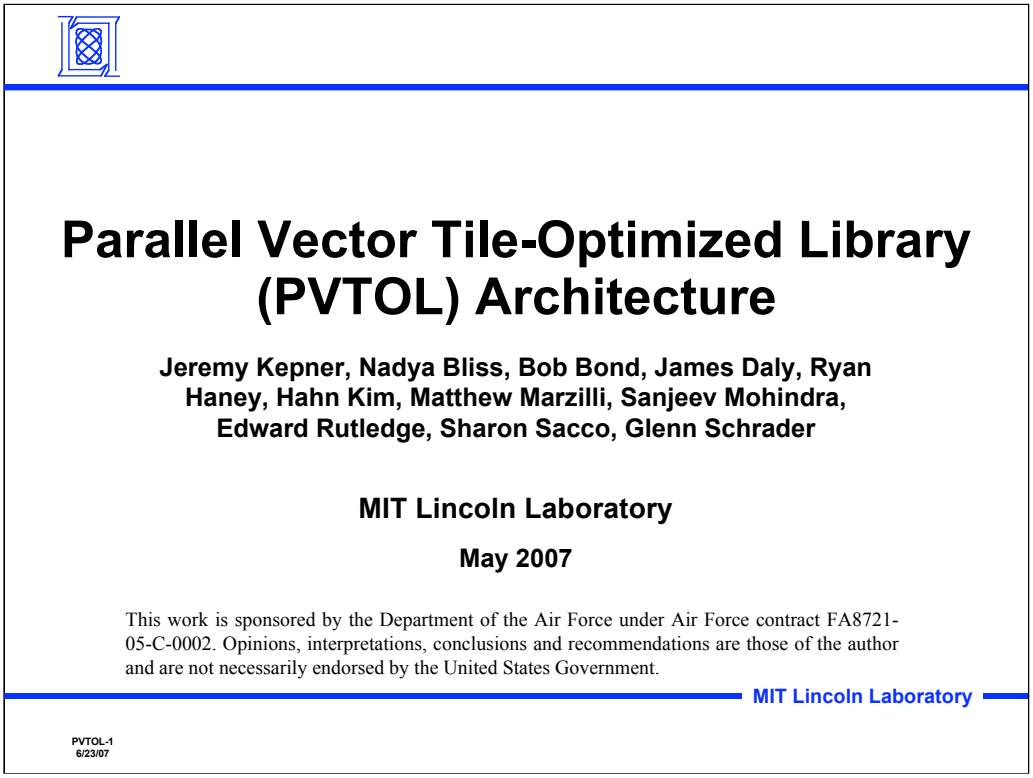
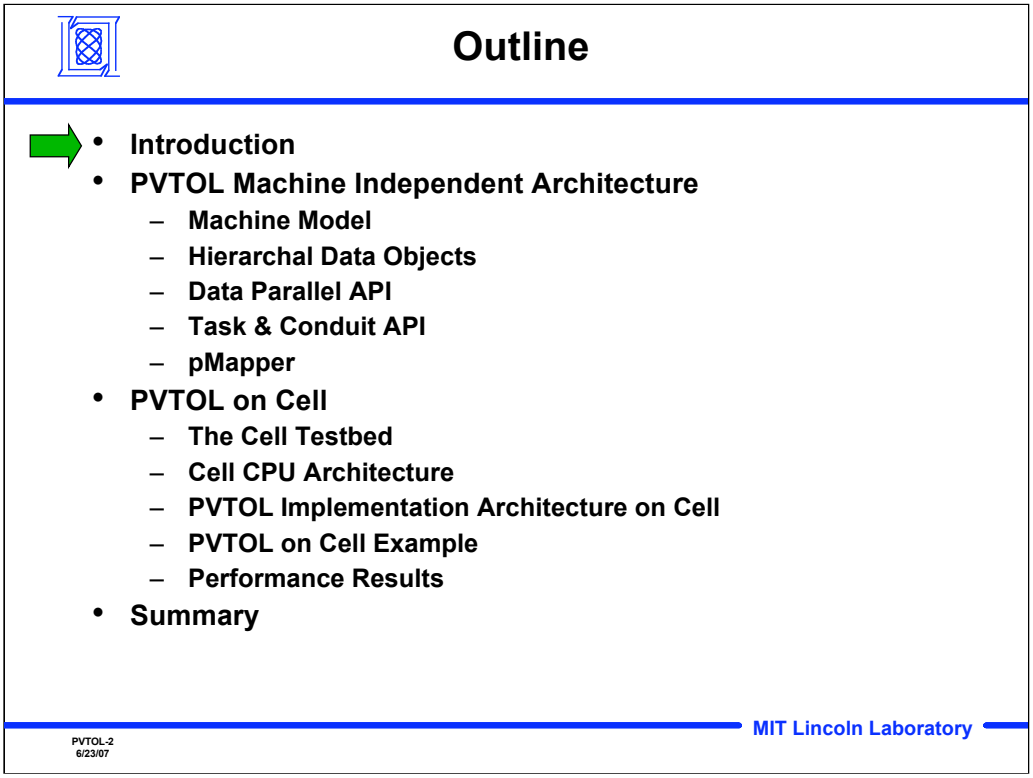
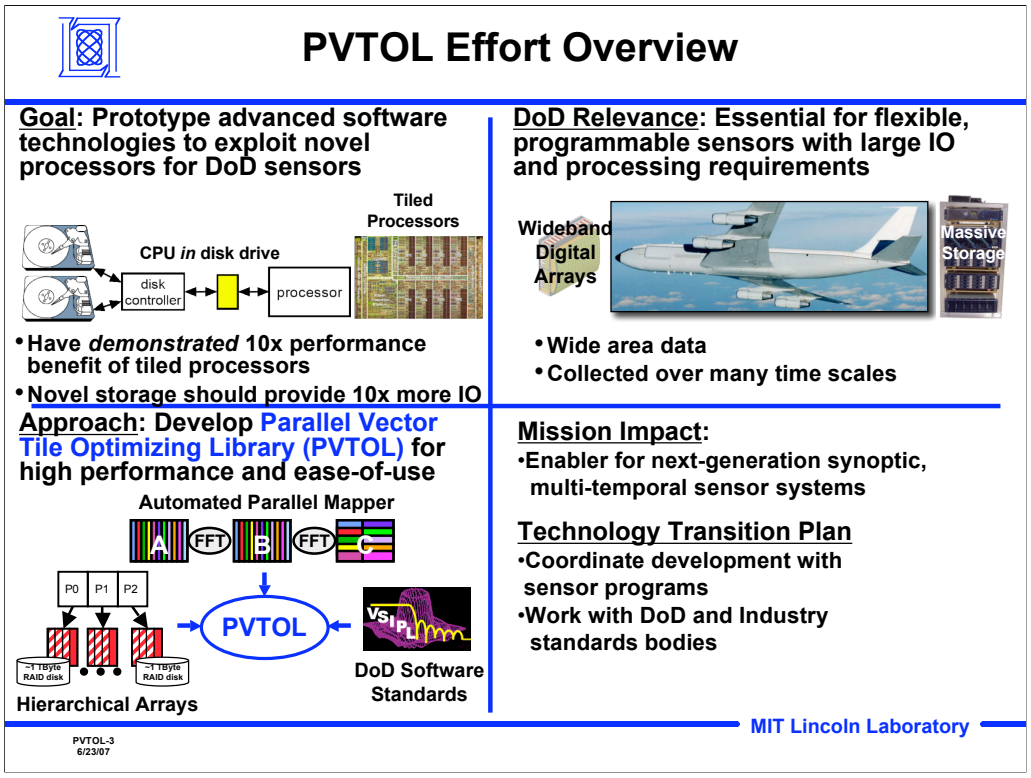

Parallel Vector Tile-Optimized Library (PVTOL) Architecture
Jeremy Kepner, Nadya Bliss, Bob Bond, James Daly, Ryan Haney, Hahn Kim, Matthew Marzilli, Sanjeev Mohindra, Edward Rutledge, Sharon Sacco, Glenn Schrader
MIT Lincoln Laboratory
May 2007
This work is sponsored by the Department of the Air Force under Air Force contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.
MIT Lincoln Laboratory
PVTOL-1
6/23/07

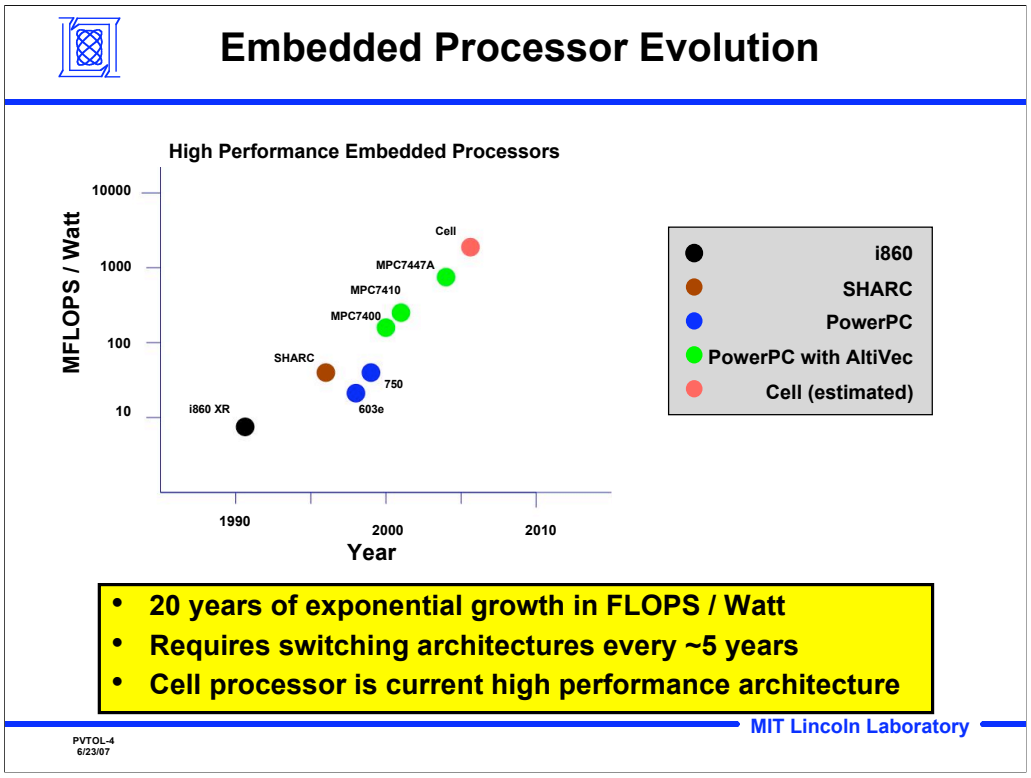
Title slide


Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-2
6/23/07

Outline



The Parallel Vector Tile Optimizing Library (PVTOL) is an effort to develop a new processor architecture for signal processing that exploits the recent shifts in technology to tiled multi-core chips and more tightly integrated mass storage. These technologies are critical for processing the higher bandwidth and longer duration data produced required by synoptic, multi-temporal sensor systems.. The principal challenge in exploiting the new processing architectures for these missions is writing the software in a flexible manner that is portable and delivers high performance. The core technology of this project is the Parallel Vector Tile Optimizing Library (PVTOL), which will use recent advances in automating parallel mapping technology, hierarchical parallel arrays, combined with the Vector, Signal and Image Processing Library (VSIPL) open standard to deliver portable performance on tiled processors.

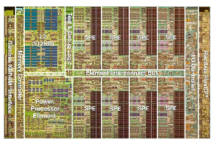


Growth in embedded processor performance, in terms of FLOPS/Watt, has grown exponentially over the last 20 years. No single processing architecture has dominated over this period, hence in order to leverage this increase in performance, embedded system designers must switch processing architectures approximately every 5 years.

MFLOPS / W for i860, SHARC, 603e, 750, 7400, and 7410 are extrapolated from board wattage. They also include other hardware energy use such as memory, memory controllers, etc. 7447A and the Cell estimate are for the chip only. Effective FLOPS for all processors are based on 1024 FFT timings. Cell estimate uses hand coded TDFIR timings for effective FLOPS.

Cell Broadband Engine

IBM



- Cell was designed by IBM, Sony and Toshiba
- Asymmetric multicore processor
  - 1 PowerPC core + 8 SIMD cores

SONY

- Playstation 3 uses Cell as main processor

Computer Systems, Inc. MERCURY

- Provides Cell-based computer systems for high-performance applications

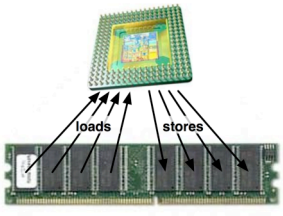
MIT Lincoln Laboratory

PVTOL-5
6/23/07

The current high performance processing architecture is the Cell processor, designed by a collaboration between IBM, Sony and Toshiba. While the Cell was initially targeted for Sony's Playstation 3, it has use in wide range of applications, including defense, medical, etc. Mercury Computer Systems provides Cell-based systems that can be used to develop high-performance embedded systems.

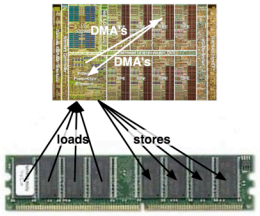
Multicore Programming Challenge

Past Programming Model: Von Neumann



- Great success of Moore's Law era
  - Simple model: load, op, store
  - Many transistors devoted to delivering this model
- Moore's Law is ending
  - Need transistors for performance

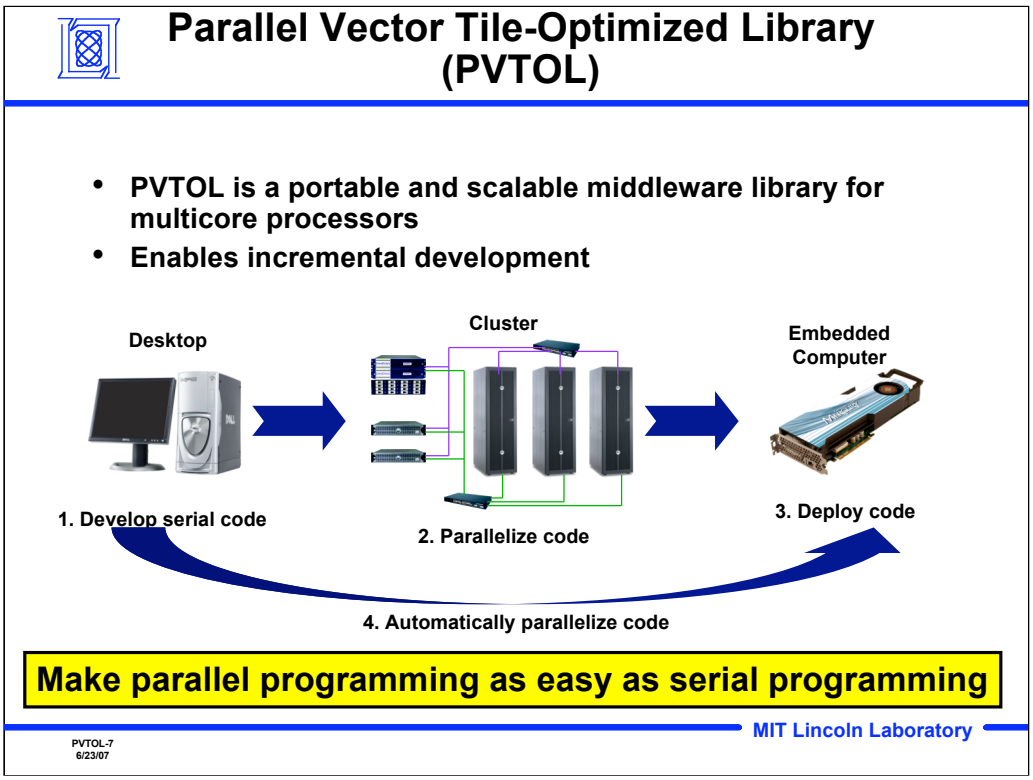
Future Programming Model: ???



- Processor topology includes:
  - Registers, cache, local memory, remote memory, disk
- Cell has *multiple* programming models

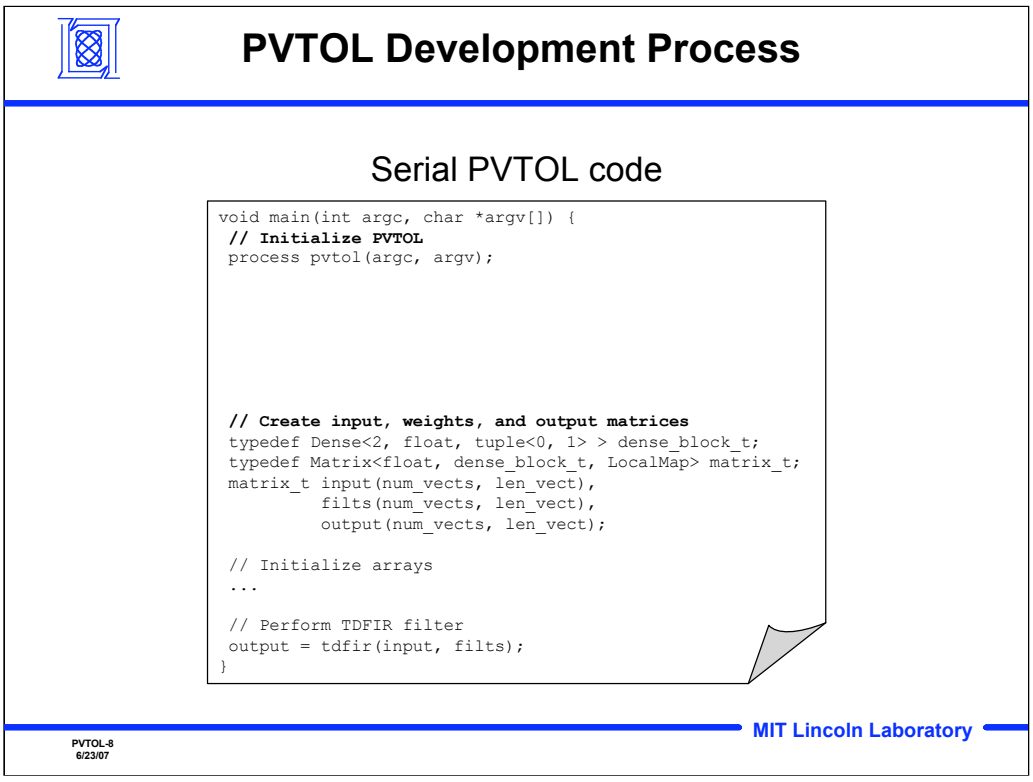
Increased performance at the cost of exposing complexity to the programmer

For decades, Moore's Law has enabled ever faster processors that have supported the traditional von Neumann programming model, i.e. load data from memory, process, then save the results to memory. As clock speeds near 4 GHz, physical limitations in transistor size are leading designers to build more processor cores (or "tiles") on each chip rather than faster processors. Multicore processors improve raw performance but expose the underlying processor and memory topologies. This results in increased programming complexity, i.e. the loss of the von Neumann programming model.



PVTOL is focused on addressing the programming complexity of associated with emerging “Topological Processors”. Topological Processors require the programmer to understand the physical topology of the chip to get high efficiency. There are many such processors emerging into the market. The Cell processor is an important example of such a chip. The current PVTOL effort is focused on getting high performance from the Cell processor on signal and image processing applications. The PVTOL interface is designed to address a wide range of processors including multicore and FPGAs.

PVTOL enables software developers to develop high-performance signal processing application on a desktop computer, parallelize the code on commodity clusters, then deploy the code onto an embedded computer, with minimal changes to the code. PVTOL also includes automated mapping technology that will automatically parallelize the application for a given platform. Applications developed on a workstation can then be deployed on an embedded computer and the library will parallelize the application without any changes to the code.



This slide shows an example of serial PVTOL code that allocates data to be processed by a time-domain FIR filter. The user simply allocates three matrices containing the input and output data and the filter coefficients. The LocalMap tag in the Matrix type definition indicates that the matrices will be allocated all on the processor's local memory.

PVTOL Development Process

Parallel PVTOL code

```
void main(int argc, char *argv[]) {
 // Initialize PVTOL
 process pvtol(argc, argv);



 // Add parallel map
 RuntimeMap map1(...);

 // Create input, weights, and output matrices
 typedef Dense<2, float, tuple<0, 1> > dense_block_t;
 typedef Matrix<float, dense_block_t, RunTimeMap> matrix_t;
 matrix_t input(num_vects, len_vect , map1),
          filts(num_vects, len_vect , map1),
          output(num_vects, len_vect , map1);

 // Initialize arrays
 ...

 // Perform TDFIR filter
 output = tdfir(input, filts);
}
```


This slide shows how to parallelize the serial code shown in the previous slide. The programmer creates a map, which contains a concise description of how to parallelize a matrix across multiple processors. The RuntimeMap in the Matrix type definition indicates that the map for the matrices is constructed at runtime. The map object is then passed into the matrix constructors, which allocate memory on multiple processors as described in the map object.

PVTOL Development Process

Embedded PVTOL code

```
void main(int argc, char *argv[]) {
 // Initialize PVTOL
 process pvtol(argc, argv);

 // Add hierarchical map
 RuntimeMap map2(...);

 // Add parallel map
 RuntimeMap map1(..., map2);

 // Create input, weights, and output matrices
 typedef Dense<2, float, tuple<0, 1> > dense_block_t;
 typedef Matrix<float, dense_block_t, RunTimeMap> matrix_t;
 matrix_t input(num_vects, len_vect , map1),
          filts(num_vects, len_vect , map1),
          output(num_vects, len_vect , map1);

 // Initialize arrays
 ...

 // Perform TDFIR filter
 output = tdfir(input, filts);
}
```

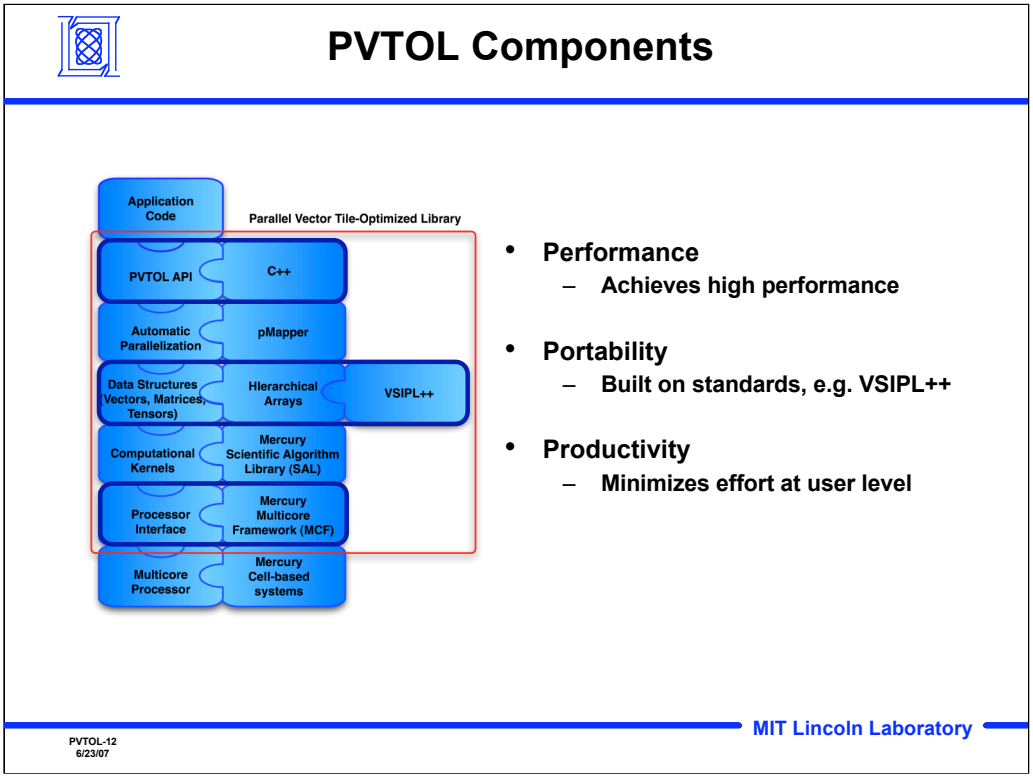

This slide shows how to deploy the parallel code from the previous slide onto an embedded multicore system. As before, map1 describes how to parallelize matrices across multiple processors. A new map, map2, is added to describe how each processor should divide its section of the matrices across cores. This allocates a “hierarchical array” across the processor hierarchy.

PVTOL Development Process

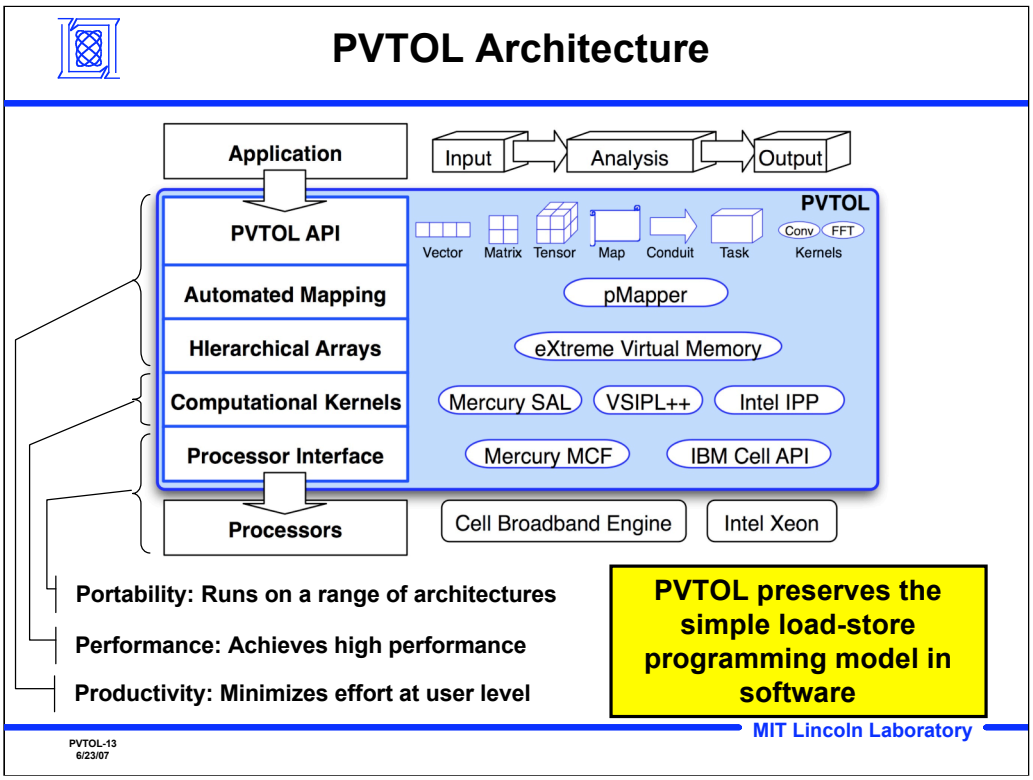
Automapped PVTOL code

```
void main(int argc, char *argv[]) {
 // Initialize PVTOL
 process pvtol(argc, argv);




 // Create input, weights, and output matrices
 typedef Dense<2, float, tuple<0, 1> > dense_block_t;
 typedef Matrix<float, dense_block_t, AutoMap> matrix_t;
 matrix_t input(num_vects, len_vect , map1),
          filts(num_vects, len_vect , map1),
          output(num_vects, len_vect , map1);

 // Initialize arrays
 ...

 // Perform TDFIR filter
 output = tdfir(input, filts);
}
```

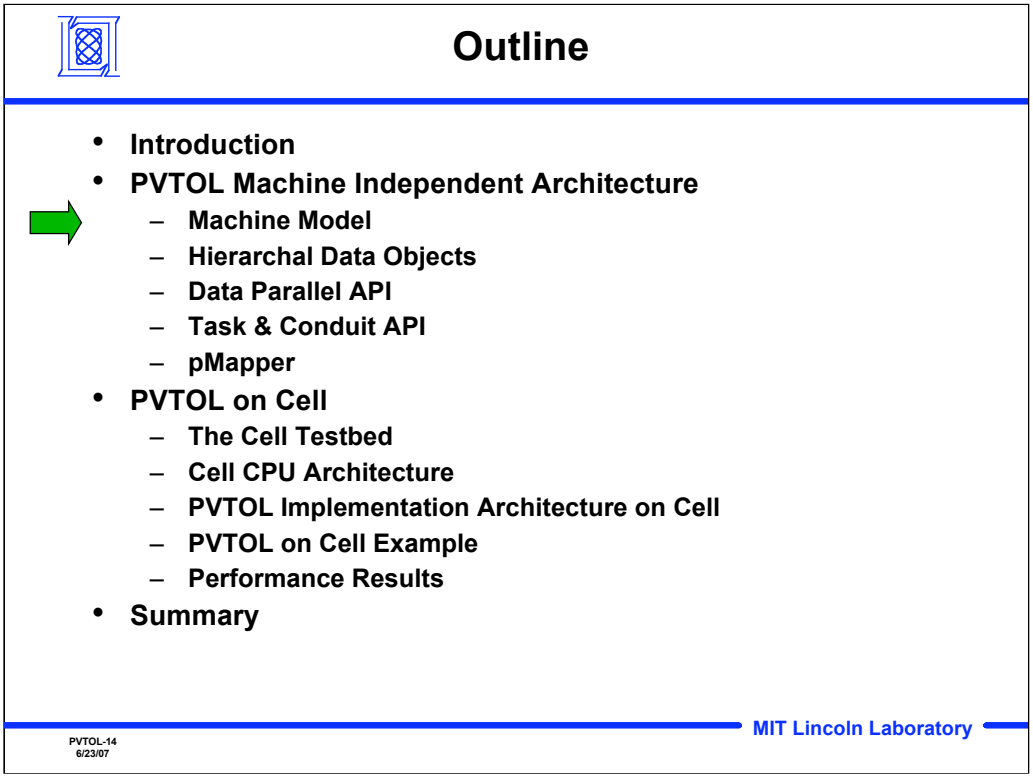

Maps give the programmer direct control over how memory should be allocated across the processor hierarchy. Alternatively, the programmer can let PVTOL automatically parallelize arrays. The only change required is to replace the LocalMap tag in the Matrix type definition with AutoMap. Note that this code can be deployed without any changes on a workstation, cluster or embedded computer.



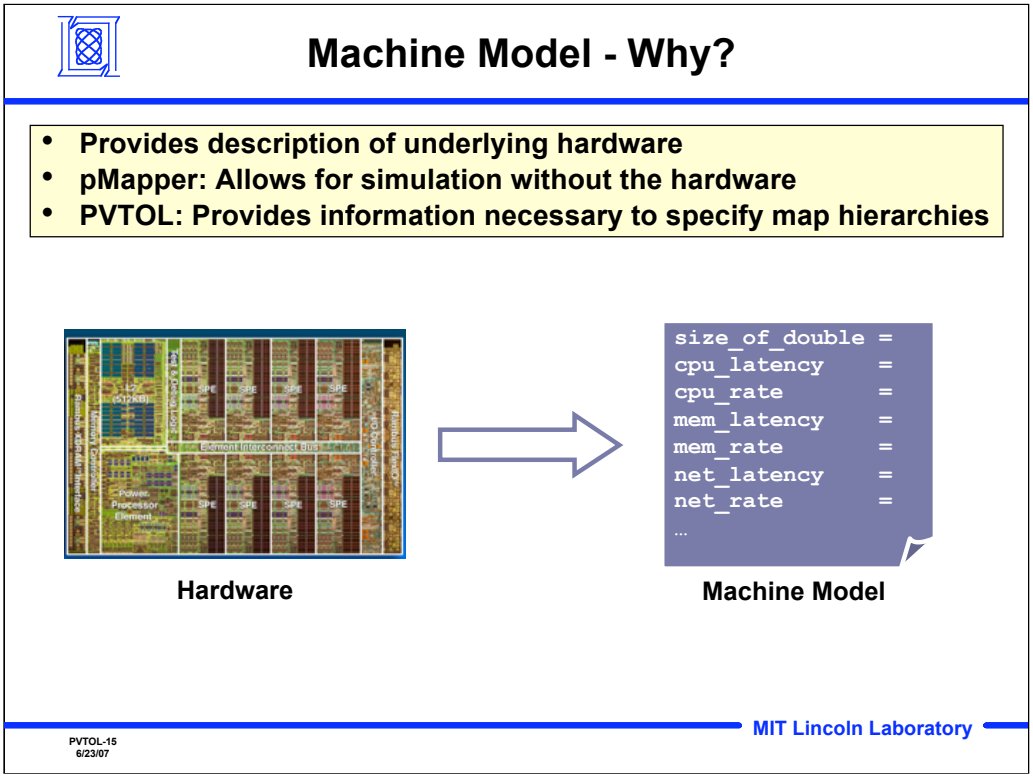
This slide shows the various software components that are used in PVTOL and how they relate to Performance, Portability and Productivity.



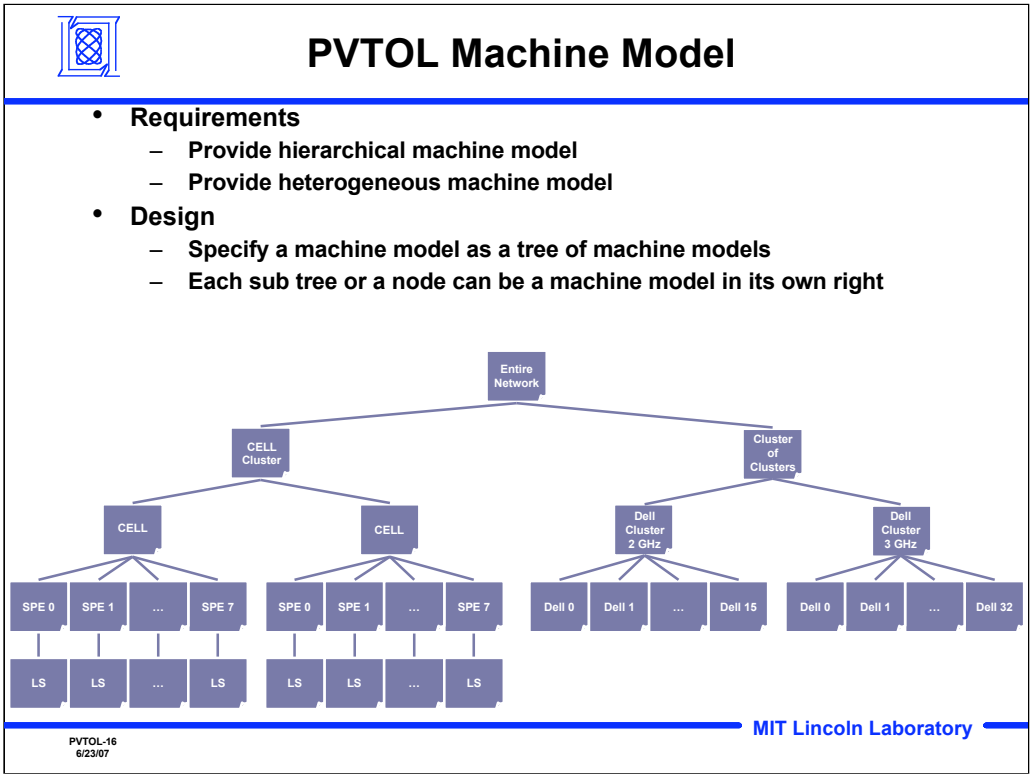
This slide shows a layered view of the PVTOL architecture. At the top is the application. The PVTOL API exposes high-level structures for data (e.g. vectors), data distribution (e.g. maps), communication (e.g. conduits) and computation (e.g. tasks and computational kernels). High level structures improve the productivity of the programmer. By being built on top of existing technologies, optimized for different platforms, PVTOL provides high performance. And by supporting a range of processor architectures, PVTOL applications are portable. The end result is that rather than learning new programming models for new processor technologies, PVTOL preserves the simple von Neumann programming model most programmers are used to.


Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-14
6/23/07

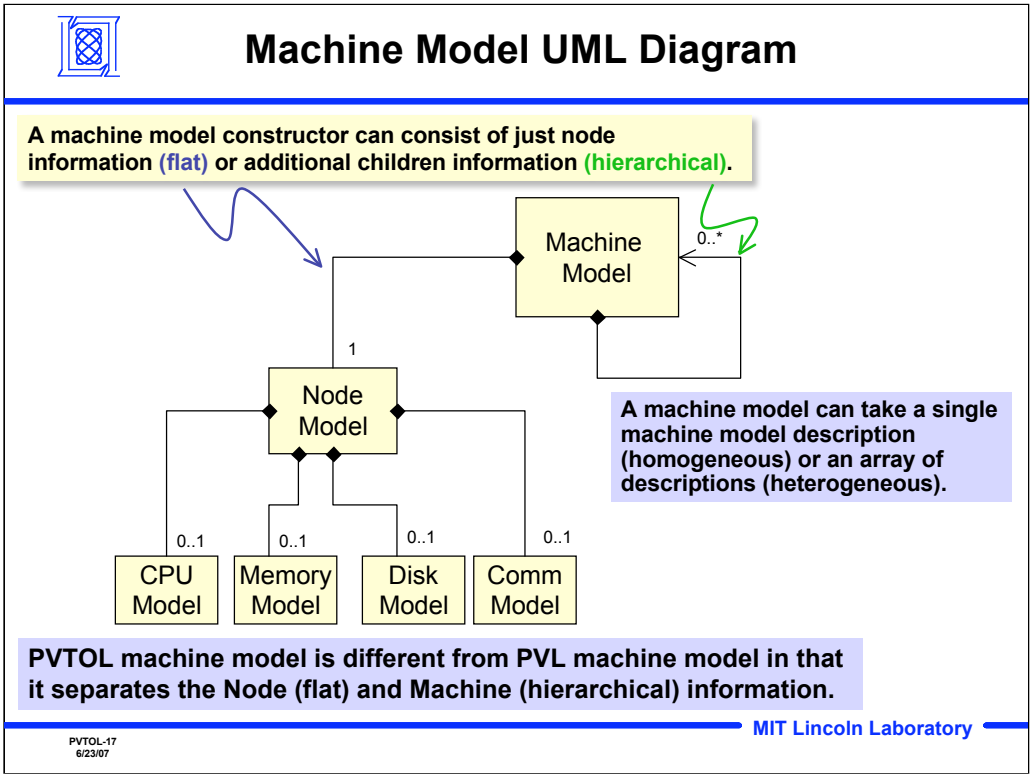
Outline



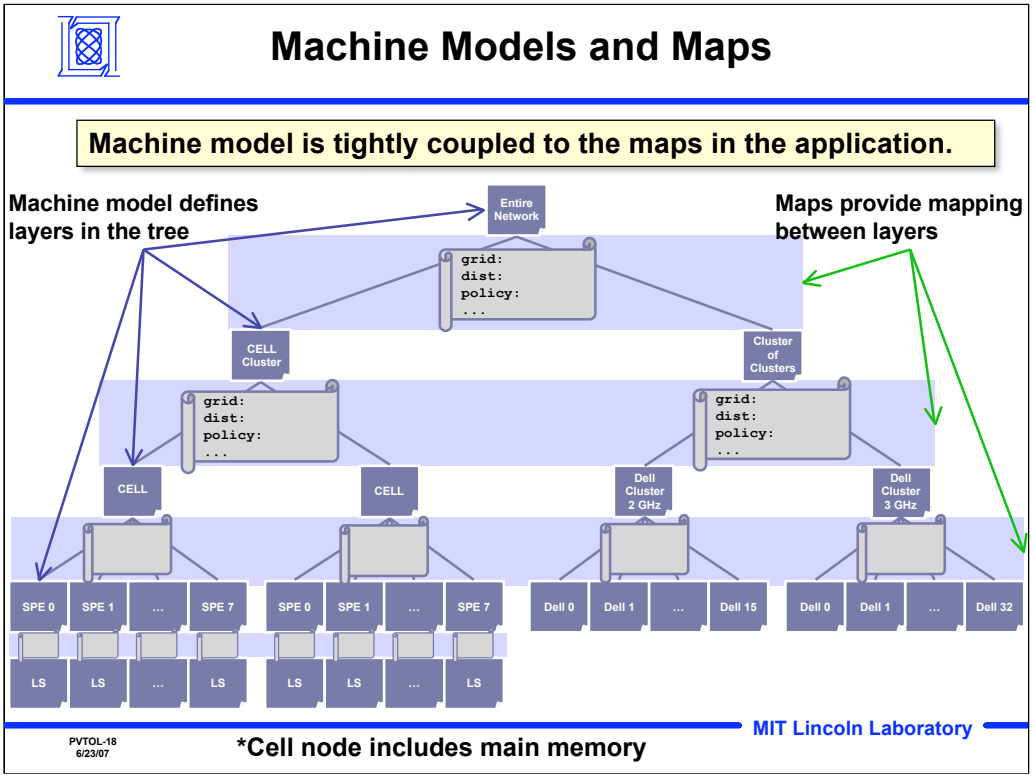
The development of a machine model that can describe different processing architectures has two purposes. First, a machine model allows the pMapper automated mapping environment to simulate how well different mappings of an application perform without executing the application on the actual hardware. Second, a machine model provides information about the architecture needed by PVTOL to specify how to map hierarchical arrays to the processing hierarchy.



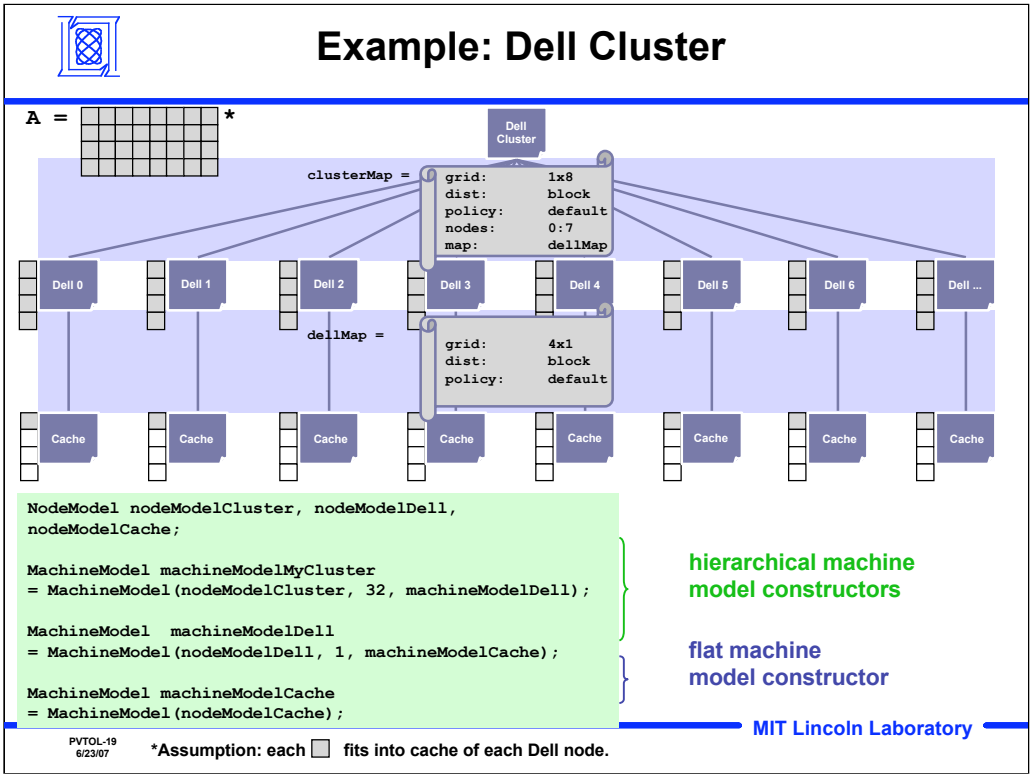
PVTOL requires that a machine model be able to describe hierarchical and heterogeneous processing architectures. For example, a cluster of Cell processors is a hierarchy of processors; at the top level the hierarchy consists of multiple Cell processors and at the next level each Cell processor consists of multiple SPE processors. Heterogeneity enables the description of processing architectures composed of different hardware architectures, e.g. Cell and Intel processors. The PVTOL machine model is recursive in that a machine model can be described as a tree of machine models.



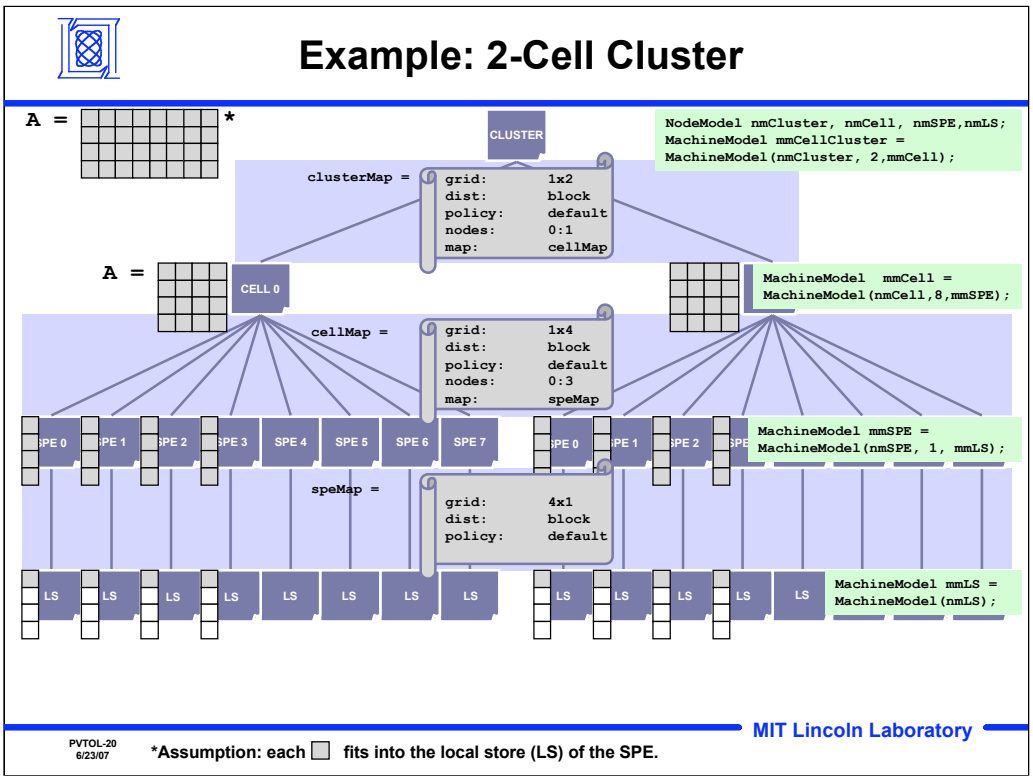
This slide shows the UML diagram for a machine model. A machine model constructor can consist of just node information (flat) or additional children information (hierarchical). A machine model can take a single machine model description (homogeneous) or an array of descriptions (heterogeneous). The PVTOL machine model extends PVL's model in that it can describe both flat and hierarchical processing architectures.



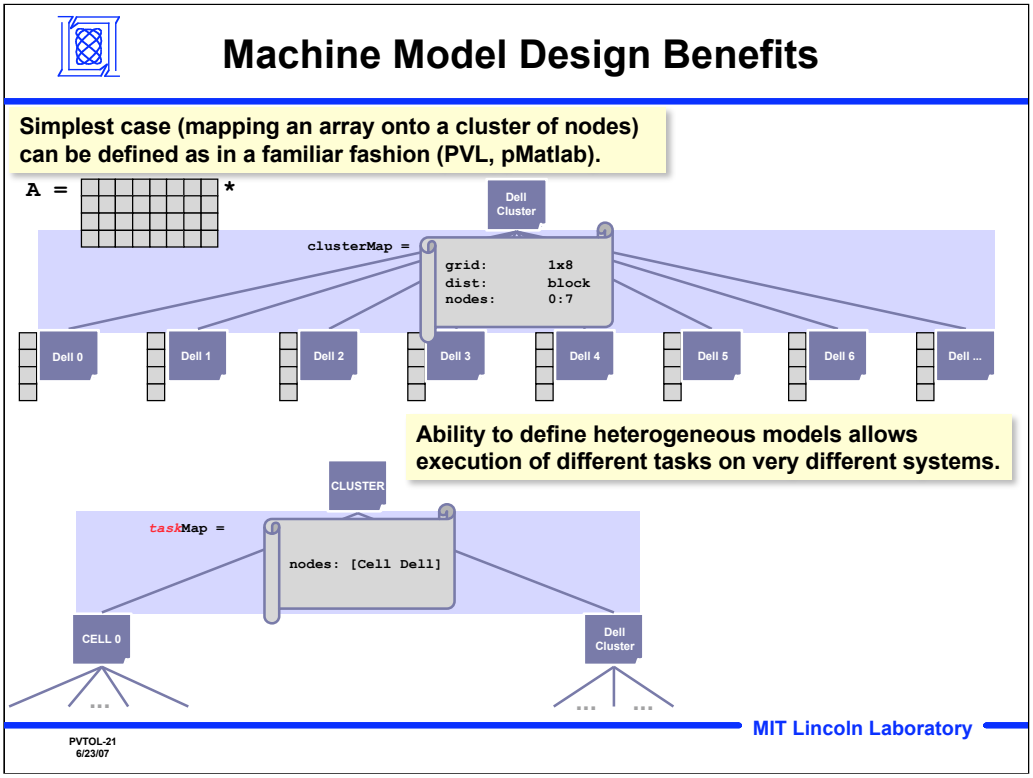
A machine model is tightly coupled to the map. Machine model descriptions define layers in the tree, while maps provide mappings *between* layers (*N* layers in the machine mode could have at most *N-1* layers of maps). The map API has to be adjusted to allow the user to pass in an array of maps for heterogeneous systems. The map API does not require the user to specify storage level. When specifying hierarchical maps, the layers specified have to start at the root and be contiguous



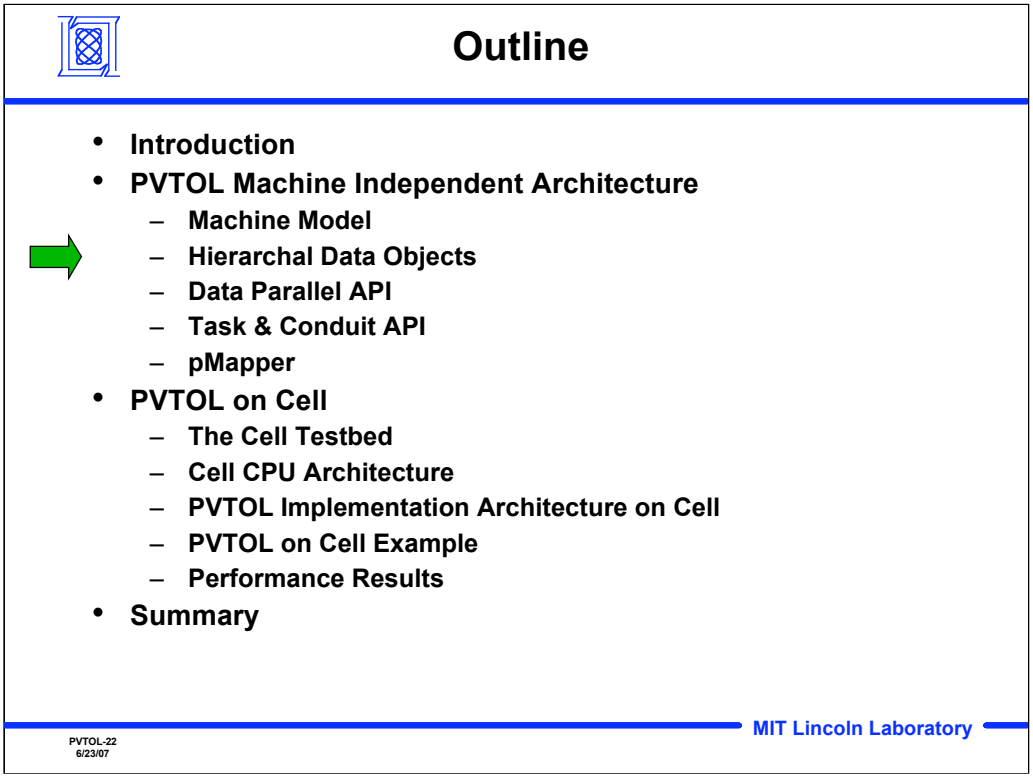
This shows an example of how to construct a machine model of a cluster comprised of Dell computers. This example describes a cluster of 32 Dell nodes, with each node containing a single processor and each processor containing cache. This machine model contains a two-level hierarchy. The matrix A can be mapped to this hierarchy by two maps: clusterMap and dellMap. clusterMap describes how to distribute data across nodes and dellMap describes how to partition data on each Dell node into blocks that will fit into cache.



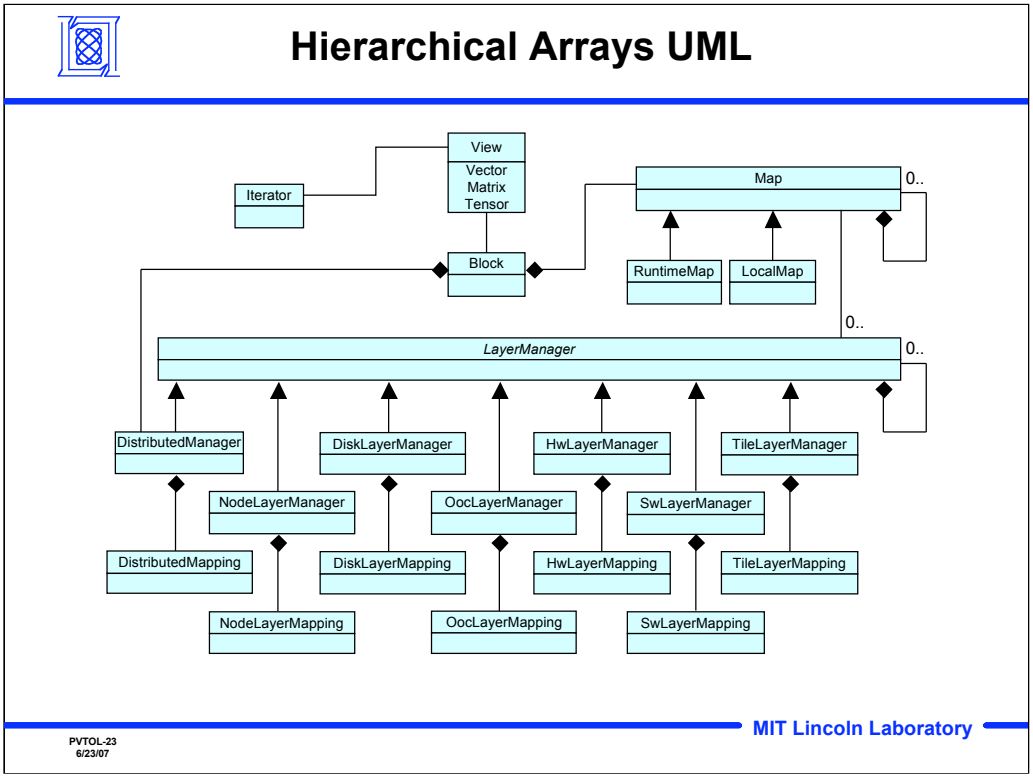
This shows an example of how to construct a machine model of a cluster comprised of Cell processors. This example describes a cluster of 2 Cell processors, with each node containing a 8 SPE's and each processor containing a local store. This machine model contains a three-level hierarchy. The matrix A can be mapped to this hierarchy by three maps: clusterMap, dellMap and speMap. clusterMap describes how to distribute data across Cell processors, dellMap describes how to distribute data on each Cell processor between SPE's and speMap describes how to partition data owned by each SPE into blocks that will fit into its local store.



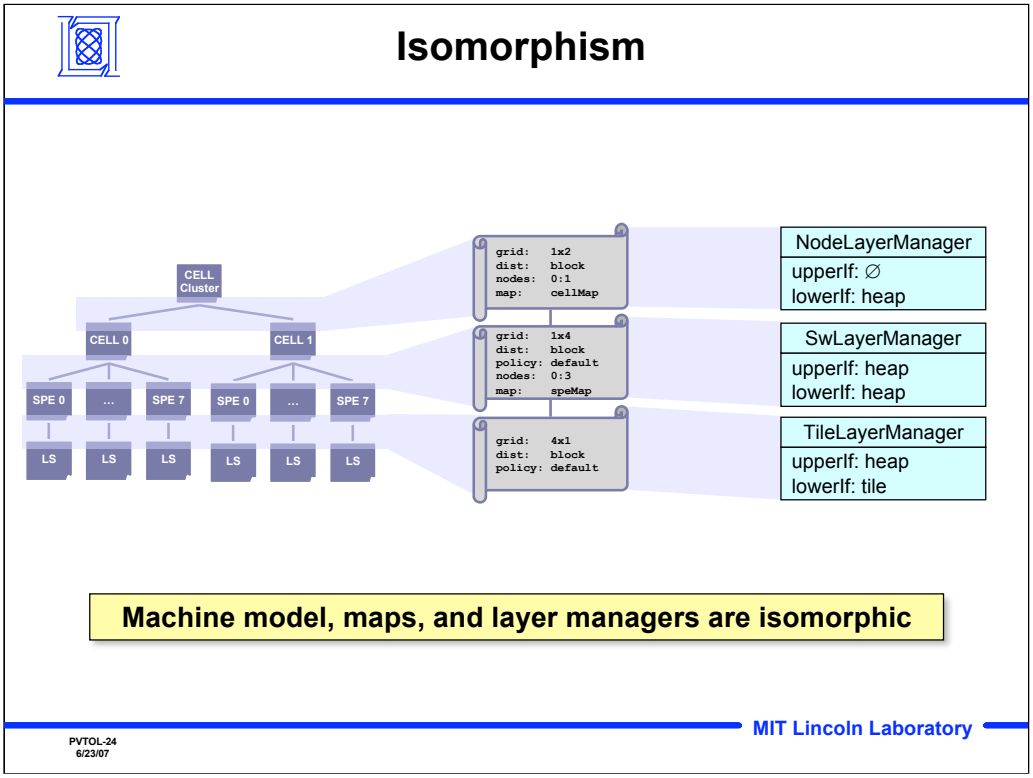
This slide describes the advantages of tree machine models/maps. In the simplest case, a machine model can be easily specified in the same way as PVL and pMatlab. From the API point of view, this is expressive enough for both flat and hierarchical machine models/maps. It allows the user to use different machine models in different tasks, since task maps do not require that programmer to specify a data distribution.


Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-22
6/23/07

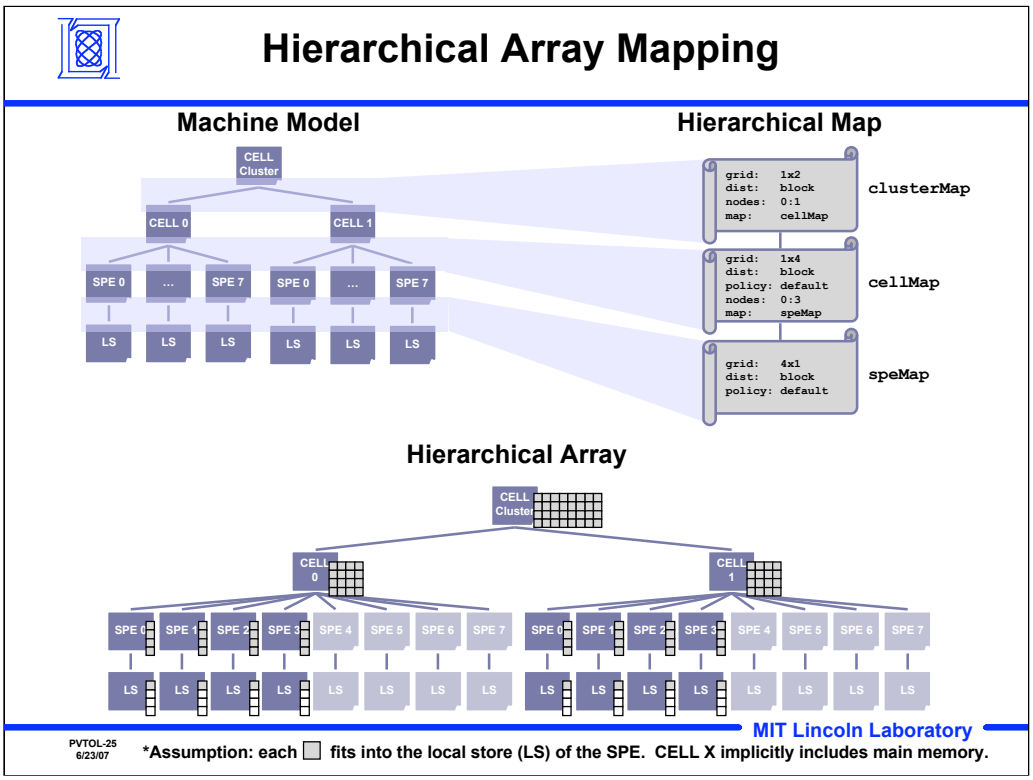
Outline



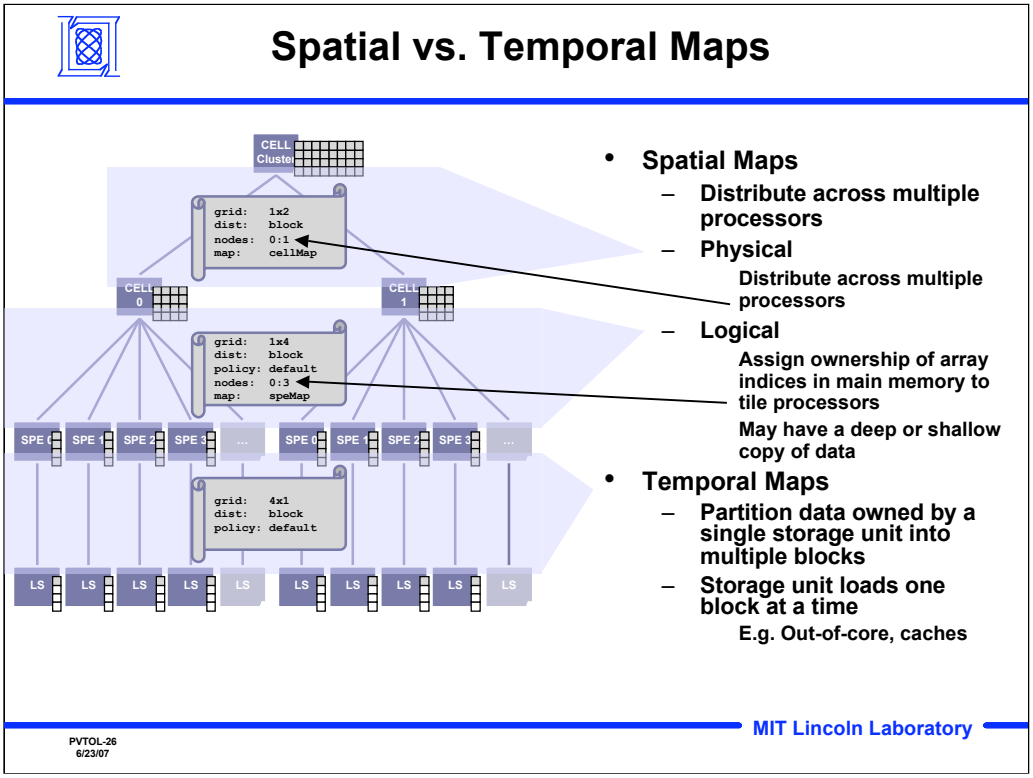
Hierarchical arrays are designed such that details of the memory hierarchy are hidden from the programmer. Programmers access data Block objects via a Vector, Matrix or Tensor View object. Iterator objects provide an architecture independent mechanism for accessing data. Map objects describe how data in a Block object is distributed across the memory hierarchy. Different layer managers are instantiated depending on which layers in the memory hierarchy data is distributed across, e.g. tile memory, local main memory, remote main memory, disk, etc.



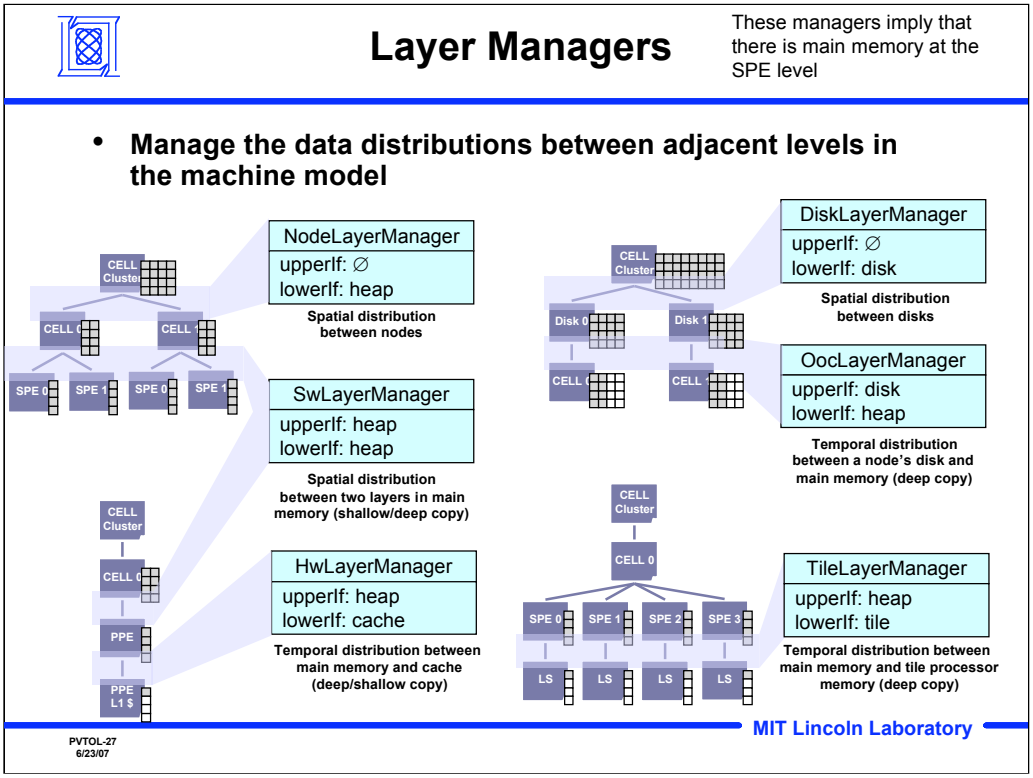
Distribution of data across a layer in the machine model has a one-to-one correspondence with one layer in the hierarchical map and with one LayerManager object.



This slide shows an example of how a hierarchical map distributes data across the memory hierarchy to construct a hierarchical array. Suppose we have a matrix with 4 rows and 8 columns (4x8). The highest level map (clusterMap) divides the the column dimension into 2 blocks across nodes 0 and 1 in the CELL cluster. This results in a 4x4 submatrix on each CELL node. The next level map (cellMap) divides the column dimension into 4 blocks across SPE's 0 through 3 on each CELL node. This results in a 4x1 submatrix owned by each SPE. Finally, the lowest level map (speMap) divides the row dimension into 4 blocks. This results in four 1x1 submatrices. Only each 1x1 submatrix is loaded into the SPE's local store one at a time for processing.



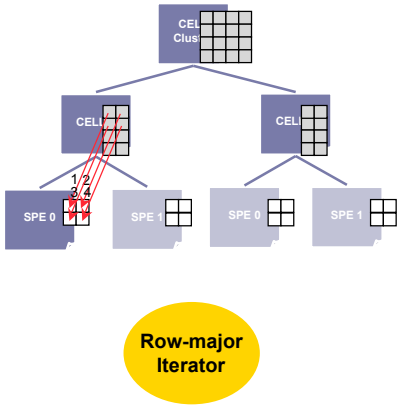
Maps describe how to partition an array. There are two types of maps: spatial and temporal. A spatial map describes how elements of an array are divided between multiple processors. A physical spatial map breaks up an array into blocks that are physically located on separate processors, e.g. dividing an array between two Cell processors. A logical map differs from a physical map in that the array being partitioned remains intact. Rather, the arrays is logically divided into blocks that are owned by different processors. For example, on a single Cell processor, the array resides in main memory. A logical map may assign blocks of rows to each SPE. Finally, temporal maps divided arrays into blocks that are loaded into memory one at a time. For example, a logicial map may divide an array owned by a single SPE into blocks of rows. The array resides in main memory and the SPE loads one block at a time into its local store.



PVTOL instantiates different layer managers for data objects, depending on where in the memory hierarchy the data object is allocated. The layer manager manages the movement of data between the adjacent memory layers. The machine model restricts the types of layer managers; managers are implemented only for pairs of layers that can be adjacent, e.g. there is no layer manager for moving data from disk to tile memory since those two layers cannot be adjacent.

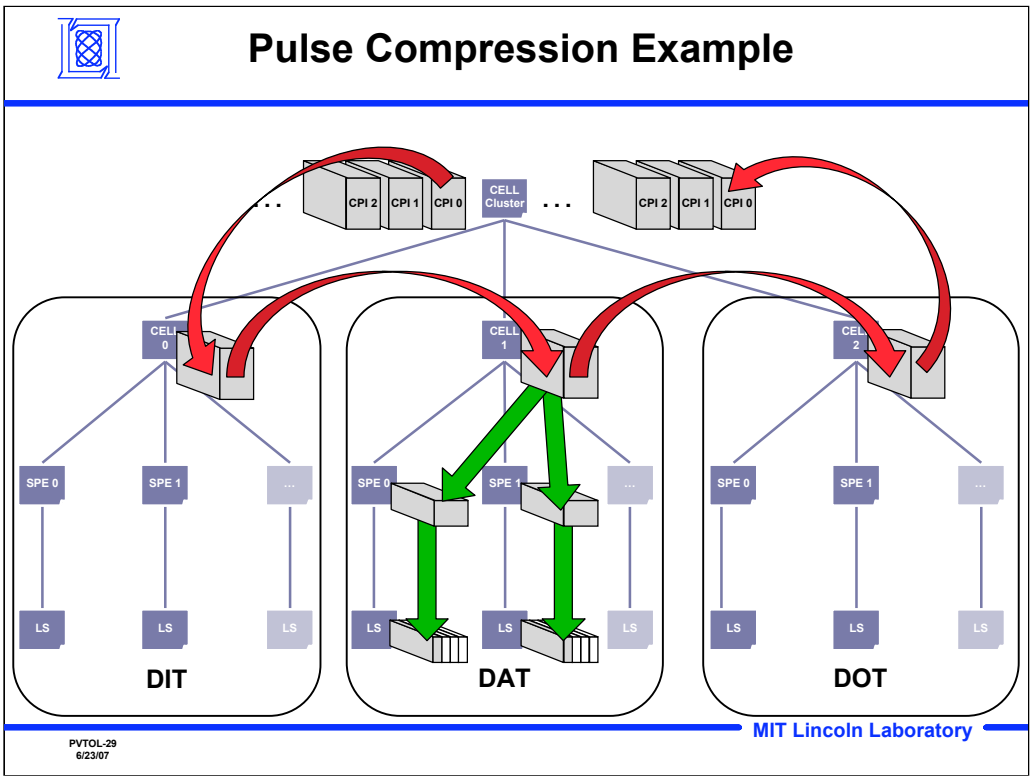
Tile Iterators

- Iterators are used to access temporally distributed tiles

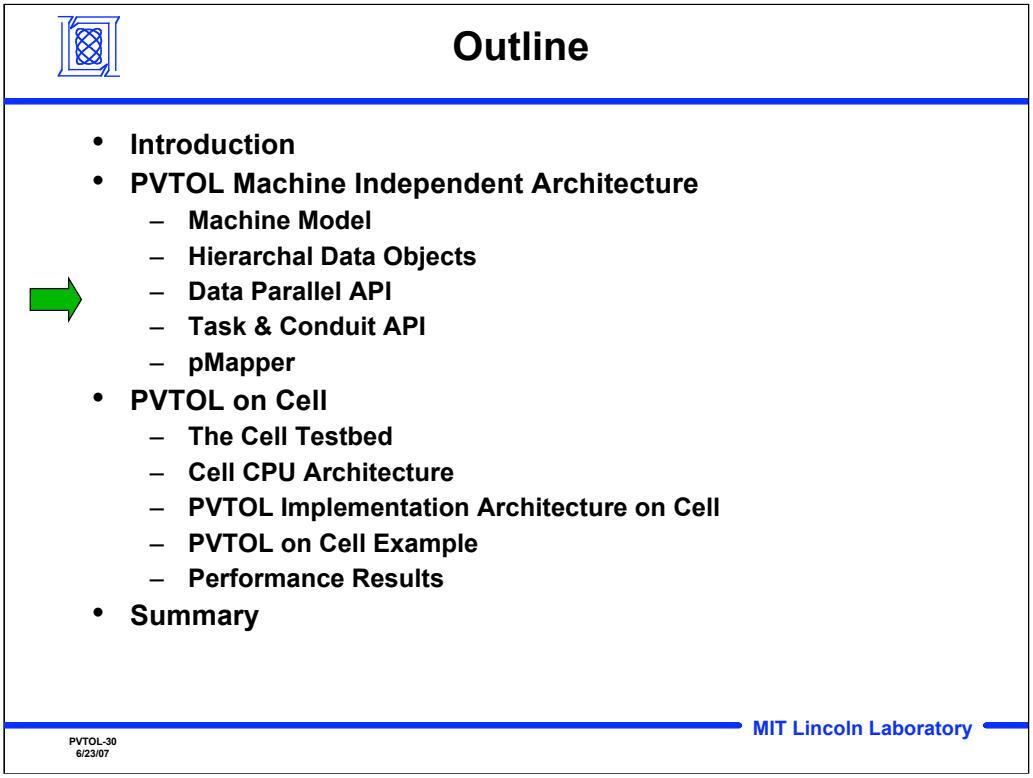


- Kernel iterators
  - Used within kernel expressions
- User iterators
  - Instantiated by the programmer
  - Used for computation that cannot be expressed by kernels
  - Row-, column-, or plane-order
- Data management policies specify how to access a tile
  - Save data
  - Load data
  - Lazy allocation (pMappable)
  - Double buffering (pMappable)

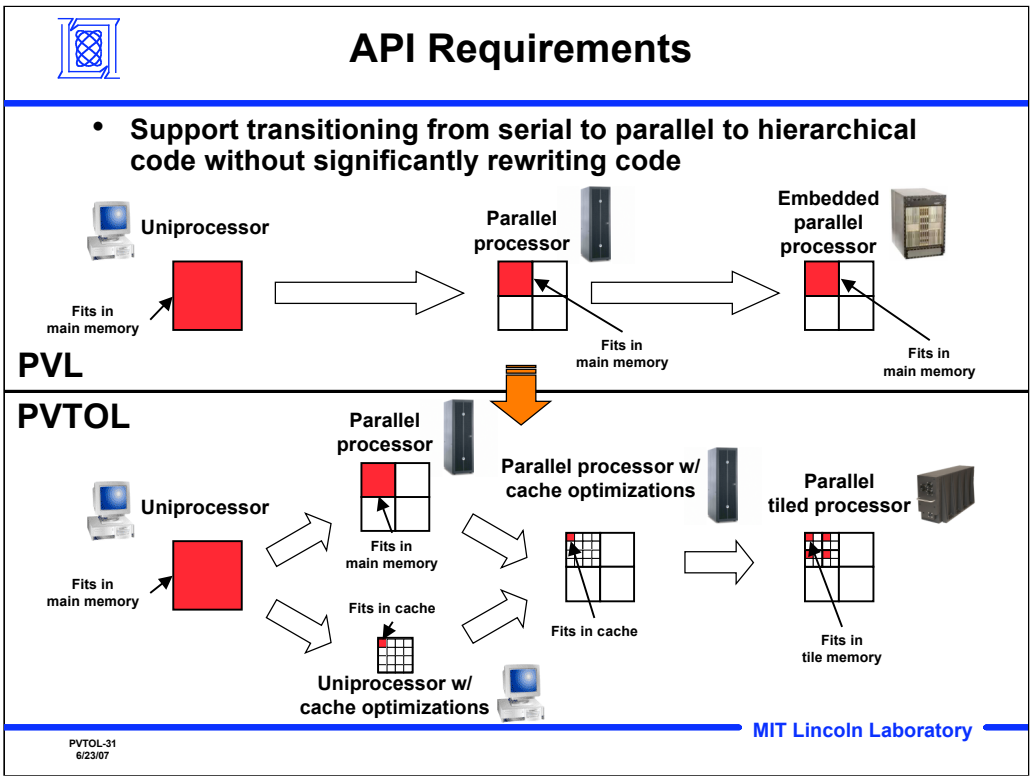
Iterators are used to access tiles that are distributed using temporal maps. The programmer can specify the direction the iterator moves through the tiles (row, column, plane-order) and how data should be transferred when accessing a tile, by specifying data management policies. For example, if the user intends to overwrite the contents of a tile, it is not necessary to actually load the tile. Rather, the tile can be allocated directly in the SPE. This eliminates the overhead of actually transferring data from main memory to the SPE's local store.



This shows an example of data flow through a pulse compression operation using hierarchical arrays. CELL 0 is responsible for the Data Input Task, which loads one CPI into the system at a time. Next, CELL 0 transfers the CPI to CELL 1 to perform the Data Analysis Task, which performs the pulse compression. CELL 1 distributes the CPI's rows across two SPE's. Each SPE divides its section of the CPI into blocks of columns. After CELL 1 has finished pulse compressing the CPI, the results are sent to CELL 2. CELL 2 performs the Data Output Task, which writes out the pulse compressed CPI. This example processing multiple CPI's in a pipelined fashion.

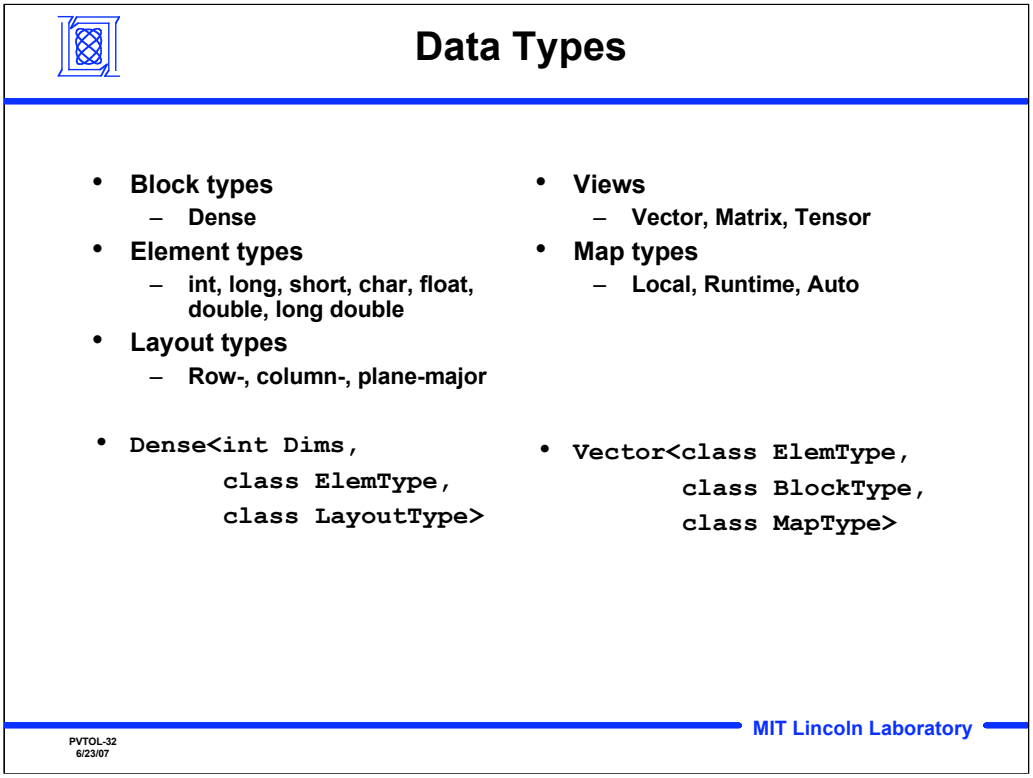

Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-30
6/23/07

Outline



PVTOL extends the incremental development process demonstrated in Lincoln's Parallel Vector Library (PVL). In PVL, programmers first develop a serial implementation of thee application on their workstation. Once the serial implementation is verified, the applications can be parallelized on a cluster. Because parallelization is achieved using maps, this requires very little change to the code. Once the parallel implementation is verified and optimized for performance, the application can be deployed onto the embedded parallel processor with little to no change to the code.

PVTOL adds an additional step to the process. As before, a serial version is developed on a workstation. Next, the programmer can either develop a parallel version on a cluster (a la PVL) or develop a serial version that is optimized for cache performance on a workstation. In both cases, maps are used to divide the data across processors or into blocks that fit into cache. Next, hierarchical maps are used to implement a parallel version in which data on each processor is optimized for cache performance. This is achieved by using hierarchical maps. Finally, the code can be deployed onto a parallel tiled processor, such that the hierarchical map partitions data between processors and between tiles on each processor.



Data is stored in Block objects, which will initially support dense data objects (as opposed to sparse). Data dimensions can be laid out in memory in any order. Block can store nearly any data type.

Data stored in Blocks are accessed via View objects, which represent vectors, matrices, or tensors. Additionally, data distributions can be specified in the View object using different types of Map objects. Finally, Views specify whether data is mapped to the local processor's memory, mapped at runtime across multiple processors, or automatically mapped by pMapper.

Data Declaration Examples

Serial

```
// Create tensor
typedef Dense<3, float, tuple<0, 1, 2> > dense_block_t;
typedef Tensor<float, dense_block_t, LocalMap> tensor_t;
tensor_t cpi(Nchannels, Npulses, Nranges);
```

Parallel

```
// Node map information
Grid grid(Nprocs, 1, 1, Grid.ARRAY); // Grid
DataDist dist(3);                    // Block distribution
Vector<int> procs(Nprocs);           // Processor ranks
procs(0) = 0; ...
ProcList procList(procs);            // Processor list
RuntimeMap cpiMap(grid, dist, procList); // Node map

// Create tensor
typedef Dense<3, float, tuple<0, 1, 2> > dense_block_t;
typedef Tensor<float, dense_block_t, RuntimeMap> tensor_t;
tensor_t cpi(Nchannels, Npulses, Nranges, cpiMap);
```


This shows an example of how to allocate serial data in PVTOL, then parallelize that data using maps. Each map requires a grid, data distribution and list of processor ID's. The grid describes how to partition each dimension of the data object. The data distribution describes whether to use a block, cyclic or block-cyclic distribution. The list of processor ID's indicates which processors to distribute data across. The map type in the Tensor datatype is changed to RuntimeMap and the map object is passed into the Tensor constructor.

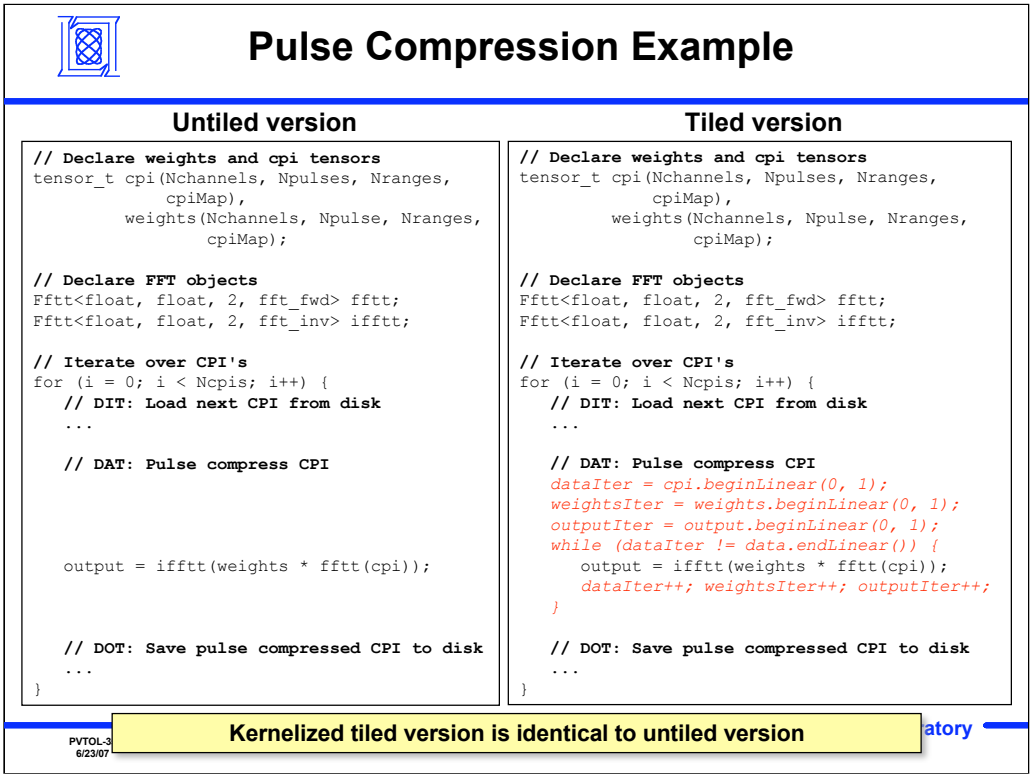
Data Declaration Examples

Hierarchical

```
// Tile map information
Grid tileGrid(1, NTiles 1, Grid.ARRAY);              // Grid
DataDist tileDist(3);                                // Block distribution
DataMgmtPolicy tilePolicy(DataMgmtPolicy.DEFAULT);   // Data mgmt policy
RuntimeMap tileMap(tileGrid, tileDist, tilePolicy); // Tile map

// Tile processor map information
Grid tileProcGrid(NTileProcs, 1, 1, Grid.ARRAY); // Grid
DataDist tileProcDist(3);                        // Block distribution
Vector<int> tileProcs(NTileProcs);               // Processor ranks
inputProcs(0) = 0; ...
ProcList inputList(tileProcs);                   // Processor list
DataMgmtPolicy tileProcPolicy(DataMgmtPolicy.DEFAULT); // Data mgmt policy
RuntimeMap tileProcMap(tileProcGrid, tileProcDist, tileProcs,
                       tileProcPolicy, tileMap); // Tile processor map

// Node map information
Grid grid(Nprocs, 1, 1, Grid.ARRAY); // Grid
DataDist dist(3);                    // Block distribution
Vector<int> procs(Nprocs);           // Processor ranks
procs(0) = 0;
ProcList procList(procs);            // Processor list
RuntimeMap cpiMap(grid, dist, procList, tileProcMap);  // Node map

// Create tensor
typedef Dense<3, float, tuple<0, 1, 2> > dense_block_t;
typedef Tensor<float, dense_block_t, RuntimeMap> tensor_t;
tensor_t cpi(Nchannels, Npulses, Nranges, cpiMap);
```


This shows an example of how to extend the parallel code on the previous slide to allocate hierarchical data using hierarchical maps. In this case, two additional layers are added to the Tensor object. The tile processor map describes how to distribute data on a processor between tile processors on each node. This is an example of a spatial map. The tile map describes how to distribute data owned by a tile processor into blocks that can be loaded into the tile processor's memory and processed. This is an example of a temporal map. Note that once a new map is created, it simply needs to be passed in as an argument to the next map constructor to create the hierarchical map.

The method of constructing these maps are nearly identical to the node map. One additional parameter is added, however: data management policies. Data management policies allow the programmer to specify how data should move between layers in the memory hierarchy, potentially eliminating unnecessary communication.



This is an example of how to implement pulse compression in PVTOL. The untiled version on the left runs on both serial and parallel processors. The tiled version on the right runs on tiled processors. On a tiled processor, data is distributed using temporal maps, i.e. data are broken into blocks that must be loaded one at a time. Consequently, iterators are used to load and process blocks one at a time.

Note that a pulse compression kernel could be written such that the library automatically recognize the pulse compression operation and automatically iterates through blocks in the hierarchical arrays. In other words, the iterator code in the tiled version would be hidden within the library. Consequently, such a “kernelized” version of the tile code would be identical to the untiled code.

Setup Assign API

- **Library overhead can be reduced by an initialization time expression setup**
  - **Store PITFALLS communication patterns**
  - **Allocate storage for temporaries**
  - **Create computation objects, such as FFTs**

Assignment Setup Example

```
Equation eq1(a, b*c + d);
Equation eq2(f, a / d);

for( ... ) {
    ...
    eq1();
    eq2();
    ...
}
```

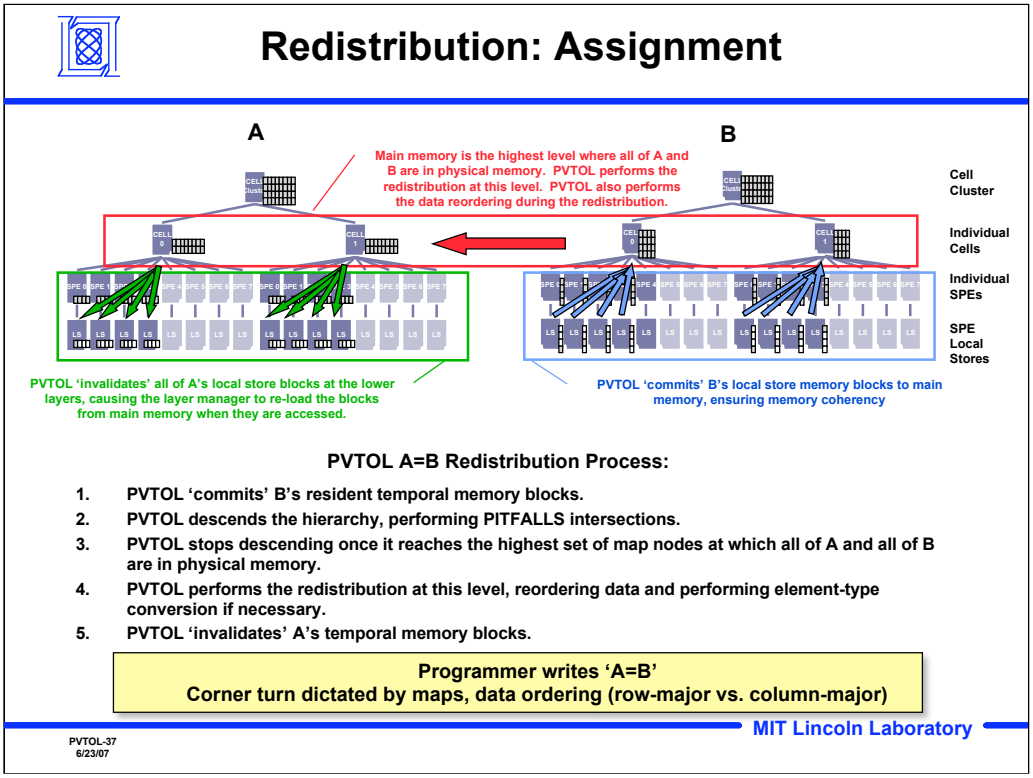
Expressions stored in Equation object

Expressions invoked without re-stating expression

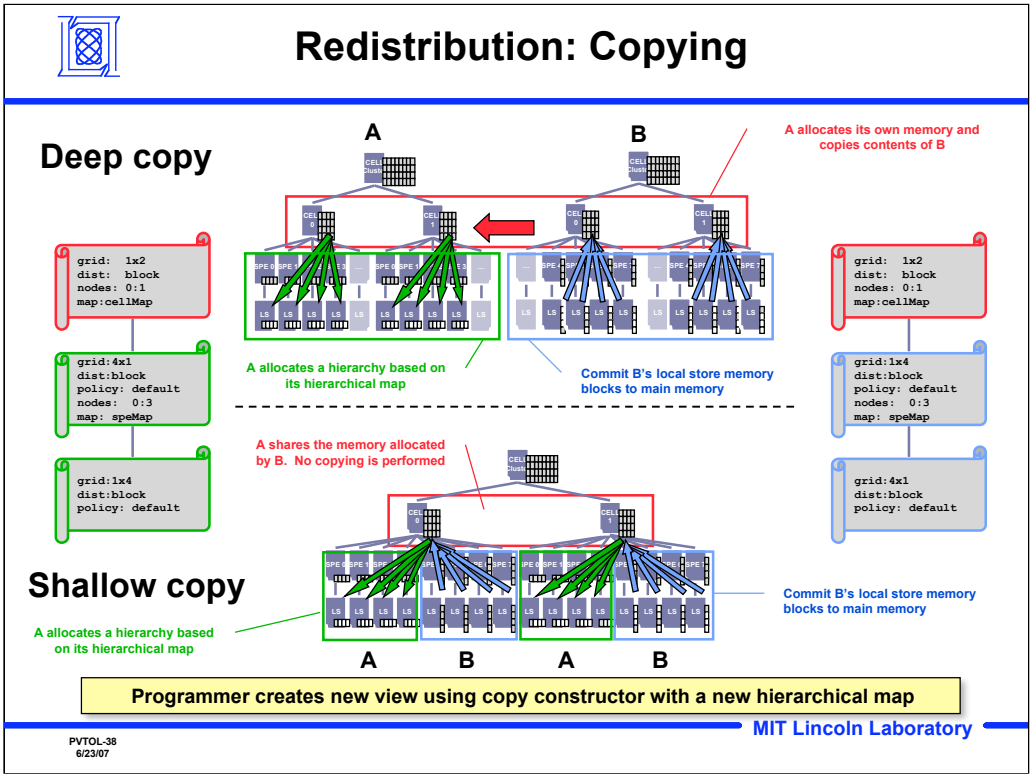
Expression objects can hold setup information without duplicating the equation

Executing expressions at runtime can incur various overheads, e.g. computing the PITFALLS communication patterns between operands with different mappings, allocating storage for temporary, intermediate values, and initializing values for certain types of computations like FFT.

Equation objects that describe expressions can be constructed. Thus, this overhead can be incurred at initialization time rather than during the computation.



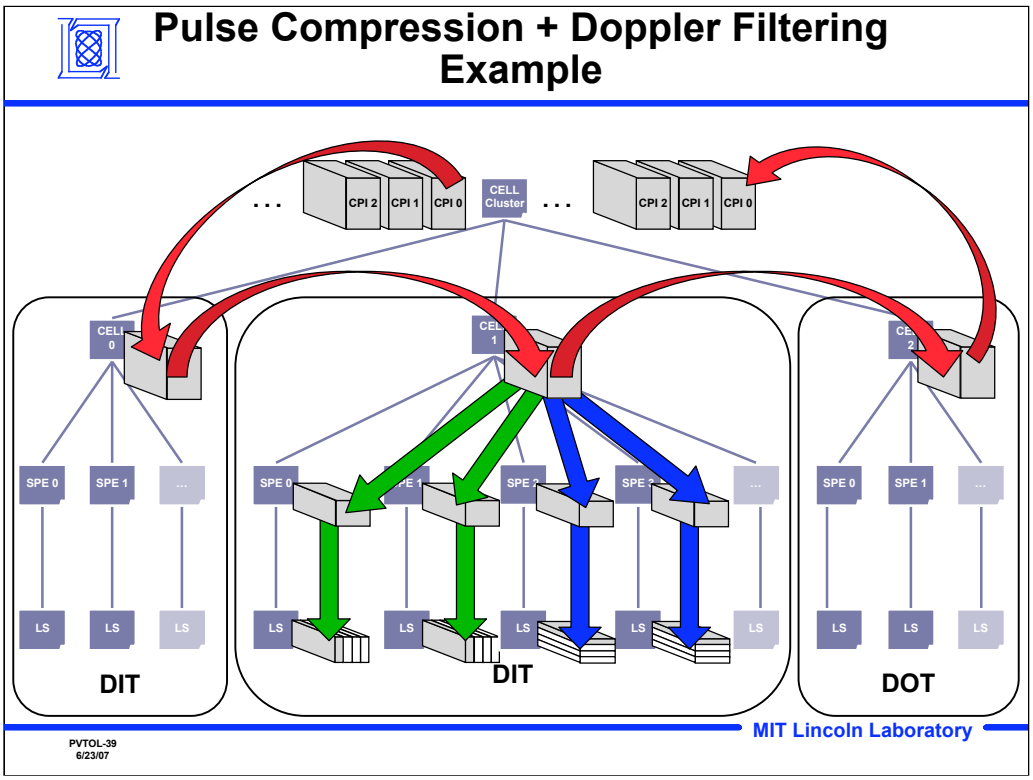
Often, the output data of an operation must have a different data distribution than the input data. A common type of operation which redistributes data is the corner turn, in which data that are distributed into blocks of columns are redistributed into blocks of rows, or vice versa. To perform a corner turn on a tiled processor, such as the Cell, all SPE's on each Cell processor write their respective columns of the array to main memory (blue). The Cell processors then redistribute the data in main memory from a column distribution to a row distribution (red). Finally, the rows on each Cell are distributed across the SPE's (green). To perform a corner turn, the programmer simply writes A=B, where A and B have different maps.



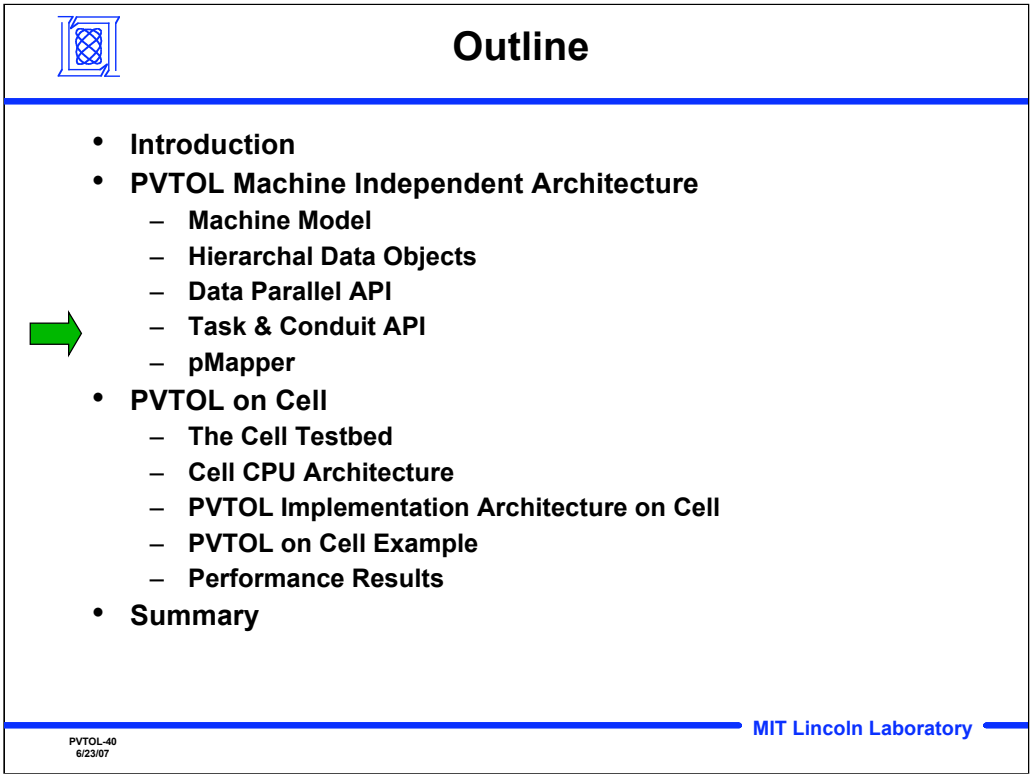
In the assignment A = B, depending on if A and B are on different processors, a deep copy is performed. If A and B are on the same processor, then the programmer can specify that A is a shallow copy of B.

If A and B are on different processors, the data on each SPE for B is written to main memory, then the contents of B are written into A, then the contents of A are distributed onto the SPE's according to A's hierarchical map.

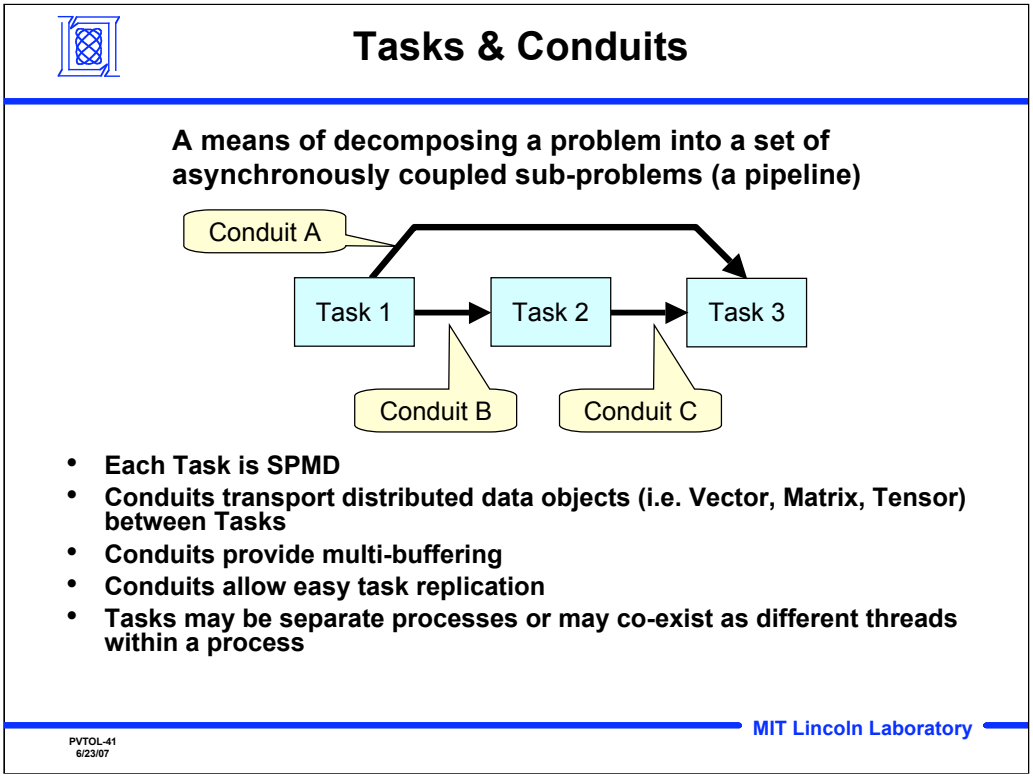
If A and B are on the same processor, but distributed on different SPE's, then A and B can share the same data in main memory, but distribute the data onto different sets of SPE's.



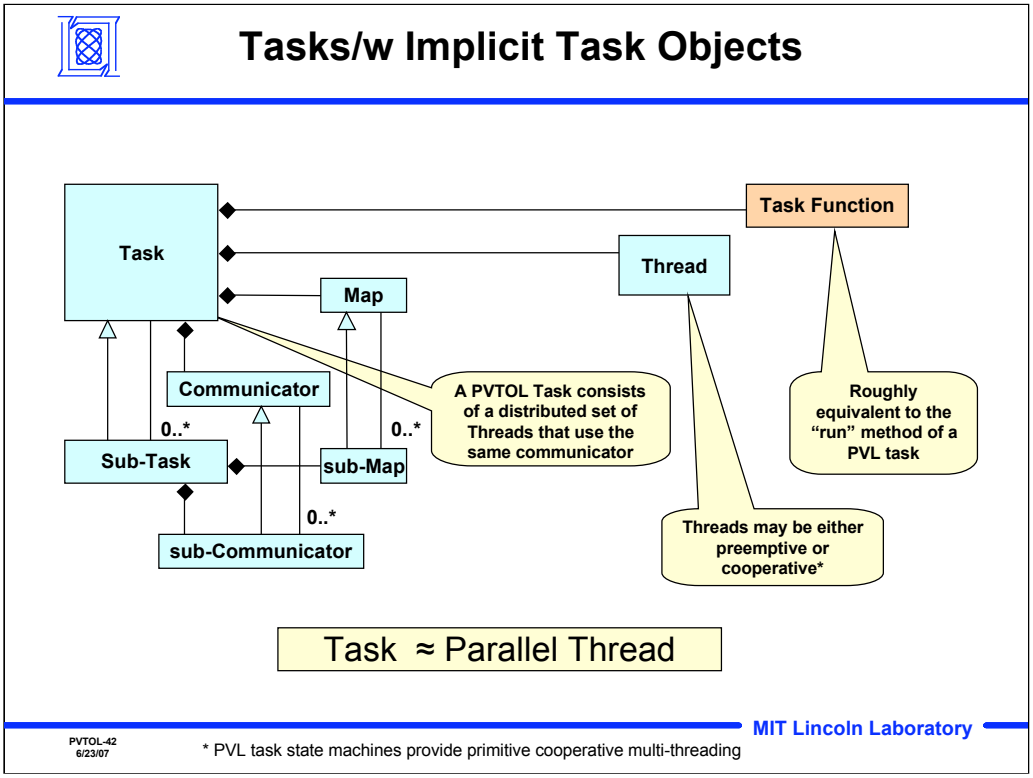
This shows an extension of the previous example of data flow through a pulse compression operation with the addition of a Doppler filtering operation. CELL 0 loads one CPI into the system at a time. Next, CELL 0 transfers the CPI to CELL 1 to perform the Data Analysis Task. CELL 1 distributes the CPI's rows across SPE 0 and SPE 1. Each SPE divides its section of the CPI into blocks of columns. Once the pulse compression is complete, the CPI is redistributed across SPE 2 and SPE 3 such that rows are still distributed across the SPE's but each SPE divides its section into blocks of rows. This redistribution is necessary because the pulse compression requires a different data distribution than the pulse compression. Once the DAT is complete, CELL 1 sends the processed CPI to CELL 2, which writes out the CPI.


Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-40
6/23/07

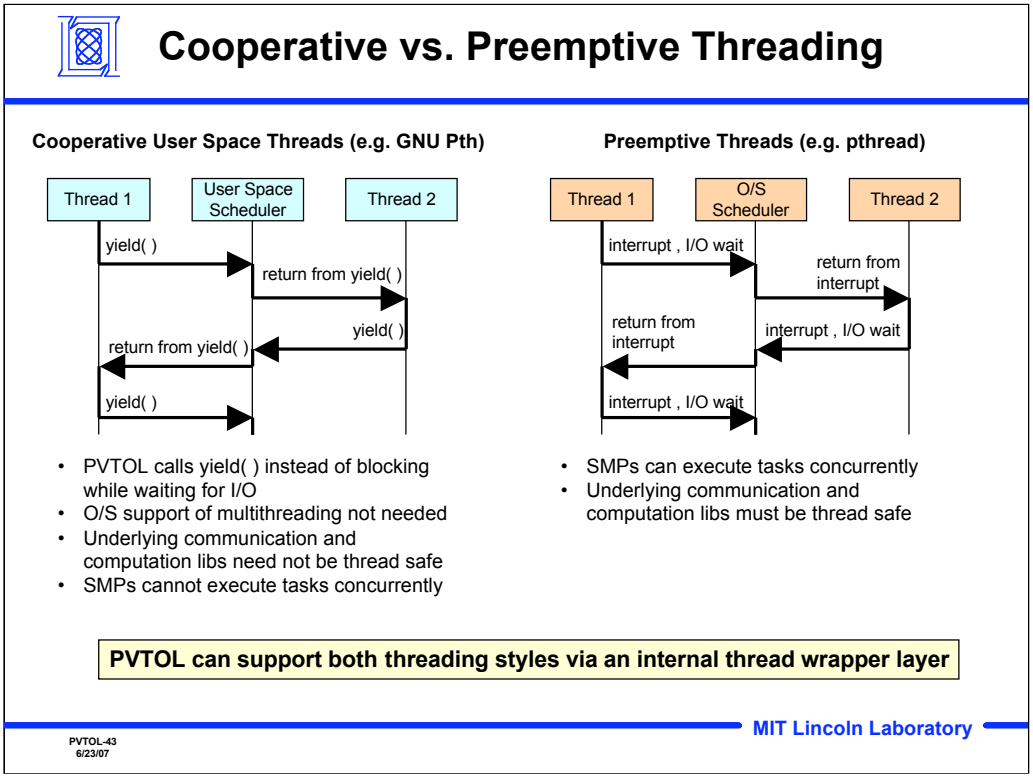
Outline



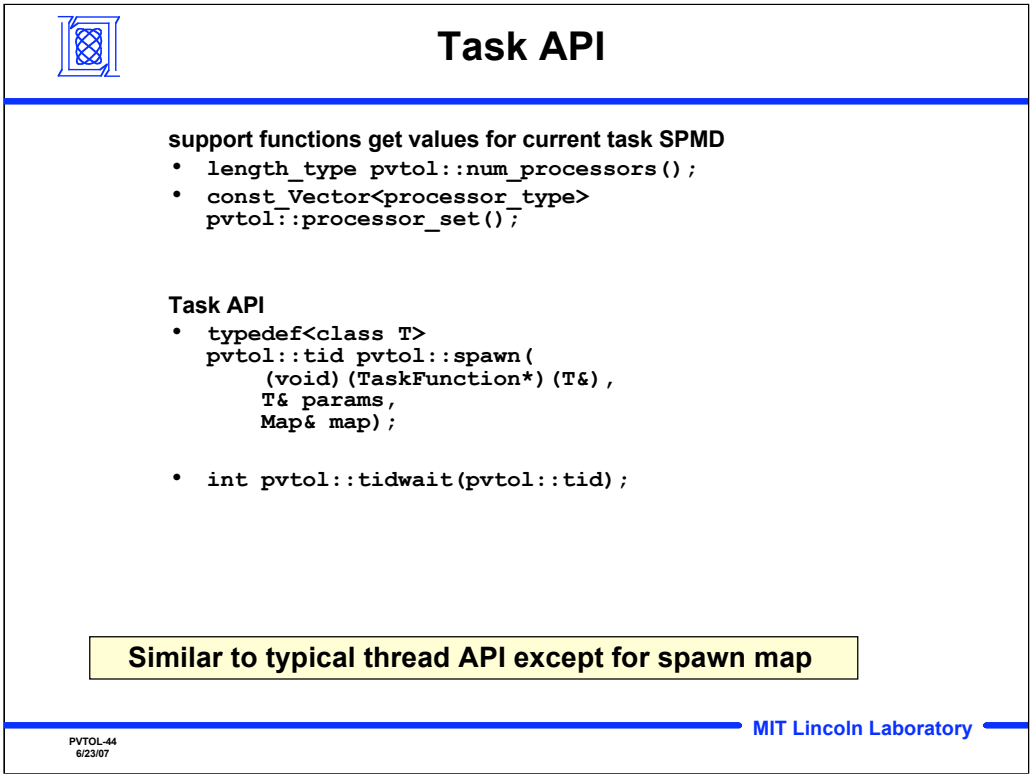
Each task in a pipeline is notionally an independent SPMD. Conduits transport distributed data objects between the tasks and isolate the mapping of data within a task from all other tasks.



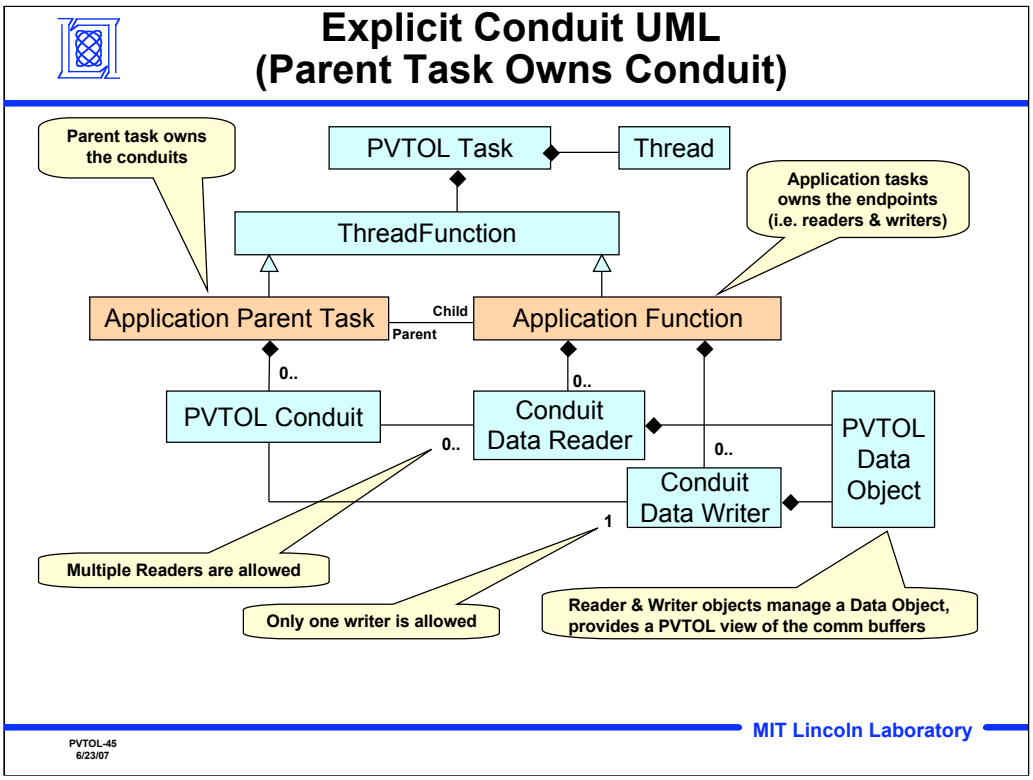
The execution of code in the different CPU ranks that a task is mapped to must happen in a SPMD manner and is therefore synchronous. Each task's SPMD need its own independent communication scope that is isolated from other tasks. In the same way that the execution of a native O/S thread implies a serialization of the threads actions, the execution of a SPMD task implies that each CPU executes SPMD actions in the same order.



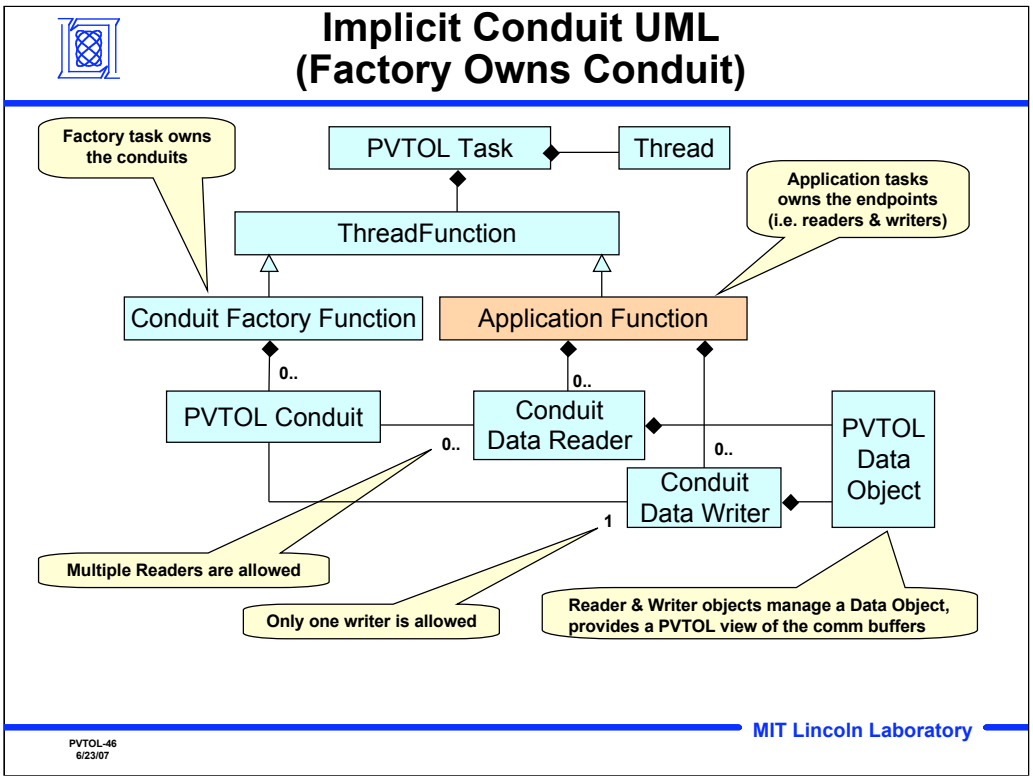
With cooperative threading a thread must explicitly yield execution back to a scheduler. With preemptive scheduling an interrupt can be used to force a thread to yield execution back to a scheduler. Most programs are in one of two major states. They are either waiting for I/O or processing data. Preemptive scheduling is mandatory only when compute bound processing can take so long that I/O can be missed. In real-time systems with bounded processing latency requirements this is rarely the case so either threading approach will usually succeed. Note that cooperative scheduling can be more efficient since preemptive context switching tends to add unnecessary overhead.



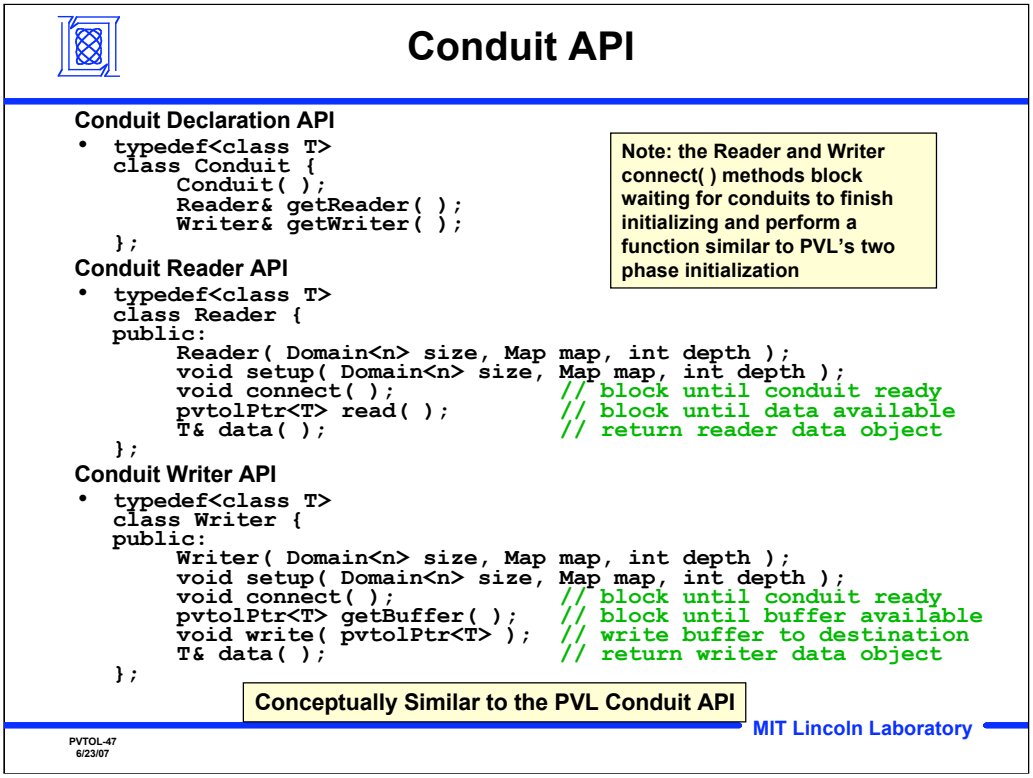
The proposed task API. Note the simplicity of this API vs. the PVL Task API.



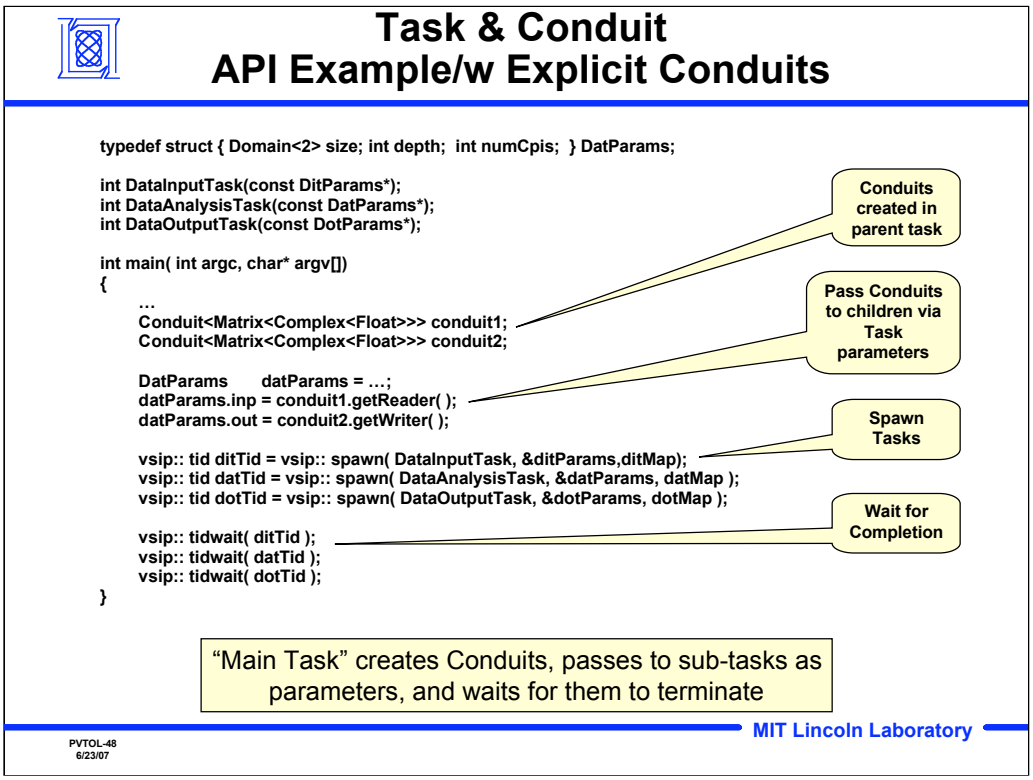
Class diagram of an explicit conduit case. This case is explicit because a parent Task actually declares a Conduit object and passes the Conduit interface objects into its child Tasks. This is less than ideal since the communication between child Tasks is hard coded into the parent Task.



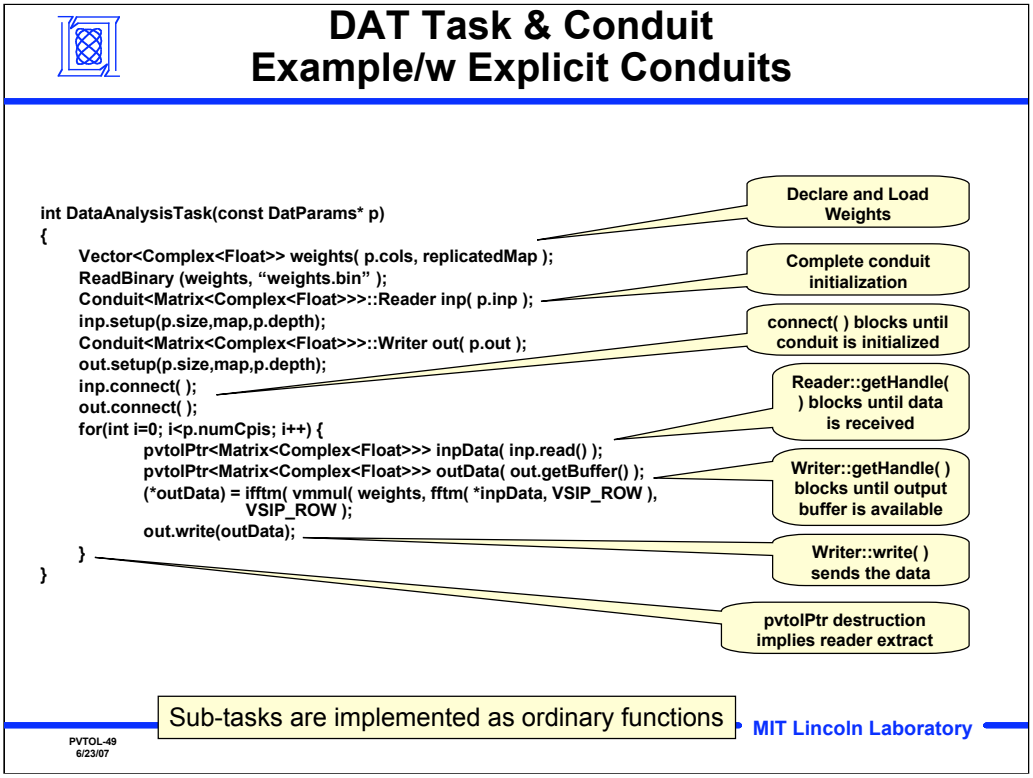
Class diagram of the implicit Conduit case. As in the explicit Conduit case a task still needs to own the Conduit. Instead of a parent Task the Conduit's owner is a Conduit factory Task. The Conduit factory Task is mapped across the entire application SPMD and therefore form any connection between Tasks. This approach is more flexible since tasks can connect to each other via a "conduit name" or "topic" rather than needing some parent task to explicitly form the connection.



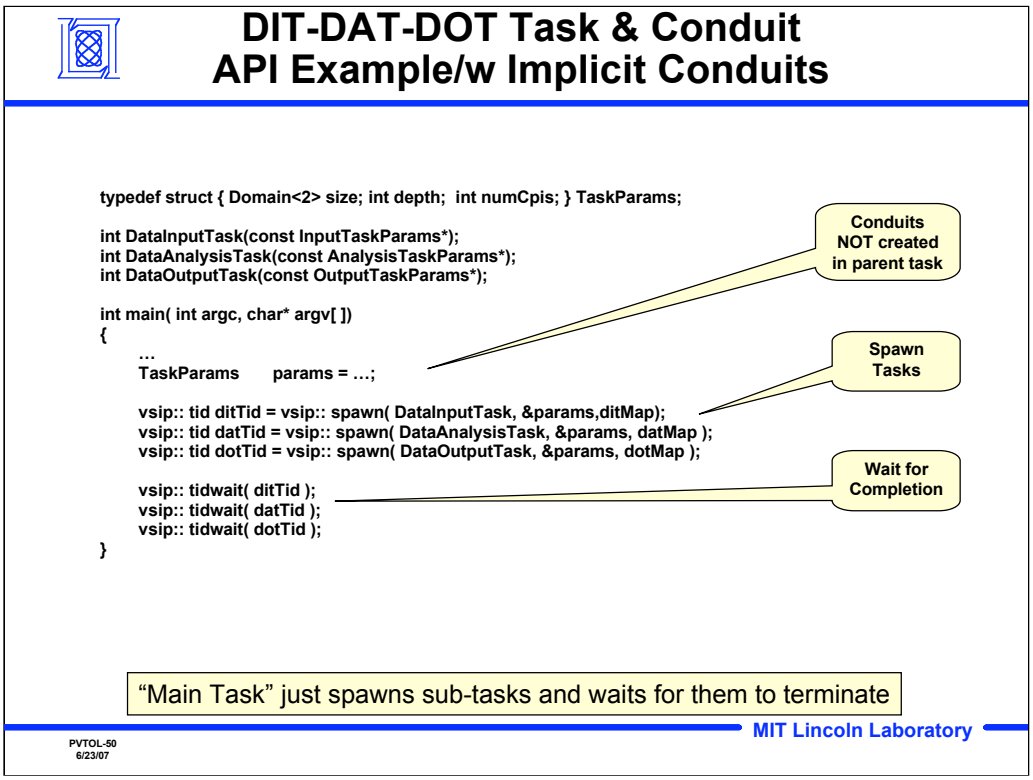
The Conduit API. The Conduit class is explicitly instantiated in an application Task in the explicit Conduit case or within a Conduit factory Task in the implicit Conduit case. Tasks that wish to send or receive data use the Reader or Writer classes to perform the required action.



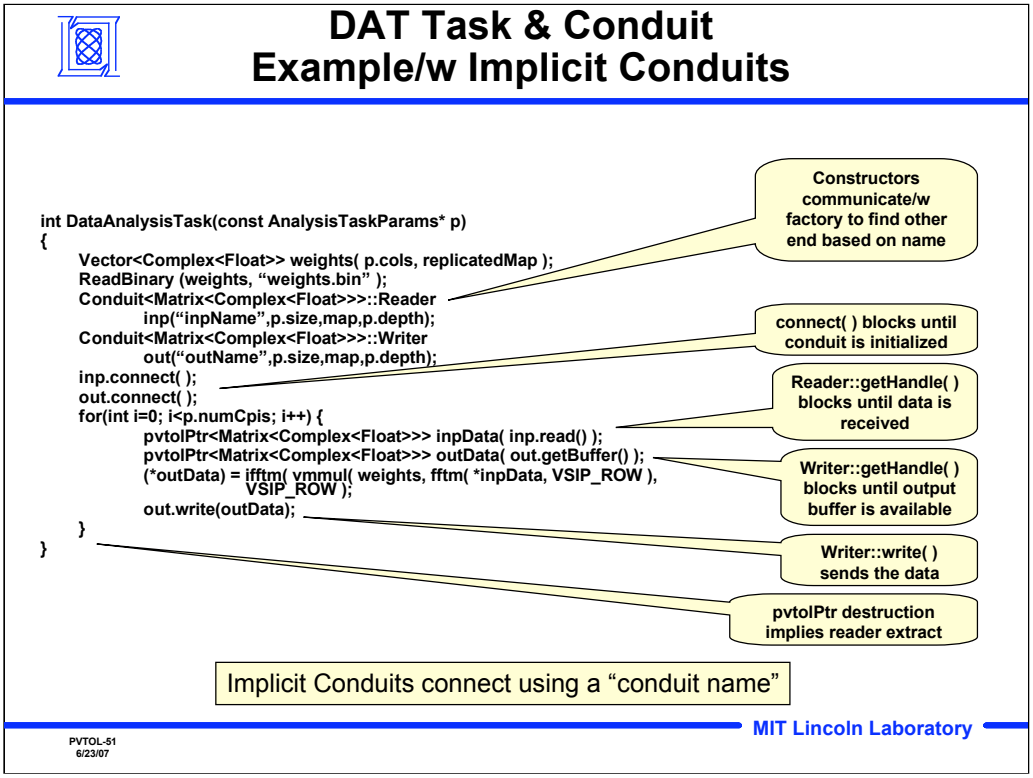
A simple explicit Conduit example, part 1. The parent Task declaration of the Conduit and the passing of conduit endpoint interfaces to the child tasks is shown.



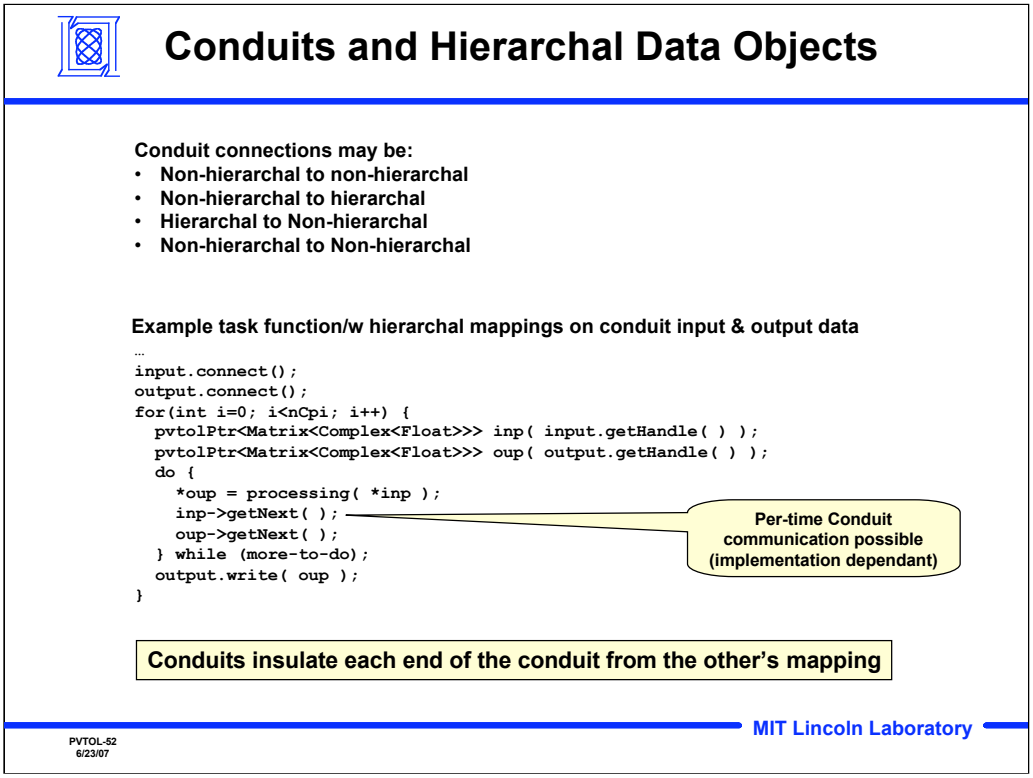
A simple explicit Conduit example, part 2. An implementation of the DataAnal child Task is show where the input and output Conduit interfaces are obtained from a parent Task.



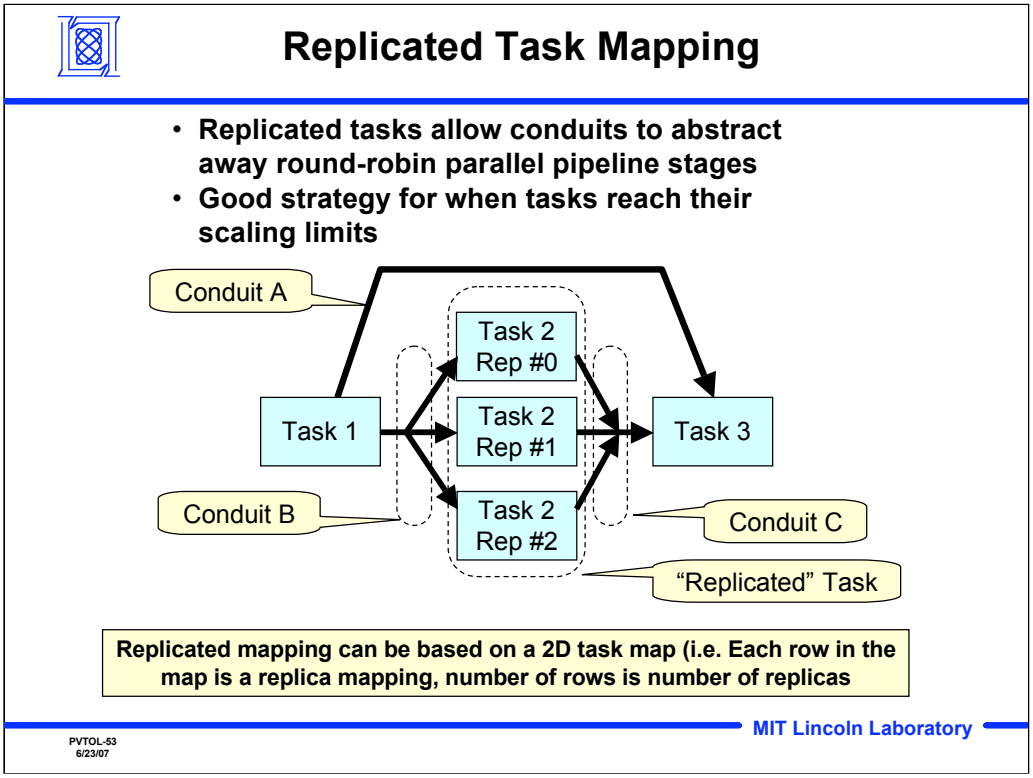
A simple implicit Conduit example, part 1. Since the child Tasks connect to each other without help from the parent Task the parent Task only needs to spawn off the child Tasks. Note that this code is nearly identical for the different child Tasks. For some classes of application, it should be possible to write a "generic parent" that spawns of the required child Tasks in a standard manner.



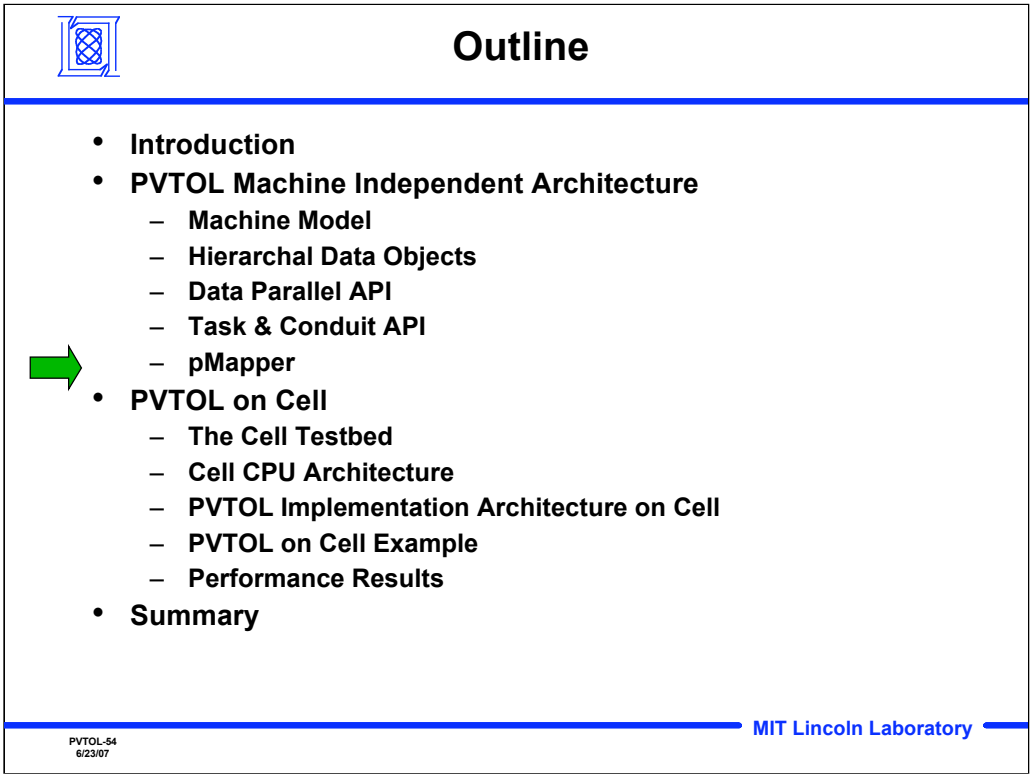
A simple implicit Conduit example, part 2. An implementation of the DataAnal child Task is shown where the Conduit endpoints are obtained from the Conduit factory.



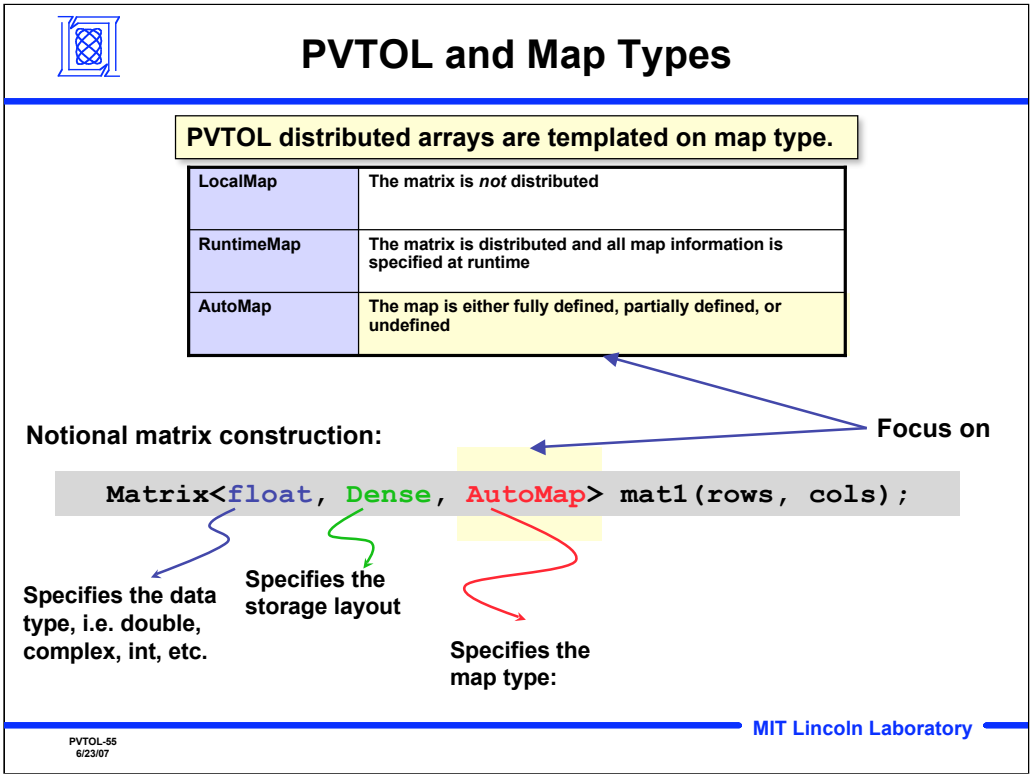
Hierarchal data objects have somewhat different access semantics than "normal" data objects. A conduit transports an entire hierarchal object which must be iterated with the hierarchal "getNext()" functions to access the objects data tiles.



Replicated Task mapping is needed in many applications. Replicated Task mapping is useful when additional throughput is needed but a Task has been scaled up to the reasonable limit of its data parallelism.

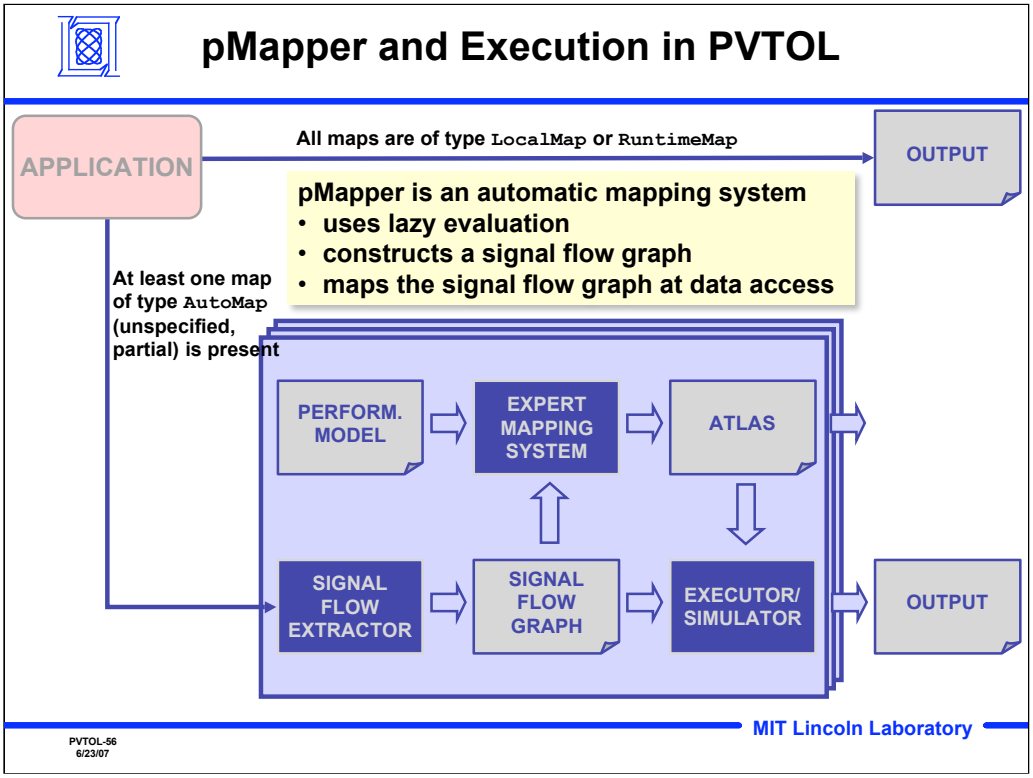

Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-54
6/23/07

Outline

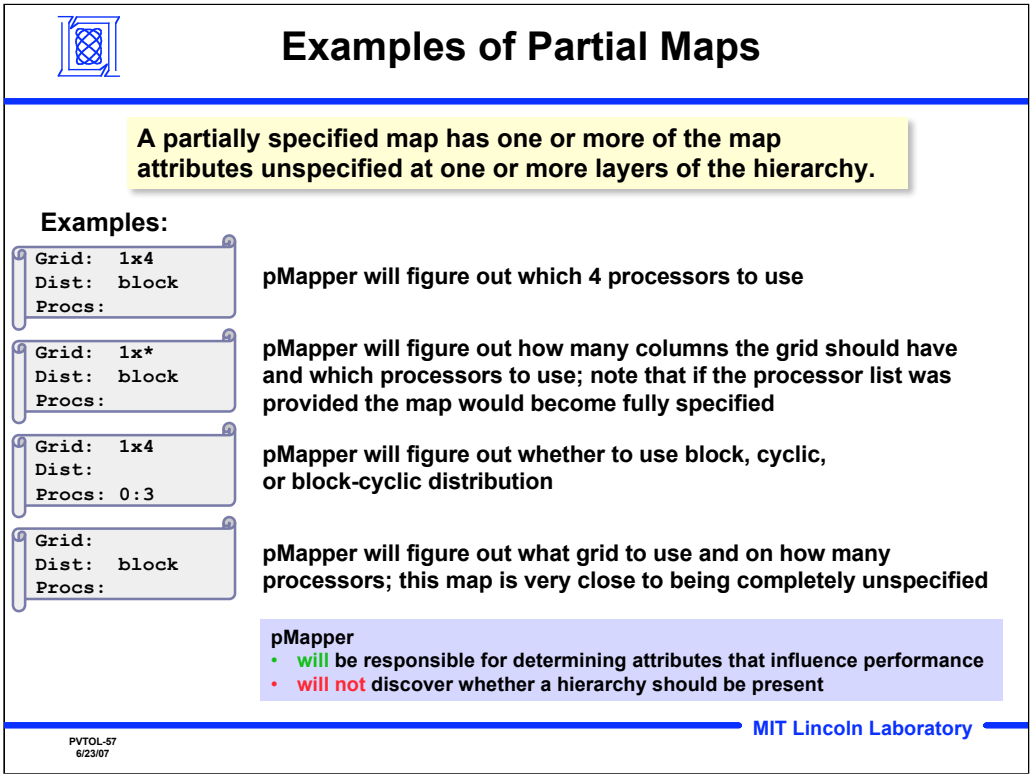


PVTOL supports three different types of maps. A LocalMap indicates that data is not distributed, but exists entirely in the processor's local memory. A RuntimeMap indicates that data is distributed across multiple processors and that the user specifies all map information. An AutoMap indicates that data may be distributed across multiple processors and that map information is fully or partially defined by the user or the map is undefined. When maps are partially defined or undefined, PVTOL automated mapping technology, pMapper, will do the following:

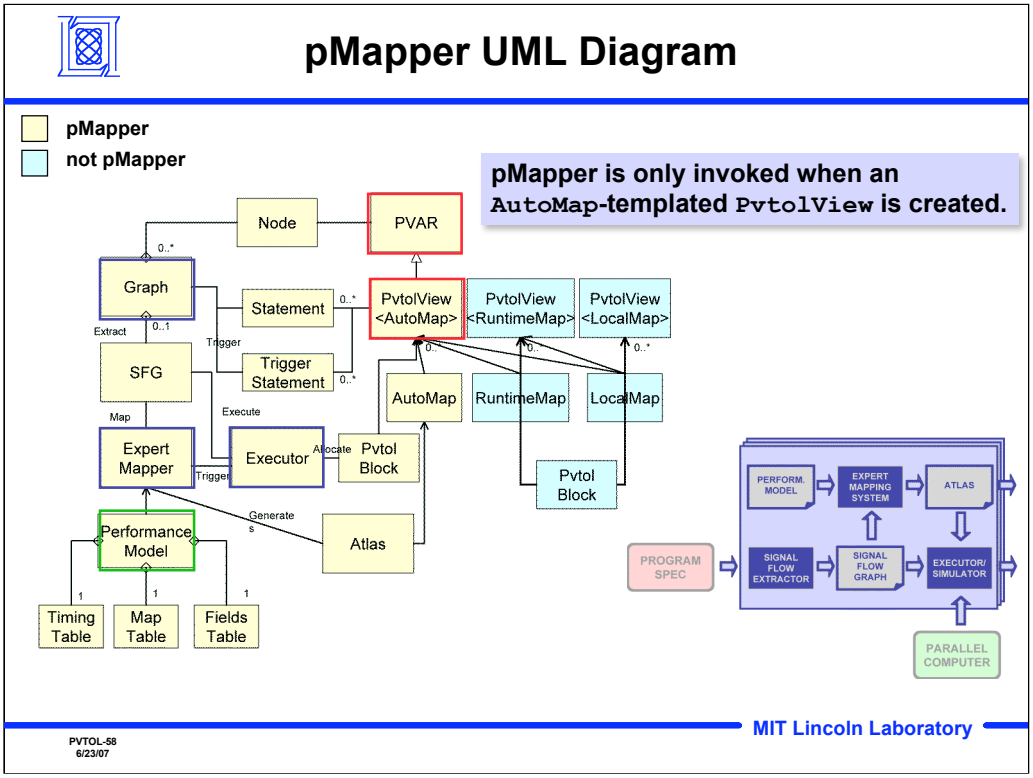
- pMapper will fill in the missing details of an hierarchical map (grid, distribution)
- pMapper will not discover whether an hierarchical map should be present
- pMapper will be responsible for discovering overlap
- pMapper will not be responsible for discovering the storage level
- pMapper will be responsible for determining some data management policies



This shows a flowchart of how data is mapped by PVTOL. When all maps are LocalMaps or RuntimeMaps, the map information is directly used to allocate and distribute arrays. If at least one map is an AutoMap, the automated mapping system in invoked, which will determine optimal map information.



This slides shows examples of partial maps. In short, pMapper will be responsible for determining attributes that influence performance, e.g. number of processors (Procs); block, cyclic or block-cycle distributions (Dist); the number of blocks to partition each dimension into (Grid); or any combination of these three attributes. Note that pMapper is NOT responsible for determining whether to distribute data across the processor hierarchy.



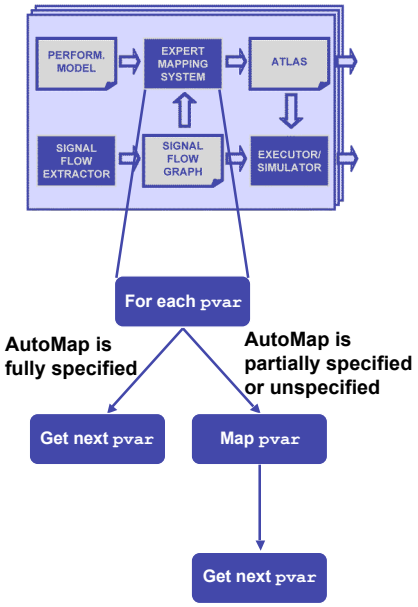
This slide shows the UML diagram for the different types of maps and pMapper. Note that pMapper is only invoked when a data object of type PvtolView is created that is templated on the AutoMap data type.

pMapper & Application

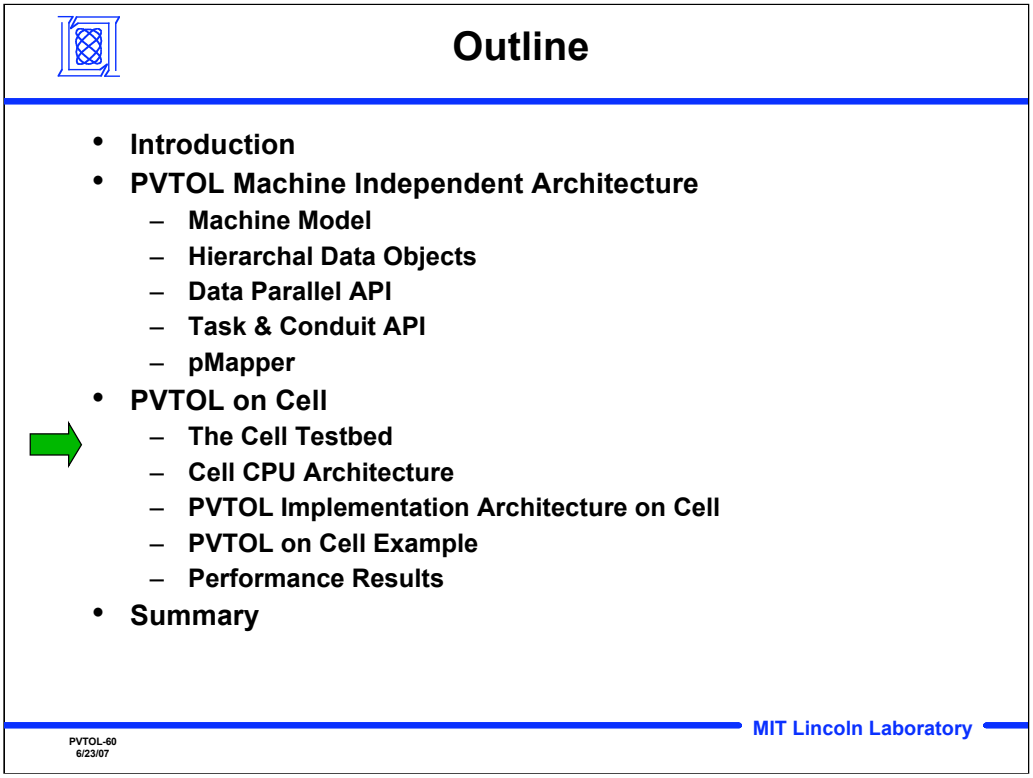
```
// Create input tensor (flat)
typedef Dense<3, float, tuple<0, 1, 2> > dense_block_t;
typedef Tensor<float, dense_block_t, AutoMap> tensor_t;
tensor_t input(Nchannels, Npulses, Nranges),
```

```
// Create input tensor (hierarchical)
AutoMap tileMap();
AutoMap tileProcMap(tileMap);
AutoMap cpiMap(grid, dist, procList, tileProcMap);
typedef Dense<3, float, tuple<0, 1, 2> > dense_block_t;
typedef Tensor<float, dense_block_t, AutoMap> tensor_t;
tensor_t input(Nchannels, Npulses, Nranges, cpiMap),
```

- For each `Pvar` in the Signal Flow Graph (SFG), pMapper checks if the map is fully specified
- If it is, pMapper will move on to the next `Pvar`
- pMapper will not attempt to remap a pre-defined map
- If the map is not fully specified, pMapper will map it
- When a map is being determined for a `Pvar`, the map returned has all the levels of hierarchy specified, i.e. all levels are mapped at the same time



This slides describes the logic PVTOL uses to determine whether or not to invoke pMapper. Two code examples are shown that construct a flat array and a hierarchical array. The first example that constructs a flat array specifies no map information. In this case, pMapper will determine all map information. In the second example that constructs a hierarchical array, because pMapper is not responsible for determining whether or not to allocate a hierarchical array, the programmer specifies a hierarchical map composed of two maps. Note that both maps contain no information. This indicates to pMapper that a two-level hierarchical array should be allocated and that pMapper is responsible for determining all map information for each level in the array.

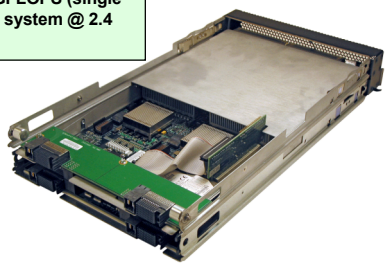

Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-60
6/23/07

Outline

Mercury Cell Processor Test System

Mercury Cell Processor System
- Single Dual Cell Blade
  - Native tool chain
  - Two 2.4 GHz Cells running in SMP mode
  - Terra Soft Yellow Dog Linux 2.6.14
- Received 03/21/06
  - booted & running same day
  - integrated/w LL network < 1 wk
  - Octave (Matlab clone) running
  - Parallel VSIPL++ compiled

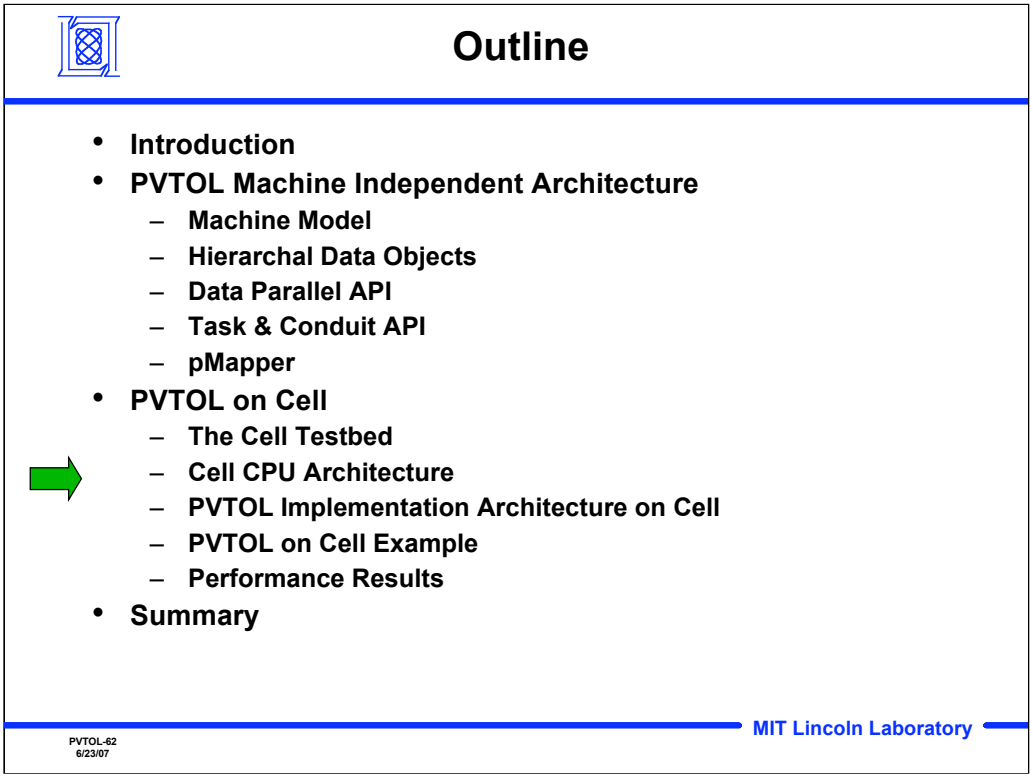
•Each Cell has 153.6 GFLOPS (single precision) – 307.2 for system @ 2.4 GHz (maximum)



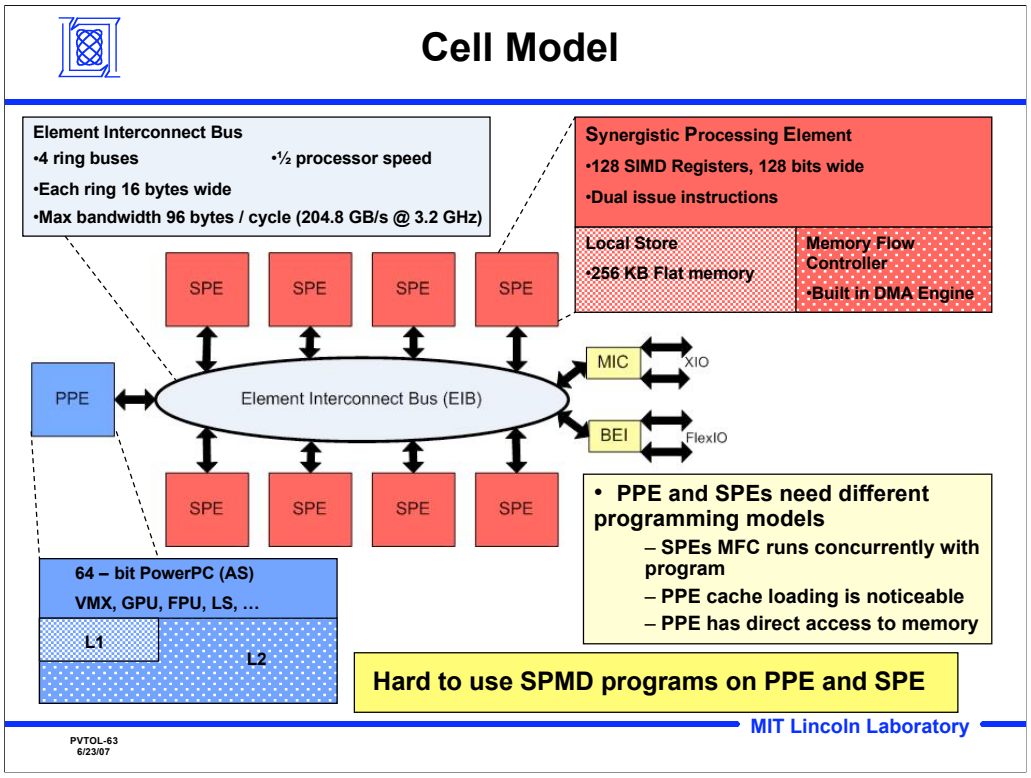
Software includes:
- IBM Software Development Kit (SDK)
  - Includes example programs
- Mercury Software Tools
  - MultiCore Framework (MCF)
  - Scientific Algorithms Library (SAL)
  - Trace Analysis Tool and Library (TATL)

This is a description of our original Cell test system. The 2.4 GHz blade was eventually replaced by a 3.2 GHz blade with 205 GFLOPS per cell or 410 GFLOPS per dual cell blade.

Picture of workstation is taken from Mercury web site. Picture of blade was supplied by Mercury.

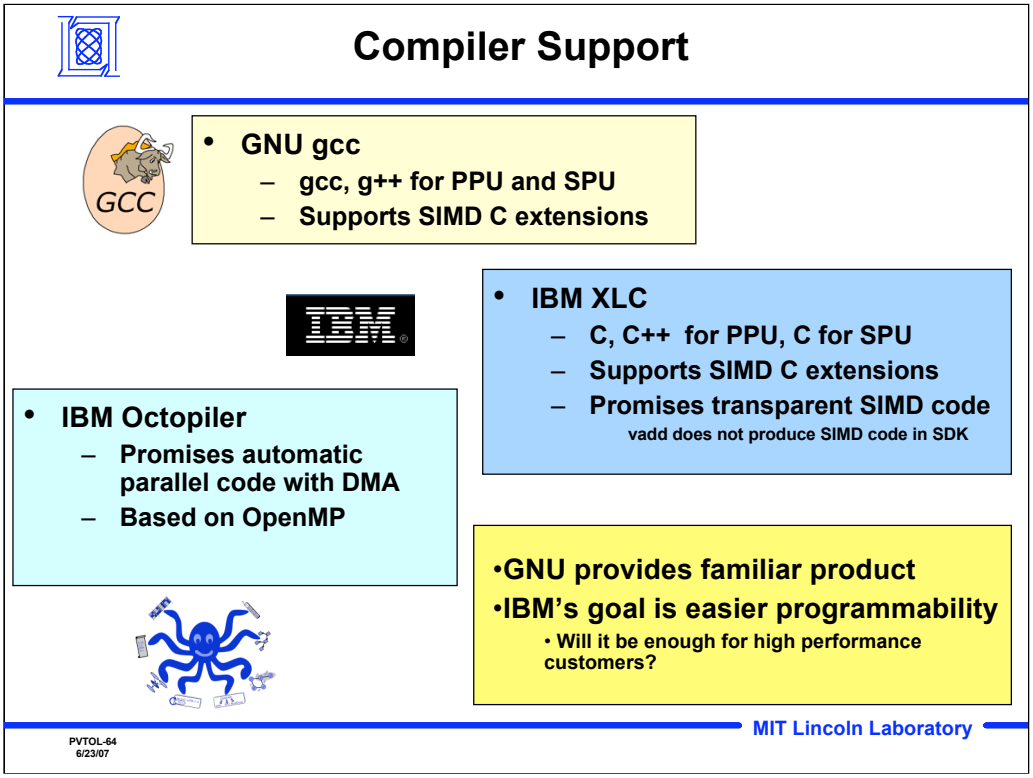

Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-62
6/23/07

Outline

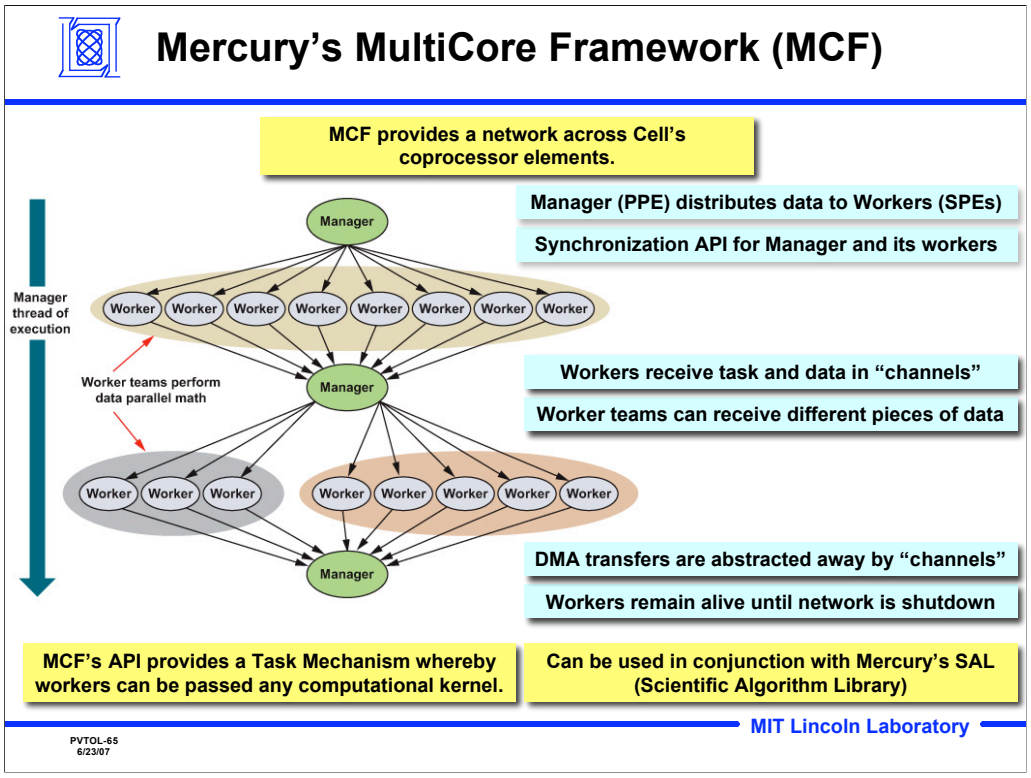


This is a block diagram of the STI Cell processor with details of some of the features.

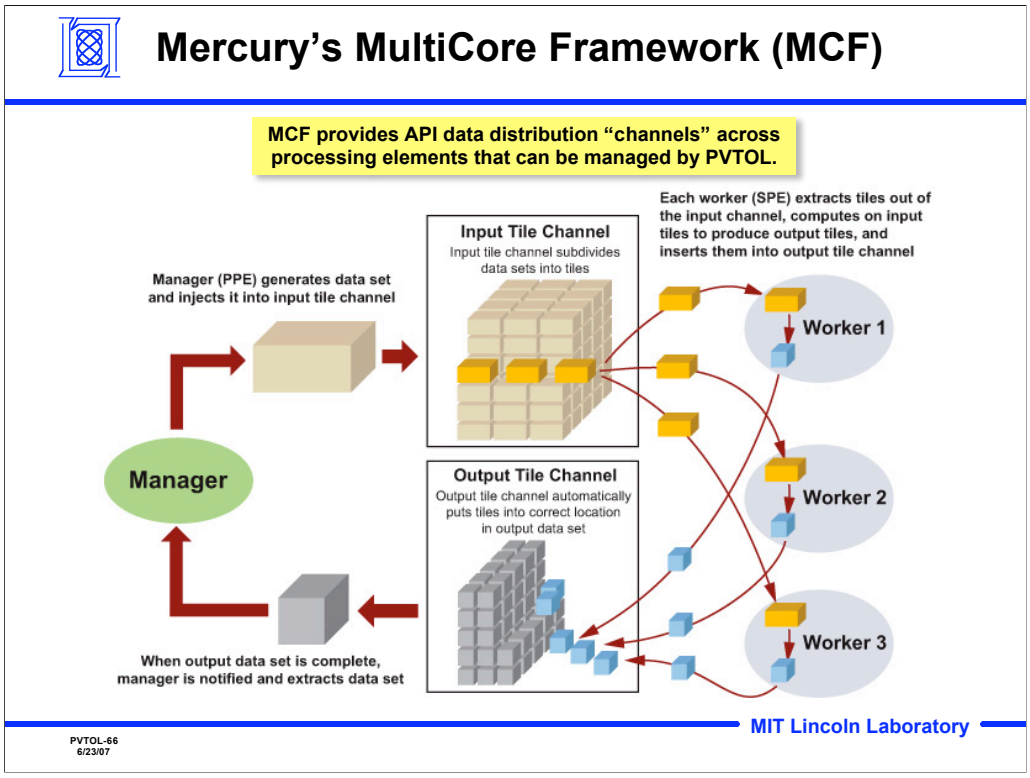
Max bandwidth taken from T. Chen et. al., *Cell Broadband Engine Architecture and its first implementation: A performance view*, IBM, 2005 .



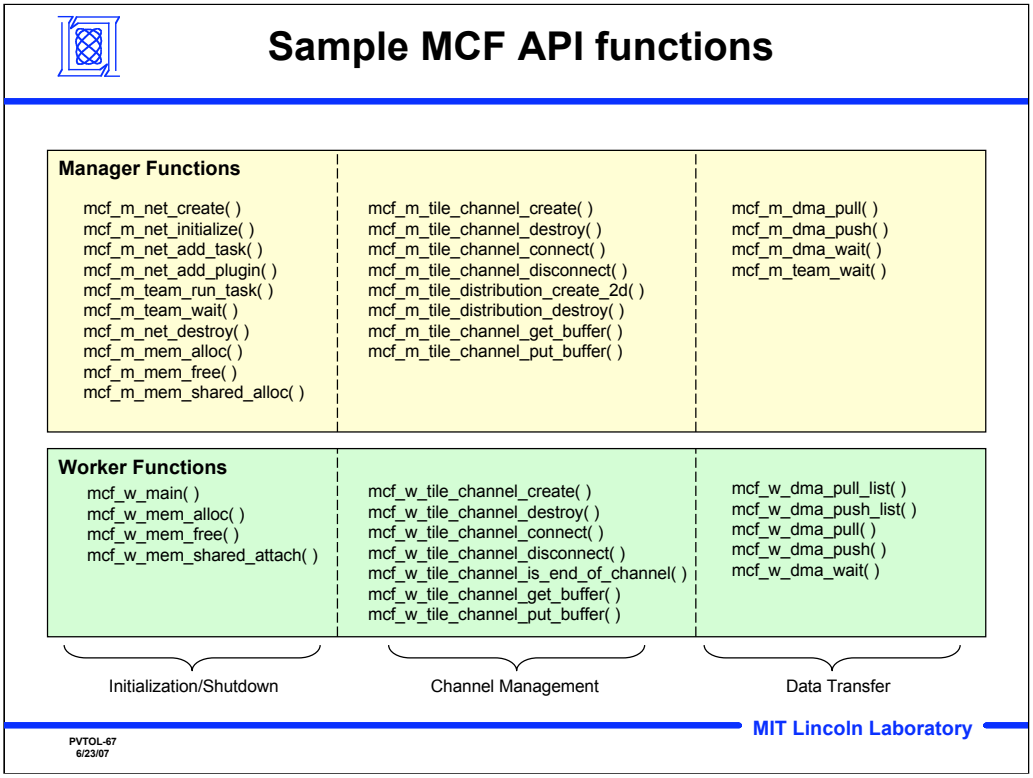
This is the state of the compilers as of Summer 2006. Both the GNU compiler and IBM's XLC are freely available and included in IBM's SDK releases.



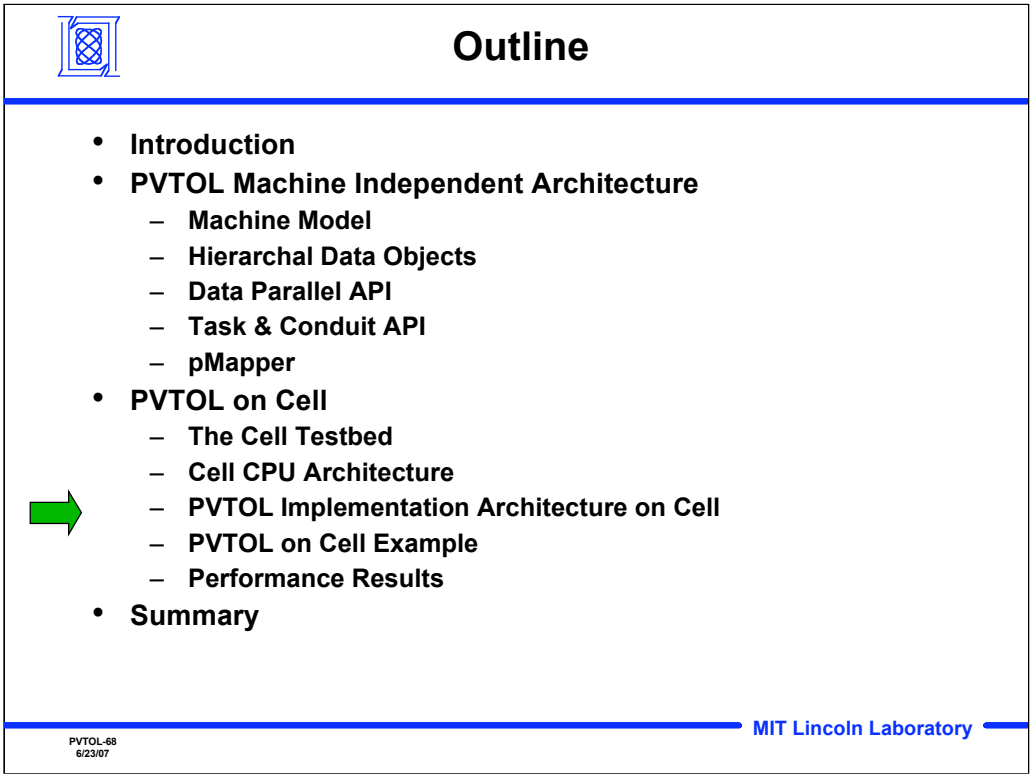
This slide shows the fundamental programming model of MCF. The advantage of MCF over threaded methods of control is that the workers have a small kernel that is started only once. In a threaded model, the worker is started again for each thread which has an expensive overhead.



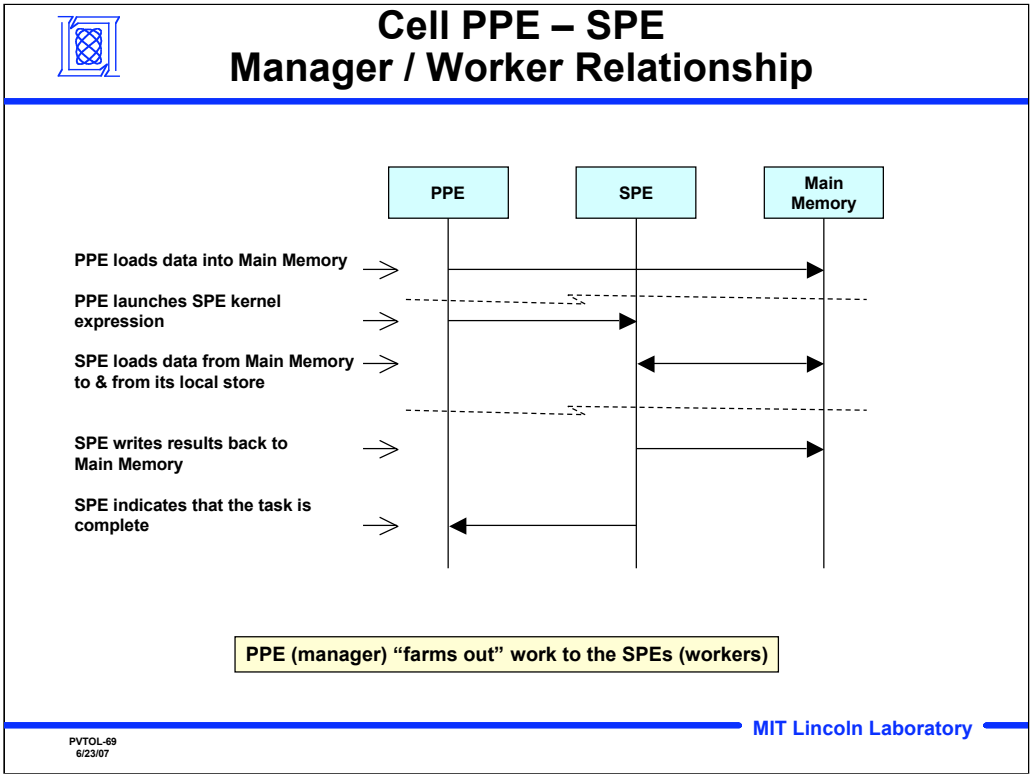
MCF also has built in data reorganization capabilities. The most common data partitioning methods used in parallel programming are encapsulated into function calls. This makes programming easier since the user does not have to reinvent the data reorganization over and over.



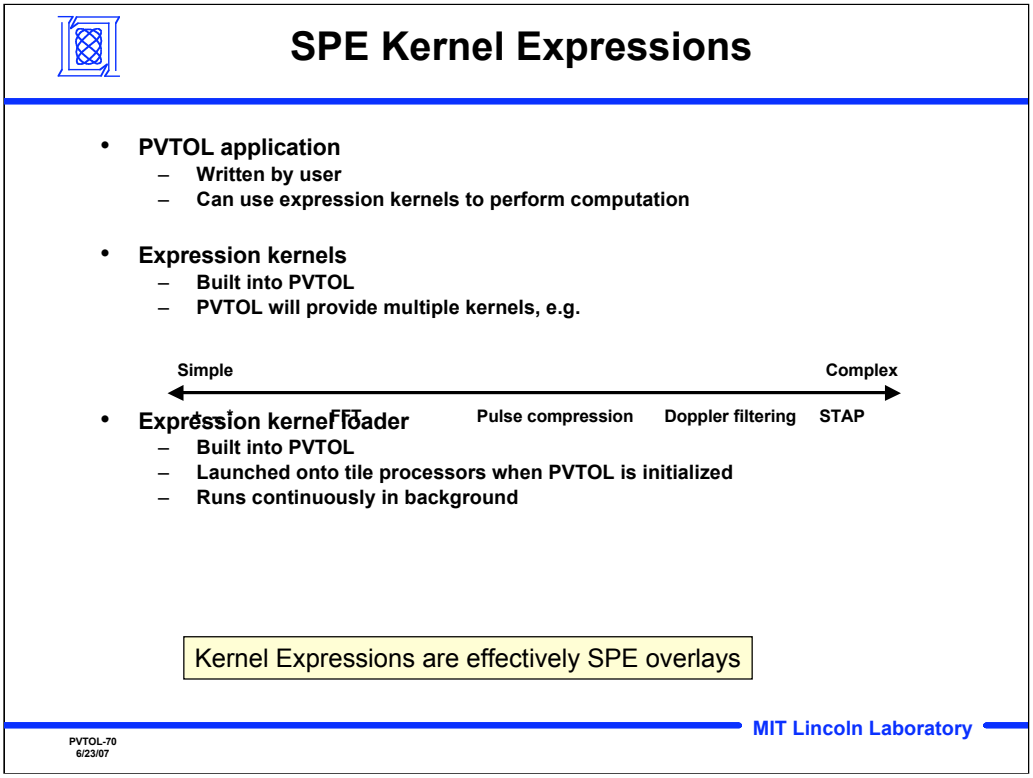
Here are some of the functions available in MCF. This list is not comprehensive, but is intended to give the viewer a flavor of the programming capabilities of MCF.


Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-68
6/23/07

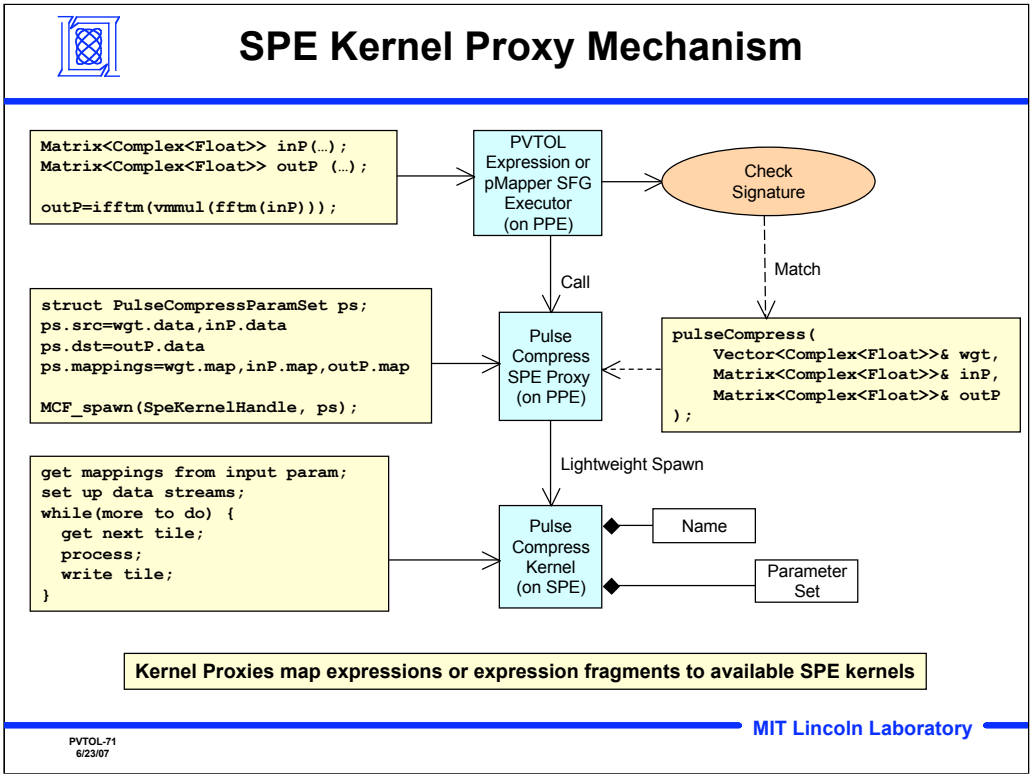
Outline



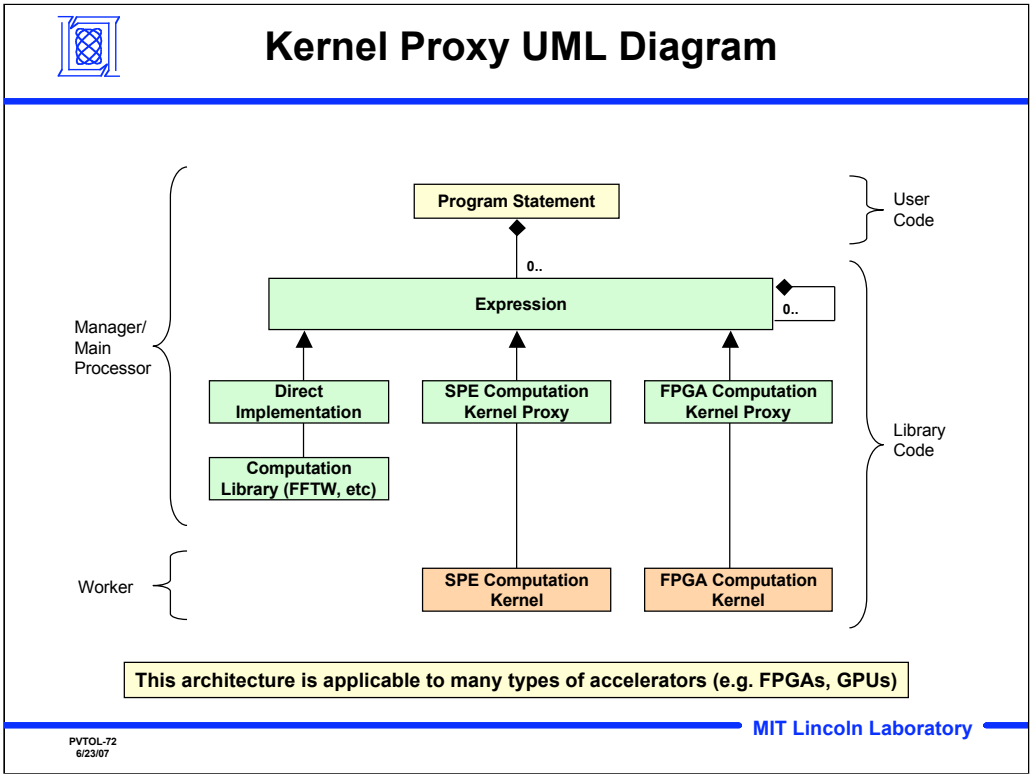
The cell consists of a general purpose processor (PPE) and a collection of 8 special processors (SPE). Each SPEs has its own small memory. The PPE is the manager which manages the SPE workers. The PPE launches a task on the SPE workers. In parallel, the SPEs load the data from main memory to their local memory, perform the computation and write the result back to main memory.



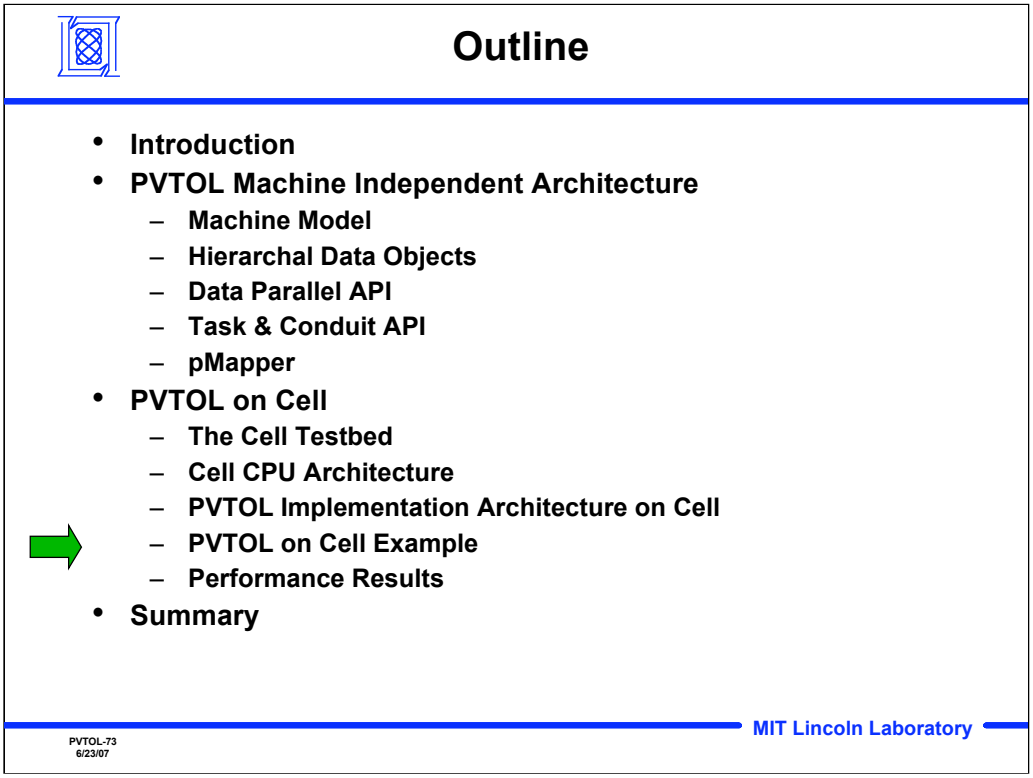
PVTOL library focuses on easy of use. The user writes an application which uses the pre-built kernels. The kernels are composed of a manager and a worker portion. In using the kernels, the user only needs to interact with the manager portion of the kernel.



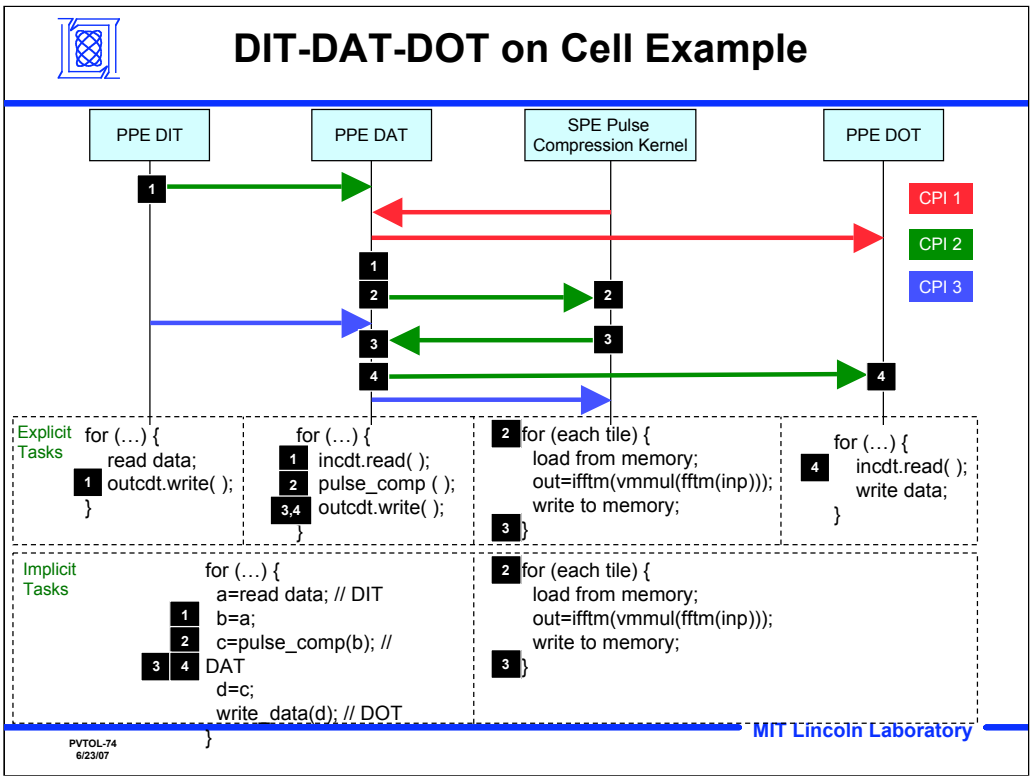
A pattern-matching scheme is employed to map user code to the available kernels. This mechanism is also useful for composing functions based on the kernels available in the PVTOL library.



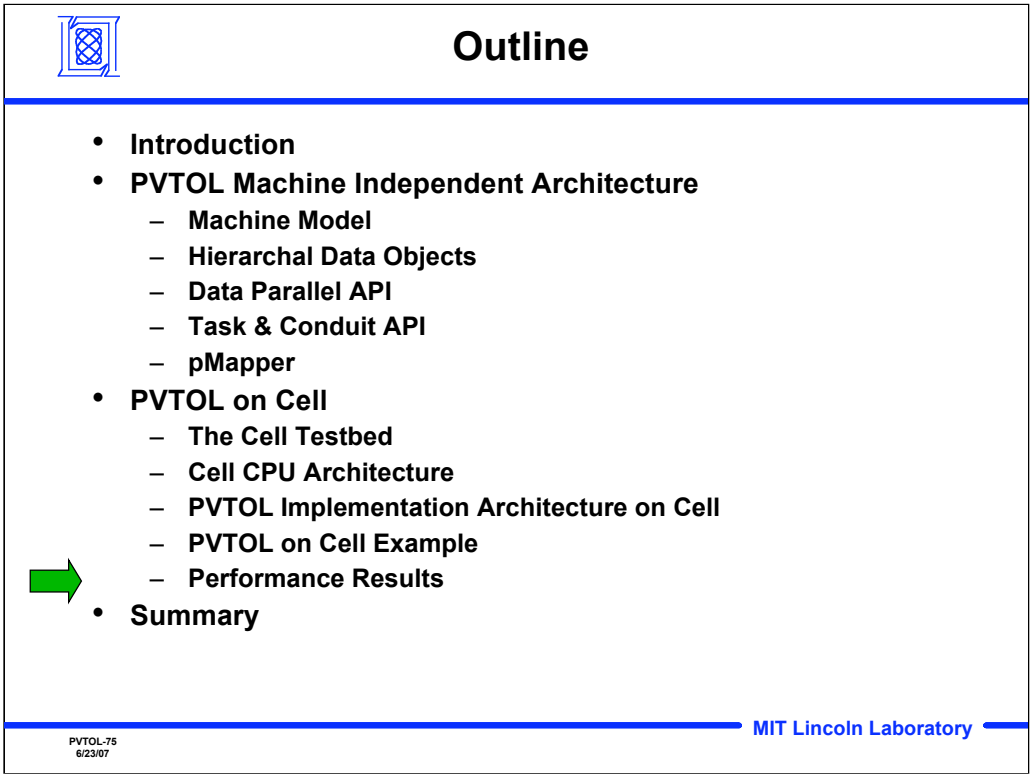
The software architecture is broadly applicable to a variety of hardware hosts. The same user code can run on the Cell, on the FPGA, or on the Grid. The PVTOL library API and the dispatch mechanism hides the implementation details from the user.


Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-73
6/23/07

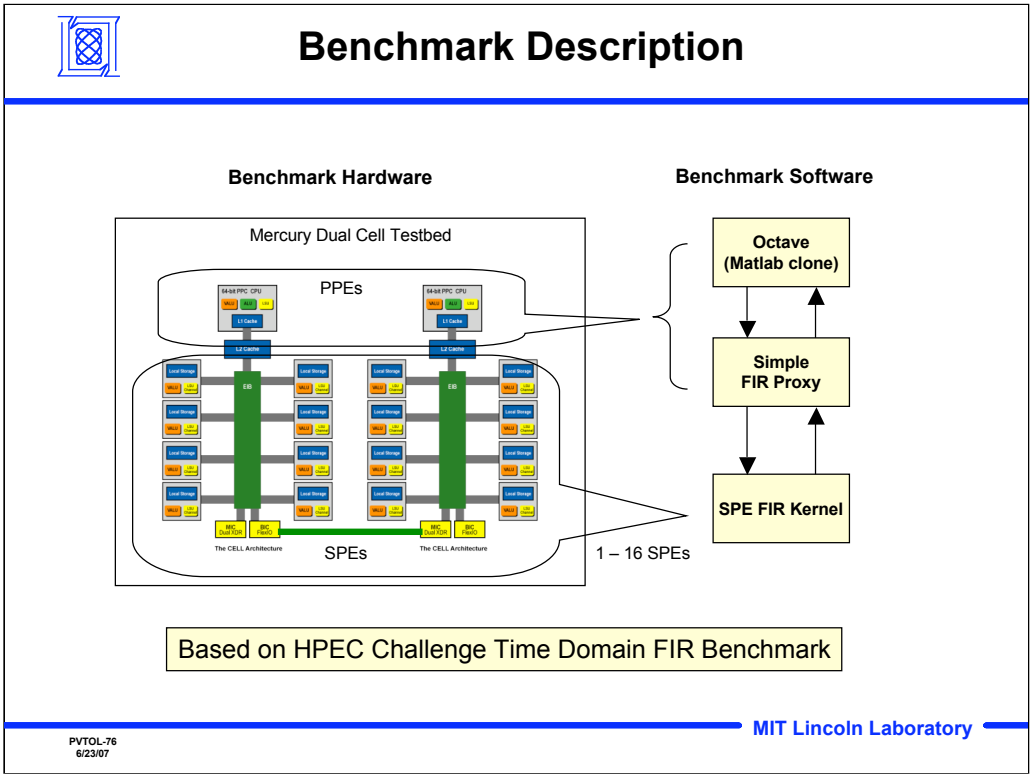
Outline



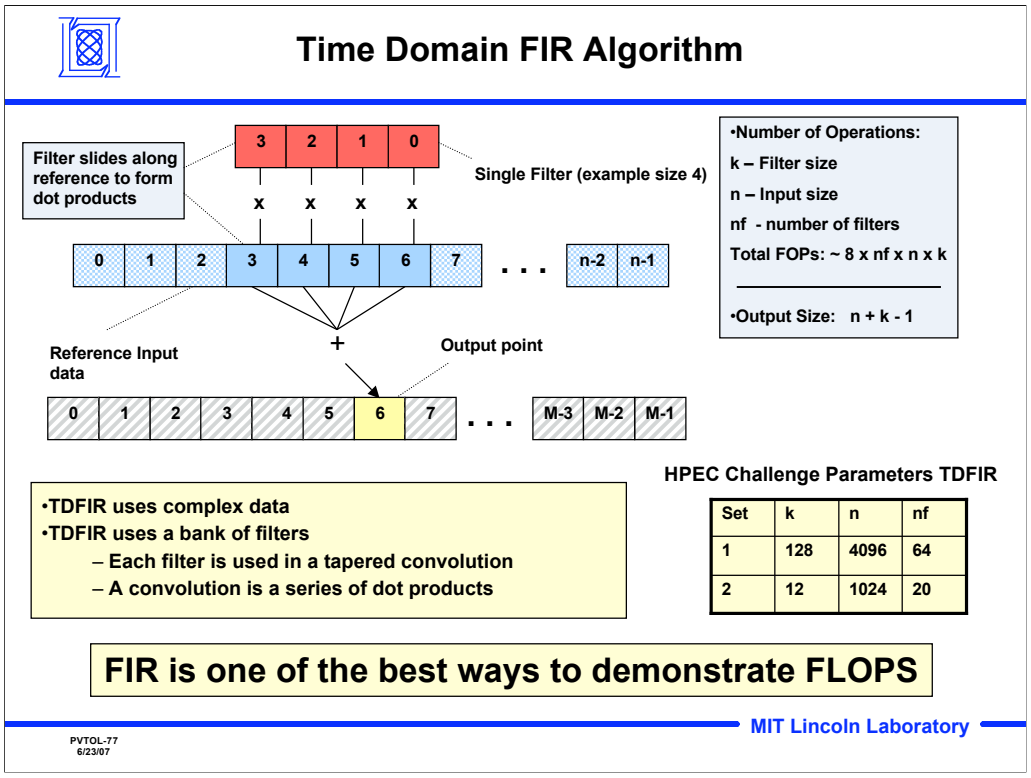
This slide shows the division of work and the sequence for a Data Input Task(DIT), Data Analysis Task(DAT) and a Data output Task (DOT). The DIT and the DOT are PPE only tasks. The DAT is managed by the PPE but the computationally expensive kernel runs in parallel on the SPEs


Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-75
6/23/07

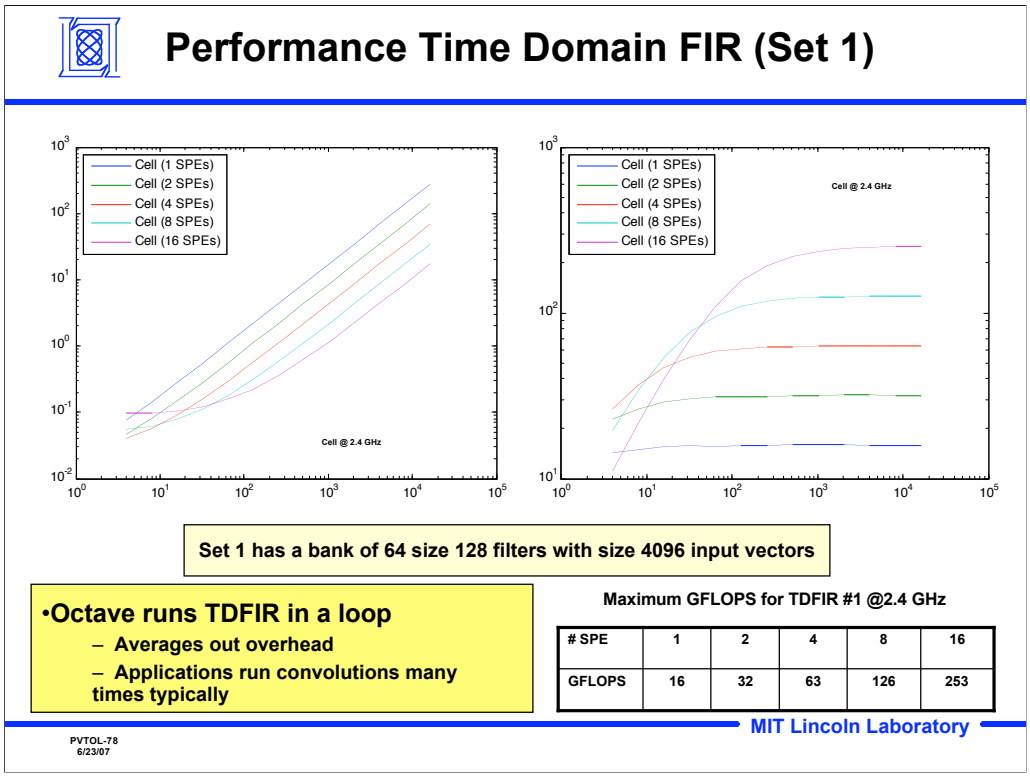
Outline



While the initial timings of the benchmark TDFIR were made local to a single SPU to determine how much performance could be expected from the computational units in a single SPU, we also need to understand how the TDFIR will perform at the chip level. Octave was chosen as the development environment since it is similar to MATLAB which is not available on Cell. From here it was relatively easy to take the high performance TDFIR for a single SPU and create the parallel version.

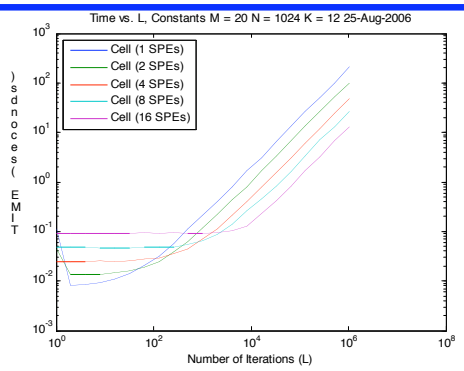


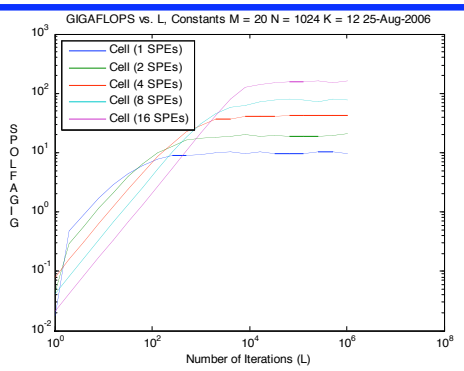
This is a graphical explanation of the TDFIR which uses complex data. Note that the kernel elements are in reversed order for convolution. Normal order gives correlation.



Here are timing results for TDFIR set 1 @ 2.4GHz. All timings include the full overhead of this application. Note that once the number of iterations become large enough, this application scales nicely with the number of processors. Cell DMAs have a limit of 16KB transfer size. Using the DMA list option is one way of getting around this limitation.

Performance Time Domain FIR (Set 2)





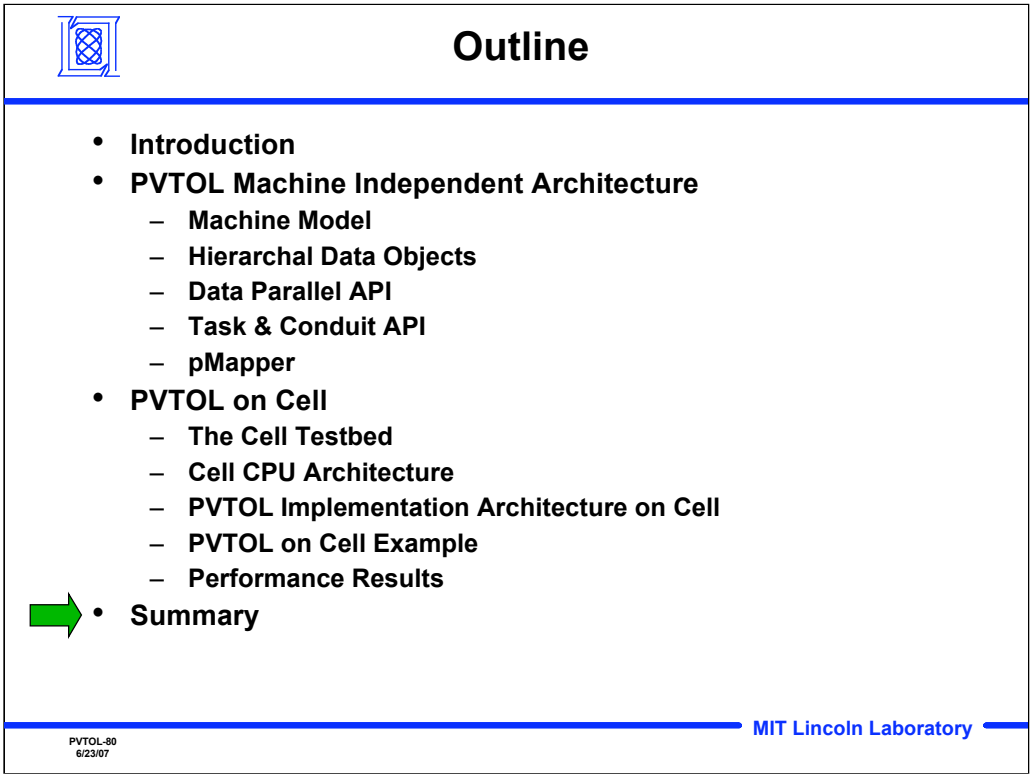
Set 2 has a bank of 20 size 12 filters with size 1024 input vectors

- **TDFIR set 2 scales well with the number of processors**
  - **Runs are less stable than set 1**

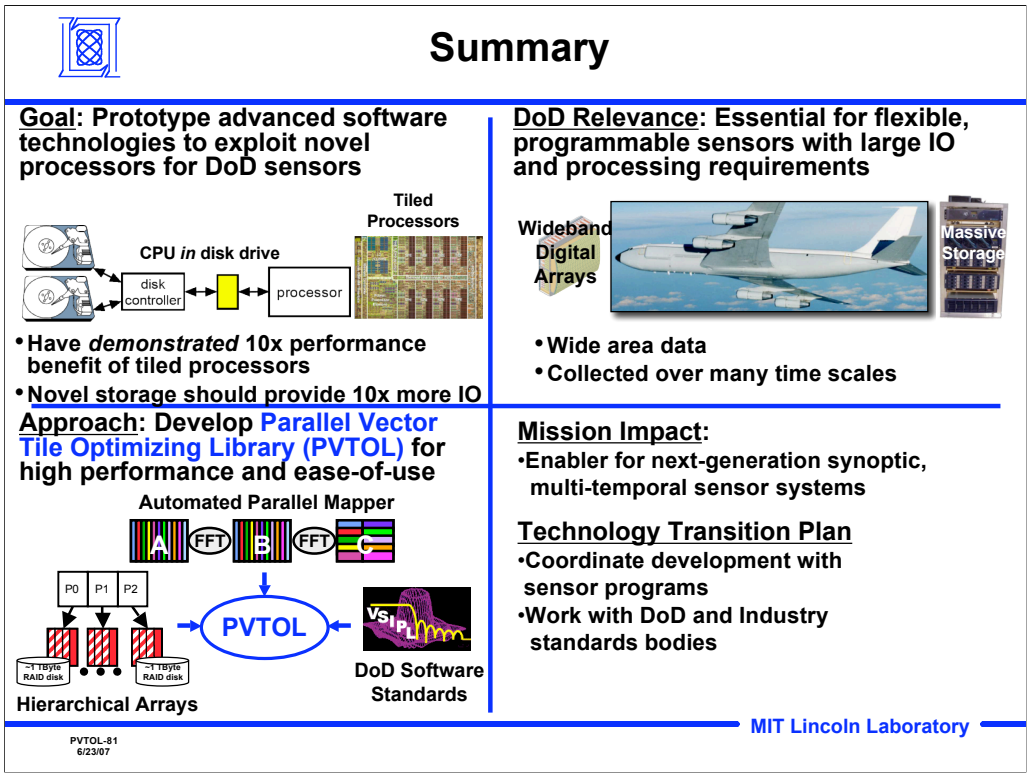
GFLOPS for TDFIR #2 @ 2.4 GHz

# SPE	1	2	4	8	16
GFLOPS	10	21	44	85	185

These are the timing results for set 2 of TDFIR which uses smaller convolutions than set 1. Here the overhead needs more iterations to be averaged out since the convolutions are significantly faster. Again, once the overhead is averaged out, these results scale nicely with the number of processors.


Outline
Introduction
PVTOL Machine Independent Architecture
Machine Model
Hierarchal Data Objects
Data Parallel API
Task & Conduit API
pMapper
PVTOL on Cell
The Cell Testbed
Cell CPU Architecture
PVTOL Implementation Architecture on Cell
PVTOL on Cell Example
Performance Results
Summary
MIT Lincoln Laboratory
PVTOL-80
6/23/07

Outline



The Parallel Vector Tile Optimizing Library (PVTOL) is an effort to develop a new processor architecture for signal processing that exploits the recent shifts in technology to tiled multi-core chips and more tightly integrated mass storage. These technologies are critical for processing the higher bandwidth and longer duration data produced required by synoptic, multi-temporal sensor systems.. The principal challenge in exploiting the new processing architectures for these missions is writing the software in a flexible manner that is portable and delivers high performance. The core technology of this project is the Parallel Vector Tile Optimizing Library (PVTOL), which will use recent advances in automating parallel mapping technology, hierarchical parallel arrays, combined with the Vector, Signal and Image Processing Library (VSIPL) open standard to deliver portable performance on tiled processors.