

Is port to new hardware desired? (yes → Code compiled on target platform; no → Done)

Done

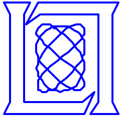
PVL Code is:

- **Map Independent**
- **Flexible**
- **Scalable**
- **Capable of being debugged on a workstation**

Traditional Code is:

- **Map Dependent**
- **Inflexible**
- **Non-scalable**

- **Developers change Maps, not Code**



Conclusion

- Parallel applications written on top of PVL can be fully portable:
  - 0 lines of code changed when scaling the PVL application
- Applications written with VSIPL and MPI are not fully portable:
  - 74 lines of code were added to scale to three processors
  - 23 lines of code were added to scale from 3 to six processors

- A high-level signal processing library with task and data parallel constructs provides a huge increase in productivity for engineers developing signal processing applications because:
  - application code is more flexible - complicated changes to maps can be made without changes to code
  - application code is scalable - applications will work on 1 or 100 node systems without code modification
  - application programs can be written in a more natural way
  - ease of portability enables rapid COTS insertion and technology refresh