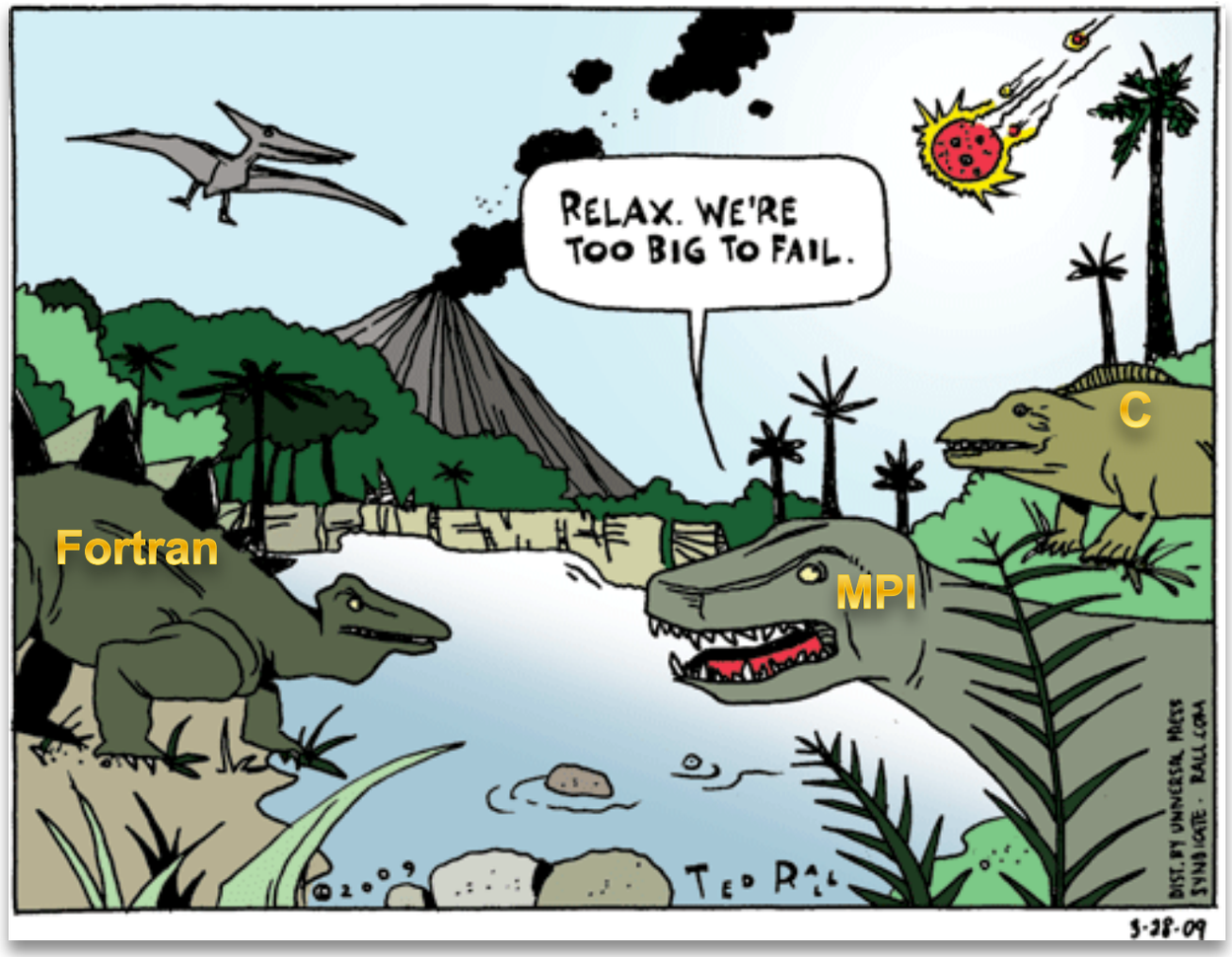
ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

CSCS
Swiss National Supercomputing Centre

Looking Forward to a New Age of Large-Scale Parallel Programming and the Demise of MPI

...hopes and dreams of an HPC educator



Tim Stitt Ph.D., National Supercomputing Services
Swiss National Supercomputing Centre (CSCS), Manno

Acknowledgements

- Chapel (*Brad Chamberlain*, Cray Inc.)
- Co-Array Fortran (*Bill Scherer*, Rice University)
- UPC (*Costin Iancu*, LBNL)
- OpenMP (*Mark Bull*, EPCC)
- X10 (Dave Cunningham, David Grove, IBM)
- Fortress (David Chase, Sun)

In 2012 we will be ushering in a new epoch for HPC systems and software...

In 2012 we will be ushering in a new epoch for HPC systems and software...


Image courtesy of LLNL

Sequoia (LLNL)

- IBM BlueGene/Q
- 20 PetaFLOP/s
- 96 racks,
- 98,304 compute nodes
- 1.6 million cores
- 1.6 PetaBytes of memory

In 2012 we will be ushering in a new epoch for HPC systems and software...

Image courtesy of LLNL

Future systems will exploit many more cores, more threads per core, and farm off relevant tasks to accelerator devices.

Sequoia (LLNL)

- IBM BlueGene/Q
- 20 PetaFLOP/s
- [illegible] nodes
- 1.6 [illegible]
- 1.6 PetaBy[illegible]

In 2012 the ancients also predicted we will be ushering in a new epoch...

In 2012 the ancients also predicted we will be ushering in a new epoch...

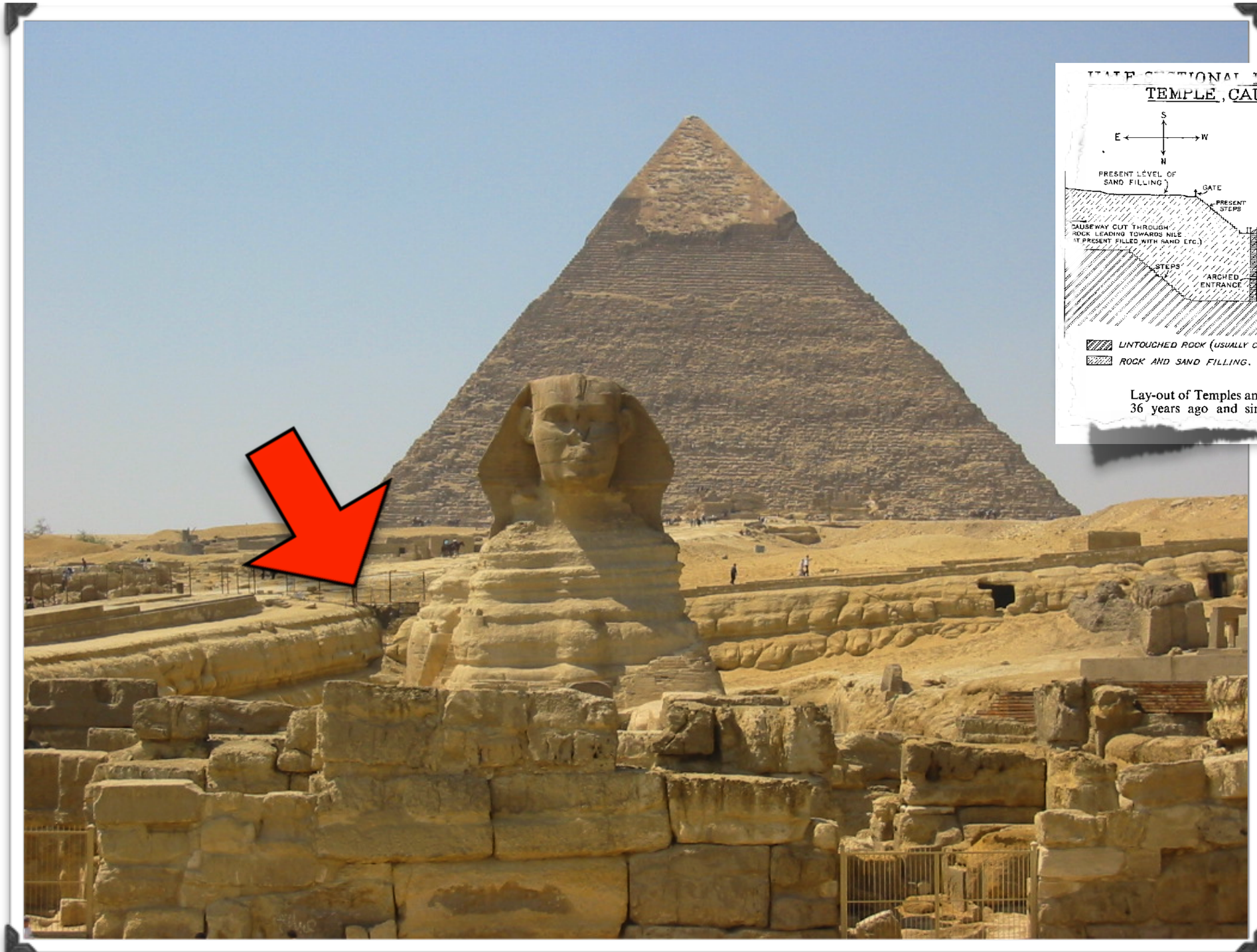
In 2012 the ancients also predicted we will be ushering in a new epoch...

In 2012 the ancients also predicted we will be ushering in a new epoch...

so, just in case they are correct...

What information would the High-Priests of HPC like to record for future generations, in case of a catastrophe?

What information would the High-Priests of HPC like to record for future generations, in case of a catastrophe?



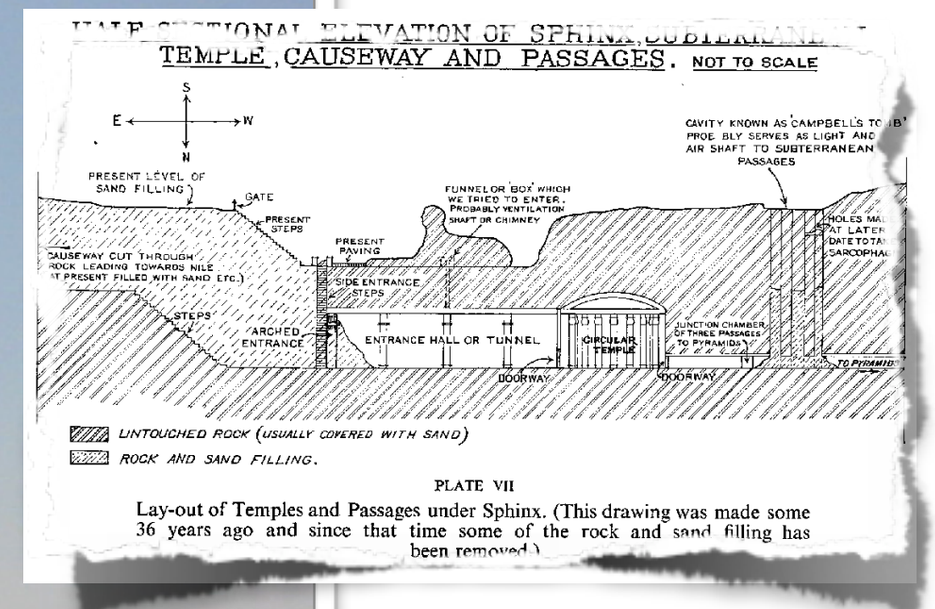
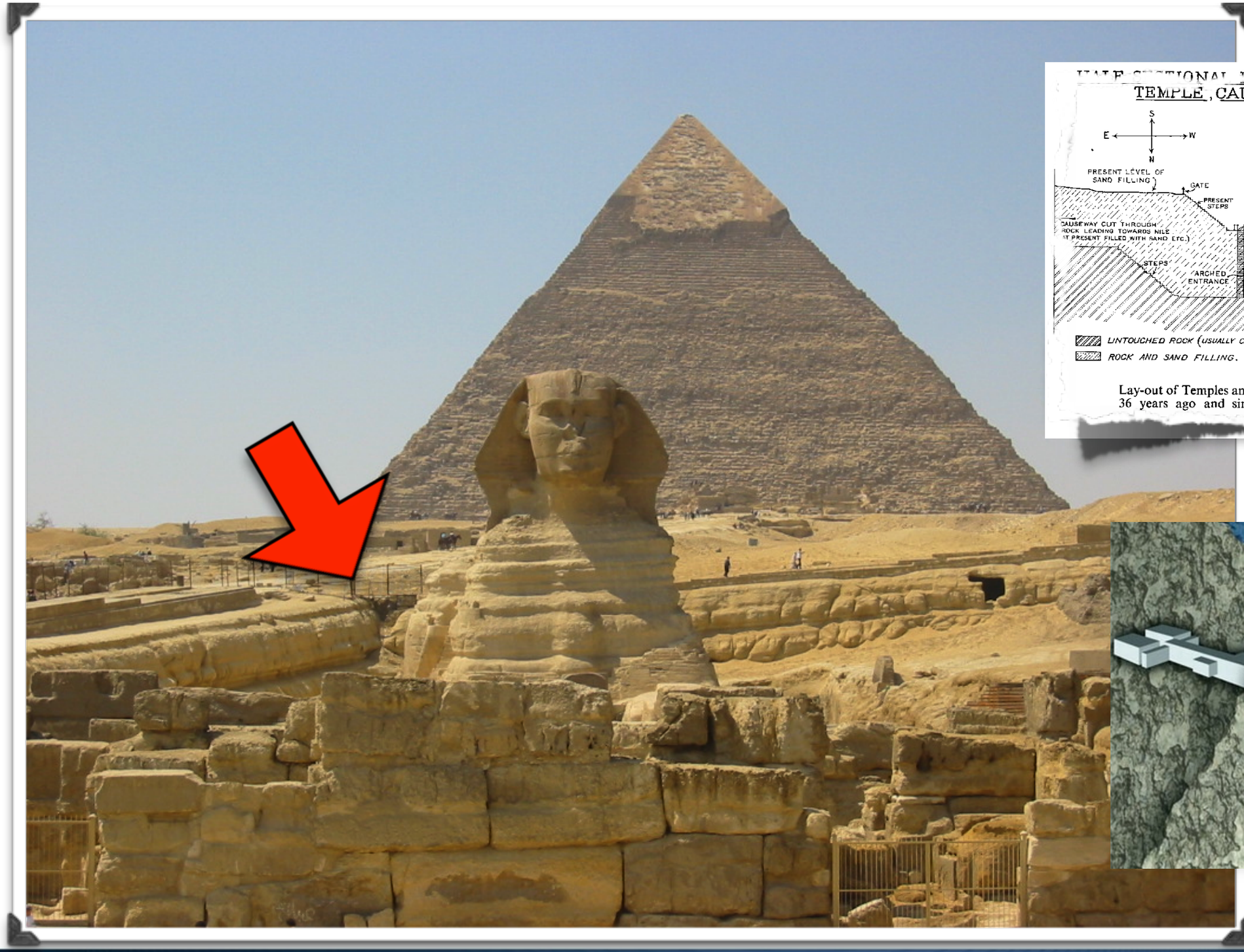


PLATE VII

Lay-out of Temples and Passages under Sphinx. (This drawing was made some 36 years ago and since that time some of the rock and sand filling has been removed.)

What information would the High-Priests of HPC like to record for future generations, in case of a catastrophe?



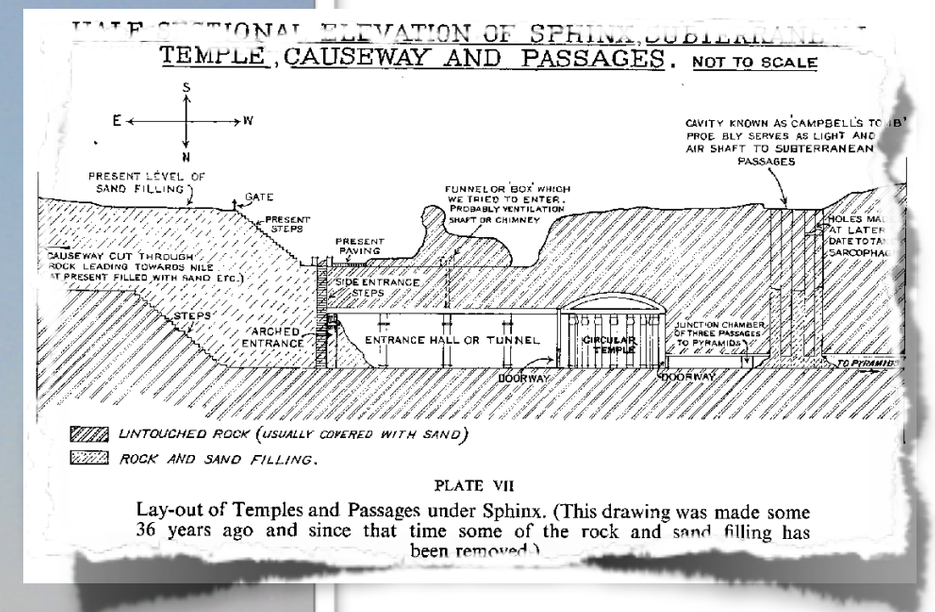
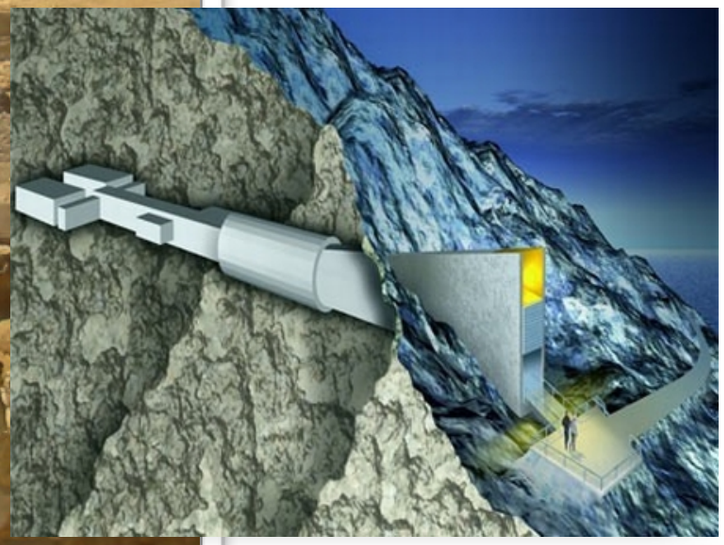


PLATE VII

Lay-out of Temples and Passages under Sphinx. (This drawing was made some 36 years ago and since that time some of the rock and sand filling has been removed.)



What information would the High-Priests of HPC like to record for future generations, in case of a catastrophe?

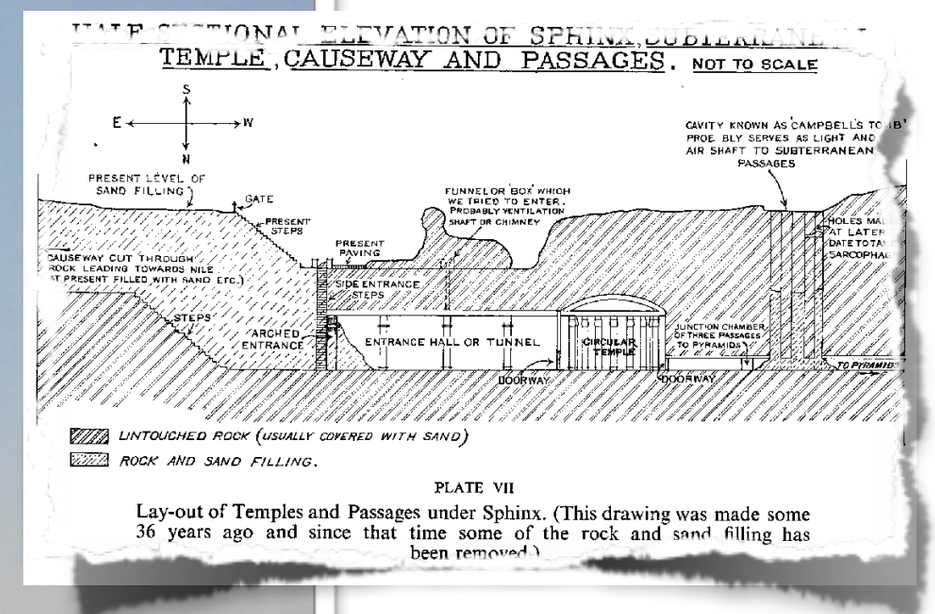
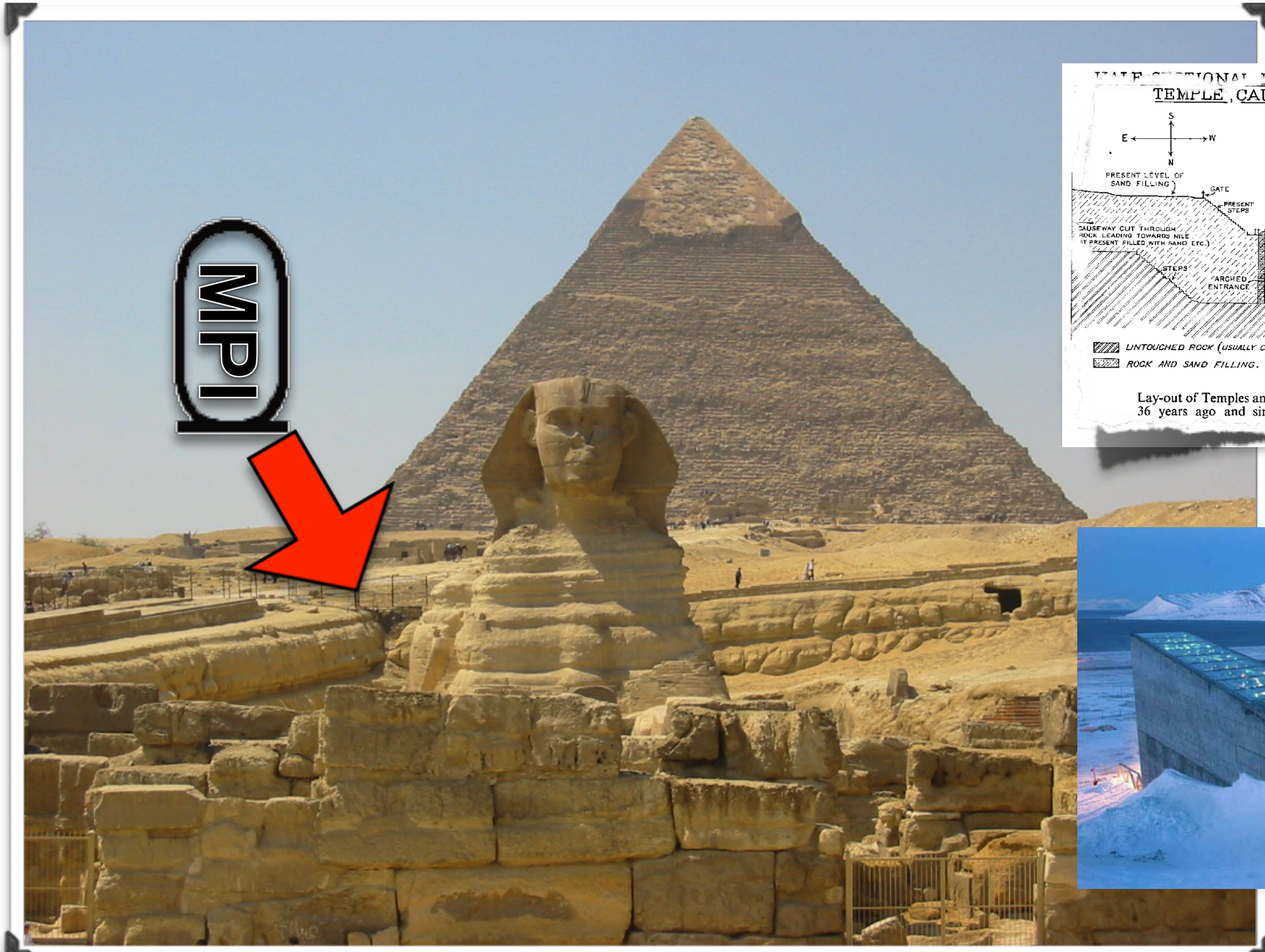


PLATE VII

Lay-out of Temples and Passages under Sphinx. (This drawing was made some 36 years ago and since that time some of the rock and sand filling has been removed.)

What information would the High-Priests of HPC like to record for future generations, in case of a catastrophe?



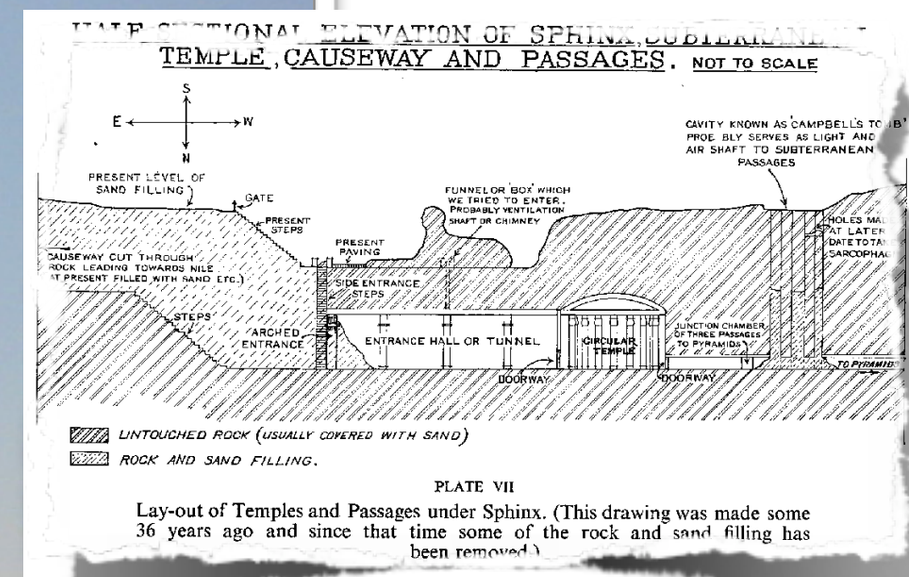
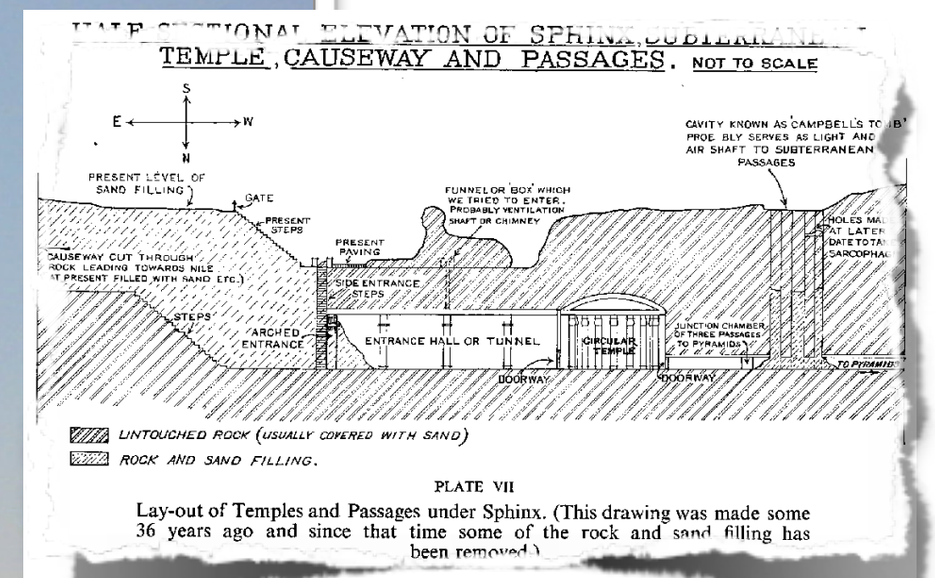


PLATE VII

Lay-out of Temples and Passages under Sphinx. (This drawing was made some 36 years ago and since that time some of the rock and sand filling has been removed.)

What information would the High-Priests of HPC like to record for future generations, in case of a catastrophe?



So What Would Life Be Like Without MPI?

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ F(n-1) + F(n-2) & n > 1 \end{cases}.$$

So What Would Life Be Like Without MPI?

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1. \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

Serial C

```
long fib_serial(long n)
{
    if (n < 2) return n;
    return fib_serial(n-1) + fib_serial(n-2);
}
```

So What Would Life Be Like Without MPI?

$$F(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1. \\ F(n-1) + F(n-2) & n > 1 \end{cases}$$

Serial C

```
long fib_serial(long n)
{
    if (n < 2) return n;
    return fib_serial(n-1) + fib_serial(n-2);
}
```

OpenMP 3.0

```
long fib_parallel(long n)
{
    long x, y;
    if (n < 2) return n;
    #pragma omp task default(none) shared(x,n)
    {
        x = fib_parallel(n-1);
    }
    y = fib_parallel(n-2);
    #pragma omp taskwait
    return (x+y);
}
```

Cilk++

```
long fib_parallel(long n)
{
    long x, y;
    if (n < 2) return n;
    x = cilk_spawn fib_parallel(n-1);
    y = fib_parallel(n-2);
    cilk_sync;
    return (x+y);
}
```

Chapel

```
def fib_serial(n): n.type
{
  var x,y: n.type;

  if (n < 2) then return n;
  cobegin {
    x=fib_serial(n-1);
    y=fib_serial(n-2);
  }
  return x+y;
}
```

X10

```
public class Fib {
  var n:int;

  public def fib_parallel() {

    if (n<2) return;
    val f1 = new Fib(r-1);
    val f2 = new Fib(r-2);
    finish {
      async f1.fib_parallel();
      f2.fib_parallel();
    }
    r = f1.n + f2.n;
  }
...
}
```

Fortress

```
fib_parallel(n: ZZ): ZZ requires { n >= 0 } =
  if n < 2 then n else fib_parallel(n-1) + fib_parallel(n-2) end
```

instead of...

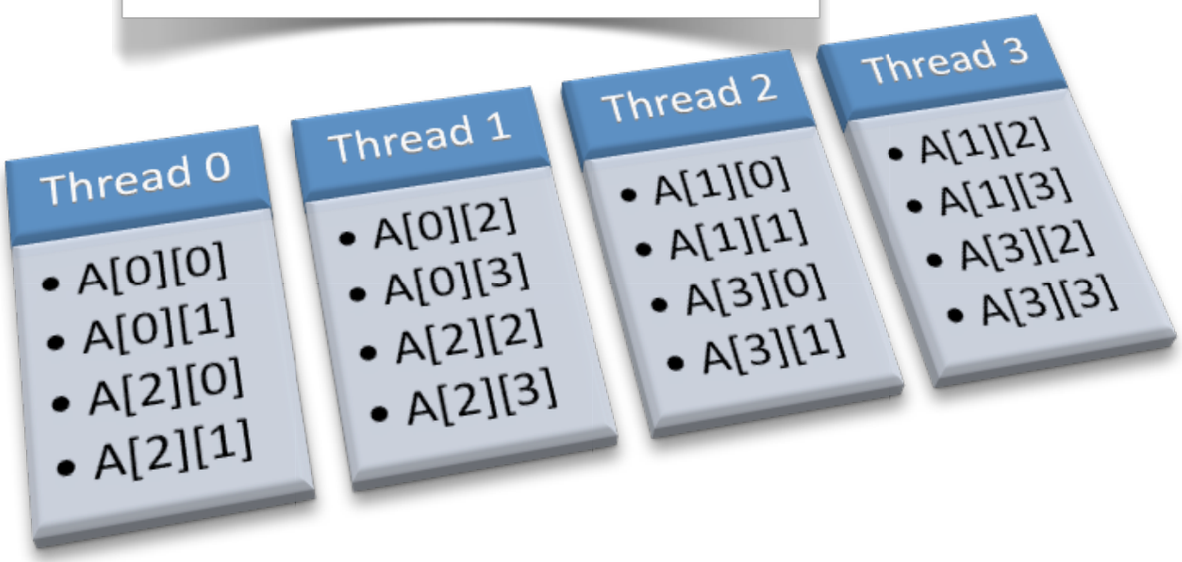
And With MPI...

MPI-2

```
MPI_Comm_get_parent(&parent);
if (n < 2) {
MPI_Isend (&n, 1, MPI_LONG, 0, 1, parent, &req);
}
else {
MPI_Comm_spawn ("Fibo", argv, 1, local_info, myrank, MPI_COMM_SELF, &children_comm[0],
errcodes);
MPI_Comm_spawn ("Fibo", argv, 1, local_info, myrank, MPI_COMM_SELF, &children_comm[1],
errcodes);
MPI_Recv (&x, 1, MPI_LONG, MPI_ANY_SOURCE, 1, children_comm[0], MPI_STATUS_IGNORE);
MPI_Recv (&y, 1, MPI_LONG, MPI_ANY_SOURCE, 1, children_comm[1], MPI_STATUS_IGNORE);
fibn = x + y;
MPI_Isend (&fibn, 1, MPI_LONG, 0, 1, parent, &req);
}
MPI_Finalize ();
```

Data Distribution Can Also Be More Natural

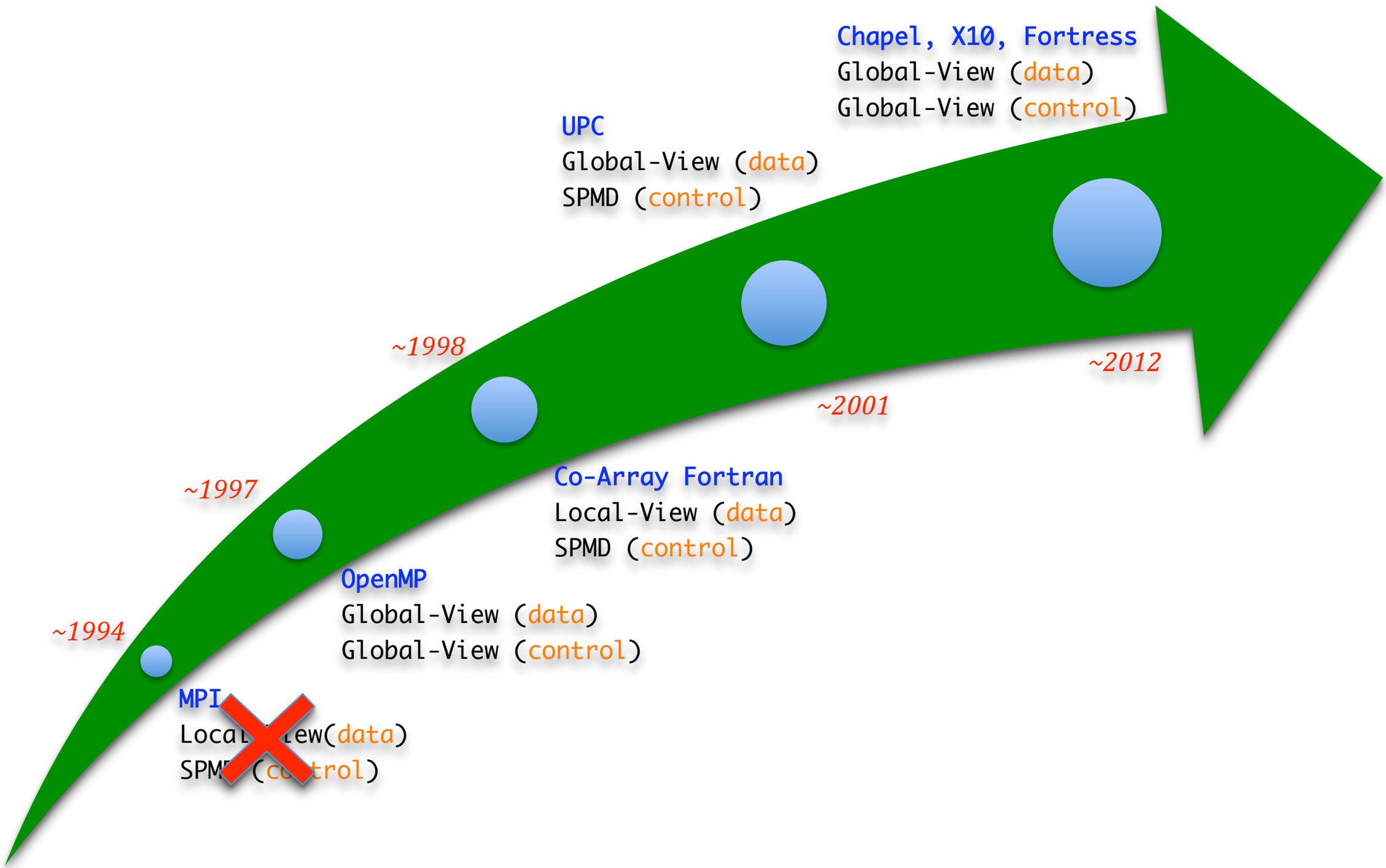
```
UPC (4 threads)
shared [2] int A[4][THREADS];
```



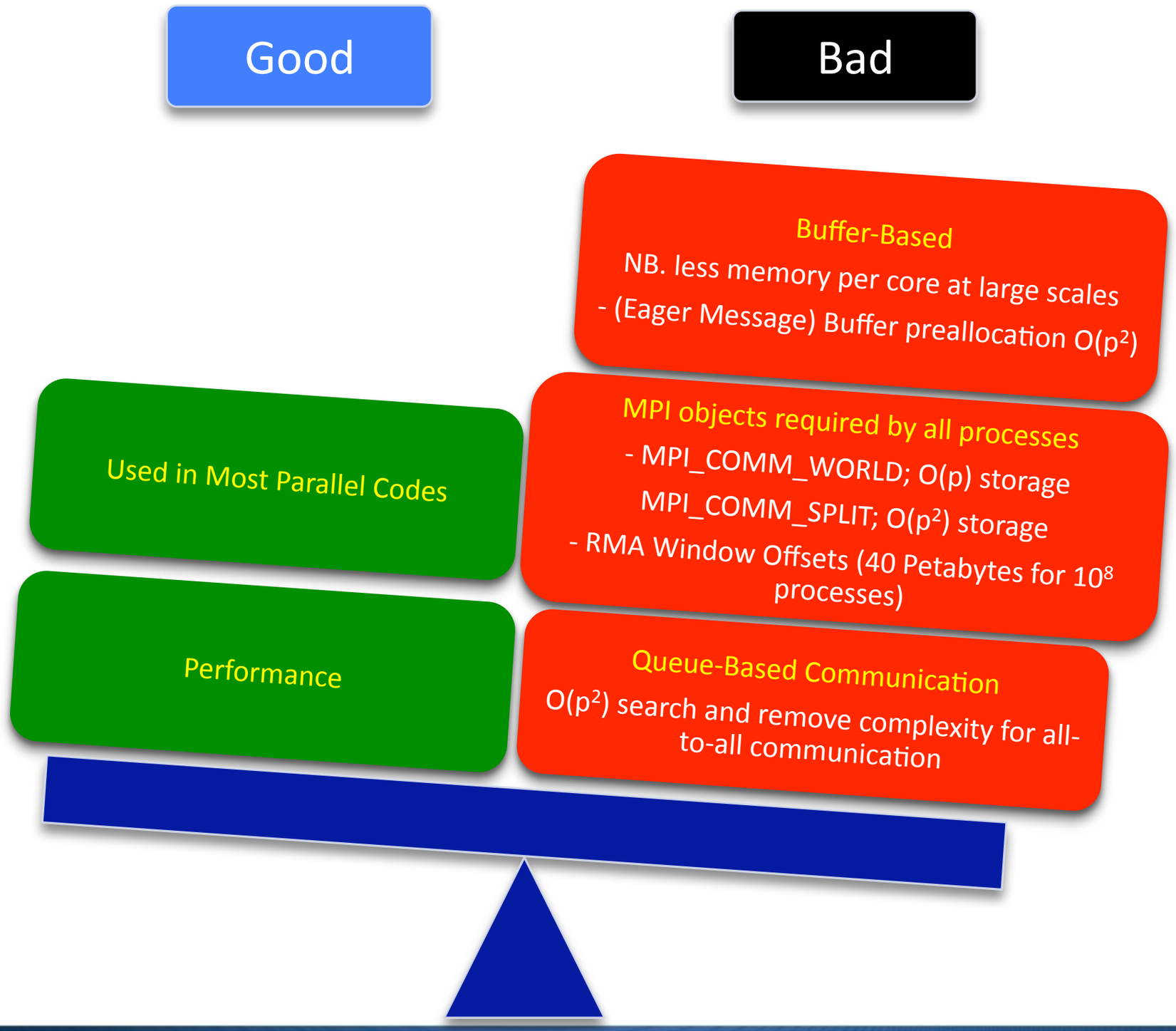

```
Co-Array Fortran (4 images)
REAL :: A(2,2)[*]
```


```
Chapel (4 locales)
const Dom: domain(2) distributed Block(rank=2, bbox=[1..4, 1..4]) = [1..4,1..4];
var A: [Dom] int;
```

Post-MPI...Who Are The Main Contenders?



The Good, the Bad and the MPI



Some (Necessary) Attributes of Large-Scale Parallel Programming Solutions

- Parallelism and Concurrency
  - Allow programmers to *expose large amounts of parallelism/concurrency*. The language solutions should yield this information as *naturally* and as elegantly as possible
  - Inherently provide *adaptive load-balancing*, *application-level fault tolerance* and *high computation/ communication overlap*.
- Reduced Data Movement
  - Moving data is *expensive* in terms of power. Complete *control of data locality* is paramount
  - Language Support for Parallel I/O (and In-Situ Visualization etc.).
- Heterogeneity
  - It is likely that "nodes" will *comprise different computing devices/models*. Programming solutions need to be able to accommodate these multiple models of execution (whatever they turn out to be).
- Interoperability
  - Solutions should be reconcilable with older libraries/languages as well as other well-supported competitors.
- OS Agnostic
  - Solutions should be "easily" *portable* to any future OS that may be required for Exascale computing.
- Be a complete solution
  - A solution that requires the admix of two or more separate programming languages/models should be avoided (in my opinion). Solution should also provide *robust debugging and profiling support*.

OpenMP

OpenMP (Open Multi-Processing) is an application programming interface (API) that supports multi-platform shared memory multiprocessing programming in C, C++ and Fortran on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior.

A Complete Solution?

- *No* – was never designed to be run on distributed memory systems and no intention to drive it in that direction
- For large systems, OpenMP will have a role within a node, most likely in combination with message passing e.g. *MPI* or some lighter weight one-sided interface
- A wide range of debuggers and profilers are available e.g. *Totalview*, *DDT*, *ompP*, *TAU*, *Vampir* etc.

Parallelism and Concurrency

- *Parallel regions* and work-sharing via *omp for* and *sections*. *omp task and taskwait* for task creation and *atomic* statements
- *asynchronous model* being considered for future e.g. maybe similar to StarSs model from Barcelona

OpenMP...continued

Heterogeneity Support

- Ongoing effort to see if OpenMP can be extended to support GPGPUs

OS Portability

- If platform supports *pthreads* and *posix API*, there shouldn't be significant effort.

Interoperability

- OpenMP can be intermixed with MPI. No work on further interoperability e.g. with UPC or CAF, and whether this actually makes sense

Co-Array Fortran (v2.x - Rice University)

Co-Array Fortran (CAF) is a SPMD parallel programming model based on a small set of language extensions to Fortran 90. CAF supports access to non-local data using a natural extension to Fortran 90 syntax, lightweight and flexible synchronization primitives, pointers and dynamic allocation of shared data, and parallel I/O.

A Complete Solution?

- Yes – core focus of CAF
- *Team* implementation is based on a *tree-like distributed data* structure for scalability (no more than $\log^{\text{team size}}$ remote get() operations required to discover image & rank mapping)
- MPI is not required (although there are plans to be compatible with it). *GASNet* is the choice for communication substrate.
- *HPCToolkit* works as a profiler. Debugging is more difficult (possible *gdb* extension if there is interest)

Parallelism and Concurrency

- (Possibly) *lazy spawn* (as in *Cilk*) for dynamic multithreaded parallelism. Coupled with work-stealing would provide dynamic load-balancing
- *Events* allow directed, one-way synchronization from point to point instead of requiring larger-scale barriers.
- *Synchronization variables* and *data-oriented synchronization* will make it possible to synchronize anonymously, i.e. between anonymous threads rather than point-to-point between process images known to each other by index
- *Teams* allow synchronization to be applied only to nodes that have a "need to know". *Locksets* to acquire/release multiple locks in single operation.
- *Non-blocking gets* and *puts* and *non-blocking collectives* (all under development) allow overlapping of communication and computation

Co-Array Fortran (v2.x - Rice University)...continued

Data Locality

- Mapping of data and computation is the programmer's responsibility
- [ ] notation makes it visually explicit to programmers when remote communication is involved in the program
- *Topologies* allow programmers to directly map the underlying physical hardware to logical communication channels
- Will design *synch* algorithms that work with a multi-layer architecture with non-uniform communication costs between pairs of nodes

Interoperability

- CAF to interoperate with MPI
- Glue layer between other languages (to be developed further at upcoming workshops)

OS Portability

- CAF does source-to-source translation to generate standard Fortran 90 inputs. Only a *valid F90 compiler is required*.
- *GASNet* required for communication substrate.

Heterogeneity Support

- No explicit plans to support heterogeneity at this time

UPC

Unified Parallel C (**UPC**) is an extension of the C programming language designed for high performance computing on large-scale parallel machines.The language provides a uniform programming model for both shared and distributed memory hardware. The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses a Single Program Multiple Data (SPMD) model of computation.

Parallelism and Concurrency

- *UPC* is currently a *SPMD* model (using *THREAD* id)
- *upc_forall* can be used to perform workload distribution on shared arrays. *affinity field* helps to ensure only work local to thread is performed.
- Interest in adding *dynamic tasking* to UPC.
- Currently error behaviour not well specified in the language. Thinking about checkpoint-restart for GASNet.

A Complete Solution?

- *UPC* and associated libraries (*collectives* and *I/O*) should be enough to develop applications on large systems
- Working on scalability, as well as extensions to collectives (*multi-field collectives*)
- Already interoperable with *MPI* (restricted *MPI* and *UPC* in separate program phases). ANL is working on *MPI+UPC* (UPC within the node). Ongoing work on interoperability with other languages.
- BUPC supported by *Totalview Debugger*. UPC support added to *DWARF* spec. Limited *gdb* support.
- Performance analysis can be performed with *TAU* (Oregon) and the *Parallel Performance Wizard* (Univ. Florida)

UPC...continued

Data Locality

- Notion of *memory affinity* to capture locality.
- Co-locating and distributing tasks is currently orthogonal to the language and delegated to the user.
- Currently developing *team operations* (MPI communicators)
- Almost complete package of *autotuning collectives*.
- Library Specification for *UPC I/O*.

Interoperability

- Operating with other languages which use *GASNet* should be relatively simple (e.g. CAF, Chapel)

Heterogeneity Support

- Prototype design for UPC on clusters of GPUs
- *Active Messages* for synchronization and one-sided communication that maps well to *DMAs*

OS Portability

- Effort no larger than for any other runtime.
- *GASNet* required for communication substrate.

Fortress

Fortress is a new programming language designed for high-performance computing (HPC) with high programmability. Fortress features include:

- Implicit parallelism
- Transactions
- Flexible, space-aware, mathematical syntax
- Static type-checking (but with type inference)
- Definition of large parts of the language in its own libraries

A Complete Solution?

- Fortress is *not* currently focused on high-end distributed computing. No novel petascale system is being targeted and focus now shifted to "*volume*" computing.
- Current compiler effort targeted to *single JVM* instance and exploitation of available *multithreading*
- Fortress is designed to be *growable*, and further functionality is anticipated with *add-ons* written as Fortress code
- Design goal is to free programmer from burden of explicitly mapping parts of program onto specific hardware resources. Expectation that compiler will always do a better job for all but most critical cases
- No specific support tools. Can reuse tools developed for *Java*.

Parallelism and Concurrency

- Parallelism almost by *default* e.g. *loops* over data structures, *parameter evaluation* etc.
- *Data parallelism* and *control parallelism* can be freely nested. *Atomic blocks* implemented *through optimistic transactional memory* are available for concurrent updates and can be nested
- Programmer can define *associative binary combining operations* over complicated data structures e.g. *trees*, *lists*, *sets*, *multisets* and *maps*, which can be evaluated in parallel
- Operations like *incremental checkpointing* may be possible due to its similarities with garbage-collection and transaction implementations.
- *Adaptive load-balancing* has been central to Fortress implementation from the beginning through *adaptive work-stealing*

Fortress...continued

Interoperability

- Difficult problems to solve due to differences in threading models e.g. *works-stealing*, differences in memory-model e.g. *single address space*, *transactional* and differences in data structures e.g. *arrays*
- Most time spent working on *interoperability with Java*. Not much time on UPC/CAF/Chapel and X10

Heterogeneity Support

- No immediate plans
- The *hardware-description data structure* may provide a framework for exploring this, in an extensible user-programmable manner.

Data Locality

- The design of Fortress includes a *structure* that serves as an abstract hierarchical description of processor and memory resources
- Data structures and threads can be explicitly mapped onto hardware resources by writing code to perform the mapping i.e. *custom data distribution strategies*
- Everything in Fortress is performed by a *library*. Parallel I/O would be perfect fit. Currently implemented through call-outs to Java code through a foreign function interface

OS Portability

- Fortress team believe very strongly in *virtual machines with dynamic runtime compilation* e.g. *HotSpot*. This provides independence from hardware and OS while allowing code to be optimized in response to actual runtime behaviour
- First Fortress compiler targeted at *JVM*. Plans to improve virtual machine to better support Fortress type system and thread model

X10

X10 is a type-safe, modern, parallel, distributed object-oriented languag. A member of the Partitioned Global Address Space (PGAS) family of languages, X10 highlights the explicit reification of locality in the form of places; lightweight activities embodied in async, future, foreach, and ateach constructs; constructs for termination detection (finish) and phased computation (clocks); the use of lock-free synchronization (atomic blocks); and the manipulation of global arrays and data structures.

Parallelism and Concurrency

- First-class language constructs such as *async*, *finish, futures, force, foreach* and *clocks*.
- Planned that *X10 libraries* e.g. arrays, will use async/finish "under the covers" to implement bulk operations on *data structures* concurrently
- Basic *adaptive load balancing* runtime within a *single place*. Further work required for *inter-place adaptive load-balancing*

A Complete Solution?

- Applications (particularly newer ones) should be able to be entirely written in X10. This is the design goal.
- Lower levels of X10 runtime system sit "on top" of an abstraction of an *active messaging layer*. Implementations for BlueGene, LAPI, IB etc.
- Prototype implementation of abstraction layer for MPI could enable interoperability between X10 programs and existing MPI programs.
- An *Eclipse*-based IDE is being developed with X10 with support for editing, executing, refactoring and debugging X10 code

X10...continued

Data Locality

- Notion of a *place* to describe locality e.g. SMP node.
- *Point* (index) domains are represented as *regions*. *Distributed regions* can be used to declare distributed arrays.
- Extension of *async* to support creation of activities on remote places. *ateach* used to execute iterations of a distribution on different places.
- Looking into parallel I/O libraries but nothing definite yet

Heterogeneity Support

- X10 places provide a natural way of exposing accelerators to the programmer e.g. *CPU and attached GPUs hosted as separate places*
- X10 *async* construct can be used to launch code on a *GPU place* or a *CPU place*
- *async* can spawn threads within the GPU (performed in regular fashion before main kernel code executes). *Clocks* can be used to express concurrency patterns.
- *Rail.copyto* and *Rail.copyFrom* interfaces to move data between CPU and GPU using DMA

Interoperability

- On IBM platforms, X10 sits on top *of Common PGAS runtime*, the core of which is the same for IBM's UPC and CAF implementations
- More work required to see how the potential of interoperation between X10 and UPC/CAF can be realised

OS Portability

- X10 compiler generates fairly portable *C++ code*.
- If platform supports enough of *libc++* and basic *posix* APIs, there shouldn't be significant effort.

Chapel

Chapel is a new parallel programming language that supports a multithreaded execution model via high-level abstractions for data parallelism, task parallelism, concurrency, and nested parallelism. Chapel's *locale* type enables users to specify and reason about the placement of data and tasks on a target architecture in order to tune for locality. Chapel supports *global-view* data aggregates with user-defined implementations, permitting operations on distributed data structures to be expressed in a natural manner. Chapel is designed around a *multiresolution* philosophy, permitting users to initially write very abstract code and then incrementally add more detail until they are as close to the machine as their needs require.

A Complete Solution?

- Yes
- Coupling with MPI could weaken some Chapel Features (*e.g. data distribution*)
- No active work on interfacing Chapel with MPI
- Exploratory stages of *CrayPAT* and *TAU* support. Primitive debugging with *gdb supported*.

Parallelism and Concurrency

- Whole-array operations, *forall* loops, *coforall* loops, recursive and non-recursive task parallelism with *begin* and *cobegin*
- Some ideas for fault-tolerance. Pursuing research collaborations
- Tasking interface expected to map to 3rd party tasking libraries to manage load-balancing. Working with Nanos user-level tasking team at BSC/UPC on task throttling

Chapel...continued

Data Locality

- Notion of *data distributions*, *locales*, *locale IDs, locale methods* and *on()* construct.
- Continue to refine *Locale* concept to include different levels of hierarchy and heterogeneity (*e.g. run task on core #3, run task on GPU*)

Interoperability

- Parallel-parallel language interoperability pursued in collaboration with *Babel Project* at LLNL
- Chapel's user-defined distributions may be rich enough to create distributions that describe CAF and UPC arrays

OS Portability

- Should be fairly straightforward with *pthreads support* and a standard POSIX interface.

Heterogeneity Support

- Yes
- Ongoing Research Project with UIUC

Conclusion

- These are just as exciting times in large-scale parallel programming language design, as for large-scale hardware design.
- New parallel programming languages exhibiting attributes of modern programme language design are more likely to attract new talent into the HPC gene pool
- Parallel programming languages providing elegant and clean mechanisms for task and data parallelism should simplify the burden of educating and training the next generation of HPC users