# Unintelligent Design for Asynchronous Exascale Systems

Tim Mattson (Intel Labs)

… with apologies to those who heard this talk already at the 2011 Salishan HPC workshop.

## Disclaimer

- The views expressed in this talk are those of the speaker and not his employer.
- I am in a research group so anything I say about Intel products is highly suspect.
- There are multiple groups at Intel involved with Exascale computing. While we all talk … I am not a spokesperson for any of these groups.
  - I am in a research lab … its my job to question the status quo and probe the odd corners of the solution space other (more practical) groups wisely avoid.

If you work for any press outlet … please note:
Even if it appears so … I am NOT announcing new products OR new research programs within Intel.

## Preliminaries: Some definitions

- **ExaScale Computer**: An ensemble of nodes with aggregate performance of $10^{18}$ operations per second when running a single exascale application.
- **ExaScale Application**: A loosely coupled parallel application for which a single invocation scales to make effective use of the full exaScale System.
- **Loosely Coupled**: A class of parallel applications with concurrent tasks that contain dependencies that must be resolved at irregular time intervals
- **ExaSkeptic**: A curmudgeon who questions the sanity of trying to build an exaScale computer requiring applications with O(billion) concurrency and a 20 MWatt power budget by 2018.

Cloud

A ~~grid~~ of 1000 petaFLOP computers is not an ExaScale computer.
A parameter sweep problem is not an ExaScale Application.

## Preliminaries: Concerns of an ExaSkeptic

- Most scientists are still trying to figure out what to do with TeraScale … why are we so eager for exaScale?.
- Most of our collective energy should be directed towards mega-Tera/Peta
  - MPI-mostly plus OpenMP/OpenCL/TBB/
  - Frameworks that support common patterns … programmers write apps by plugging most serials patches into these frameworks.
- **If we build an exaScale machine in 2018 running at 20 MWatts … will it be so bizarre that the techniques utilized are unlikely to inform what we do at mega-Tera/Peta?**

But for now … I will suppress my ExaSkeptic mindset and "drink the cool-aid".

*Third party names are the property of their owners.

## Current Programming Practice.

- The fundamental assumptions of programming
  - A computer is a finite state machine with well defined states.
  - The global state of the system at any point in time is known.
  - A program defines a sequence of transitions between well defined global states.

The programmer is an omniscient being with ability to control every facet of the system (at least in principle).

By analogy to the "origins debate", I call this the "Intelligent design hypothesis of programming" (ID).

## The unpleasant reality of exascale computing

- ExaScale Programmers will not be omniscient … in fact, at any given point in time, they will know very little about the system or the computation.
  - You can save “a state of the system” at fixed times in the past, but the current state is unknowable.
  - MTBF* << application runtime .. So the programmer can not be certain of the configuration of the system.
  - Silent errors **will** occur during computations… any operation has a small chance of being incorrect.

Details of the system as well as the computation are opaque … so rather than pretend knowledge where none exists, embrace ignorance.

MTBF: Mean time between failure

## What we actually have with exascale

| Intelligent Design | ExaScale Systems |
|---|---|
| A computer is a finite state machine with well defined states. | Soft errors mean the set of available states defining the computer system are fuzzy. |
| It is possible to know the state of the system at any point in time. | The global state of the system at any given time is uncertain |
| A program defines a sequence of transitions between well defined global states. | A program can only be confident of the state of a local domain |

- Intelligent design breaks down for ExaScale systems.
- We need programming models that don't require ID.

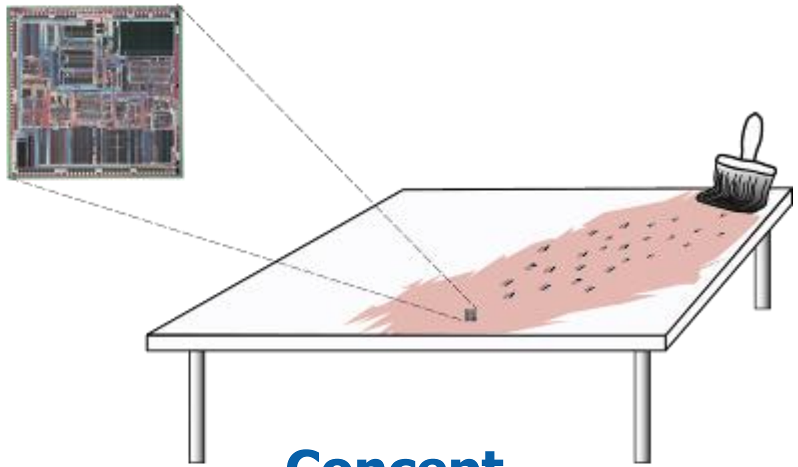
We need Unintelligent design … the only rational design mentality for exascale computers.

# An execution model for UD

- The magnitude of the problem is new, but all the problems we're talking about in UD have been encountered before.
- We can look to the past to understand how to move into the future:
  - **Understanding the system:**
    - Self aware systems, global state emerges from local behavior

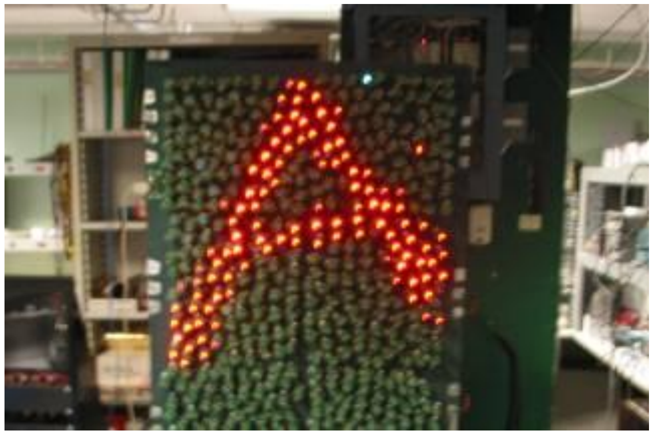
*UD: Unintelligent Design

## Self organizing Systems: Paintable Computers.



**Concept**

- Bill Butera (MIT, 2002)
  - Processing elements (PEs) the size of a large grain of sand.
  - Embedded in a paint-matrix … the computer can be painted out a table.
  - PE’s self organize into a working system … fundamentally resiliant.
  - Proof of concept … a push-pin computer display wall.



**Proof-of-concept**

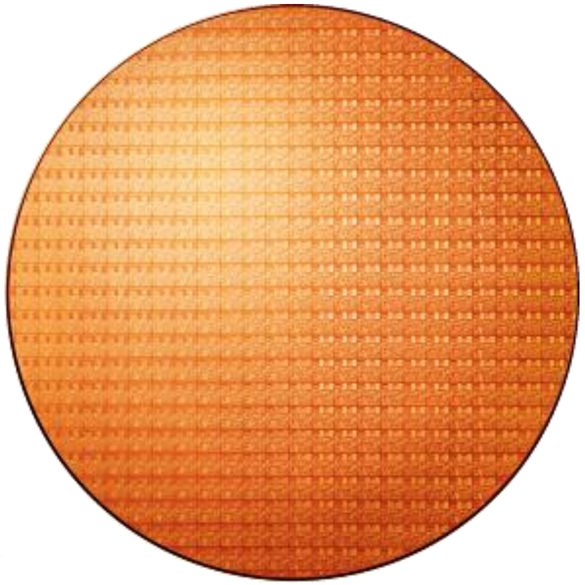
Images from Bill Butera

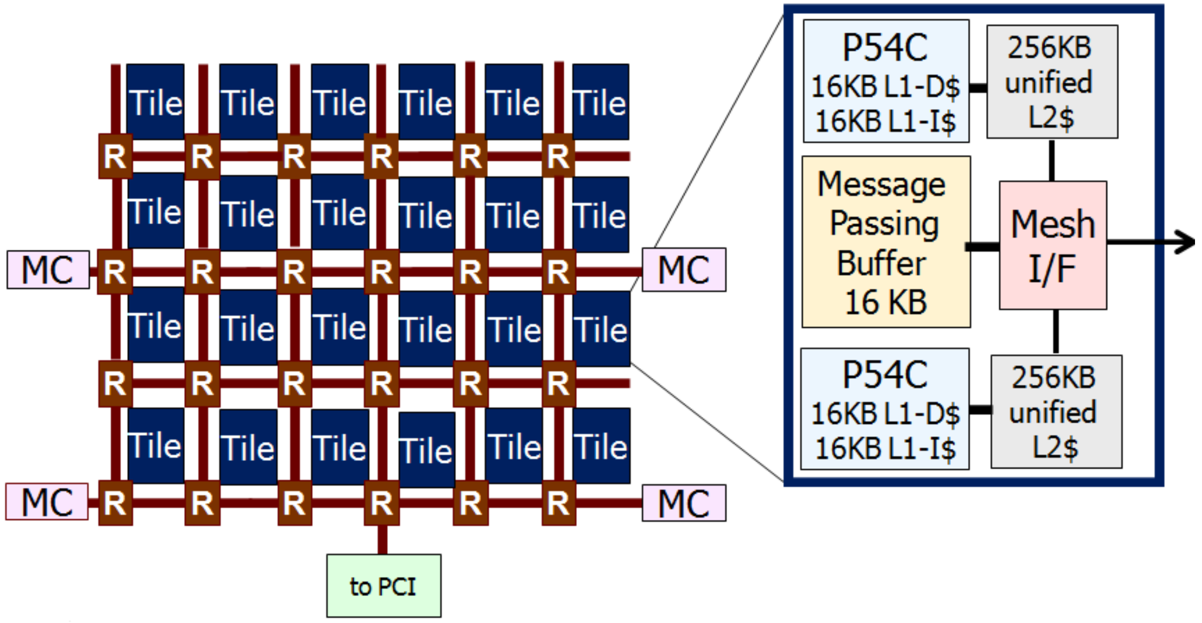
*Third party names are the property of their owners.

## Same methods could be used with a regular array of PE's..

- Place a large array of PE's on a large die … or even a full wafer*.
- Cores self organize into a system .. Reorganize to respond to faults.

* I am NOT announcing a wafer scale integration project at Intel!





- The above is just our existing work on tiled architectures carried to an extreme.

Intel SCC 48 core research processor

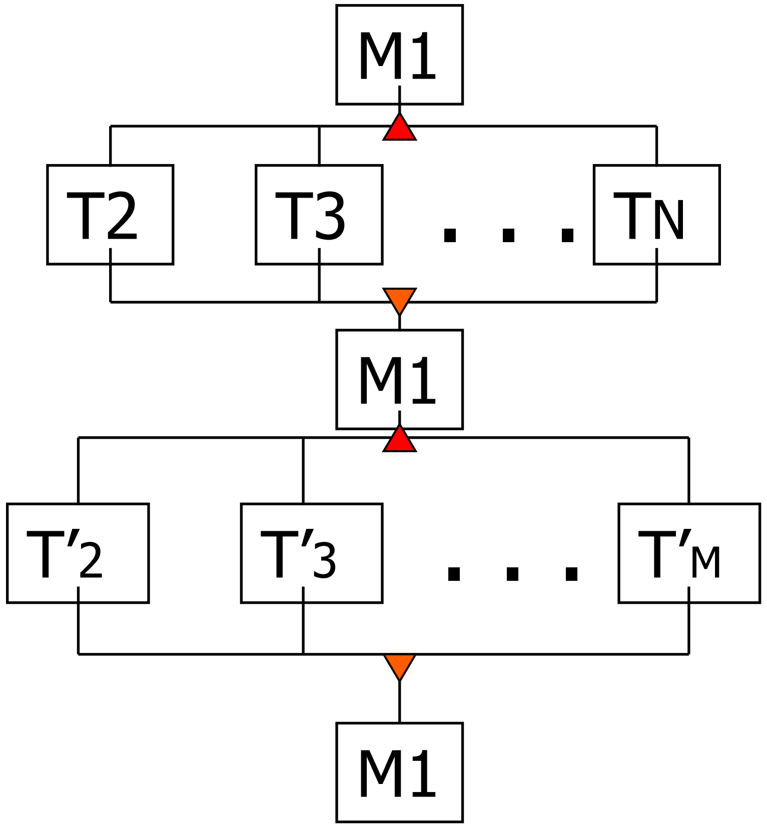
# An execution model for UD

- The magnitude of the problem is new, but all the problems we’re talking about in UD have been encountered before.
- We can look to the past to understand how to move into the future:
  - Understanding the system:
    - Self aware systems, global state emerges from local behavior
      - Scale Free architectures and Paintable computers (Bill Butera, early 00’s).
  - **Dynamic task driven execution model for reliability:**
    - Dynamic tasks driven by distributed task table to manage replication and fault recovery

*UD: Unintelligent Design

# Dynamic Task Drive Execution model

- Calypso: Eager evaluation plus two-phase idempotent execution.
- Key ideas:
  - Problem broken down into phases.
  - Shared data made consistent at beginning and end of each phase.
  - Each phase decomposed into a set of tasks.
  - Tasks tracked in a table … marked unassigned, assigned, or done.
  - Workers grab available tasks … automatically cover failed tasks.
  - Soft error detection easy to add by selectively doubling-up tasks.



▲ Logically copy shared memory to each task.

▼ Logically merge shared memory from each task.

*Third party names are the property of their owners.

Zvi Kedam and Partha Dasgupta: www.calypso.asu.edu (early 90's)

# An execution model for UD

- The magnitude of the problem is new, but all the problems we're talking about in UD have been encountered before.
- We can look to the past to understand how to move into the future:
  - Understanding the system:
    - Self aware systems, global state emerges from local behavior
      - Scale Free architectures and Paintable computers (Bill Butera, early 00's).
  - Execution environment for reliability:
    - Dynamic tasks driven by distributed task table to manage replication and fault recovery
      - Calypso: Distributed parallel computing over LANs (mid1990's).
  - **Programming model:**
    - Ensemble of tasks interacting through high level, distributed data structures:

*UD: Unintelligent Design

## Managing data

- GA and even earlier, Tuple Spaces (Linda), show the expressive power of distributed data structures to coordinate execution of tasks:
  - Referencing data (e.g. indexing into arrays) using a notation convenient to the application.
  - Tasks exploit locality by asking “give me the data closest to me”.
- Can be extended further for unique challenges of exascale:
  - Local monotonic counters as time stamps and background storage into NVRAM … provides a distributed background check -pointing capability.
  - Replicate and distribute partitions of key data (analogous to RAID) to recover data after faults
- Research question:
  - We know this works for arrays, queues, and hash tables.  Can we extend to more general data structures.

## An execution model for UD

- The magnitude of the problem is new, but all the problems we're talking about in UD have been encountered before.
- We can look to the past to understand how to move into the future:
  - Understanding the system:
    - Self aware systems, global state emerges from local behavior
      - Scale Free architectures and Paintable computers (Bill Butera, early 00's).
  - Execution environment for reliability:
    - Dynamic tasks driven by distributed task table to manage replication and fault recovery
      - Calypso: Distributed parallel computing over LANs (mid1990's).
  - Programming model:
    - Ensemble of tasks interacting through high level, distributed data structures:
      - GA (early 90's) and Linda (late 80's)

*UD: Unintelligent Design

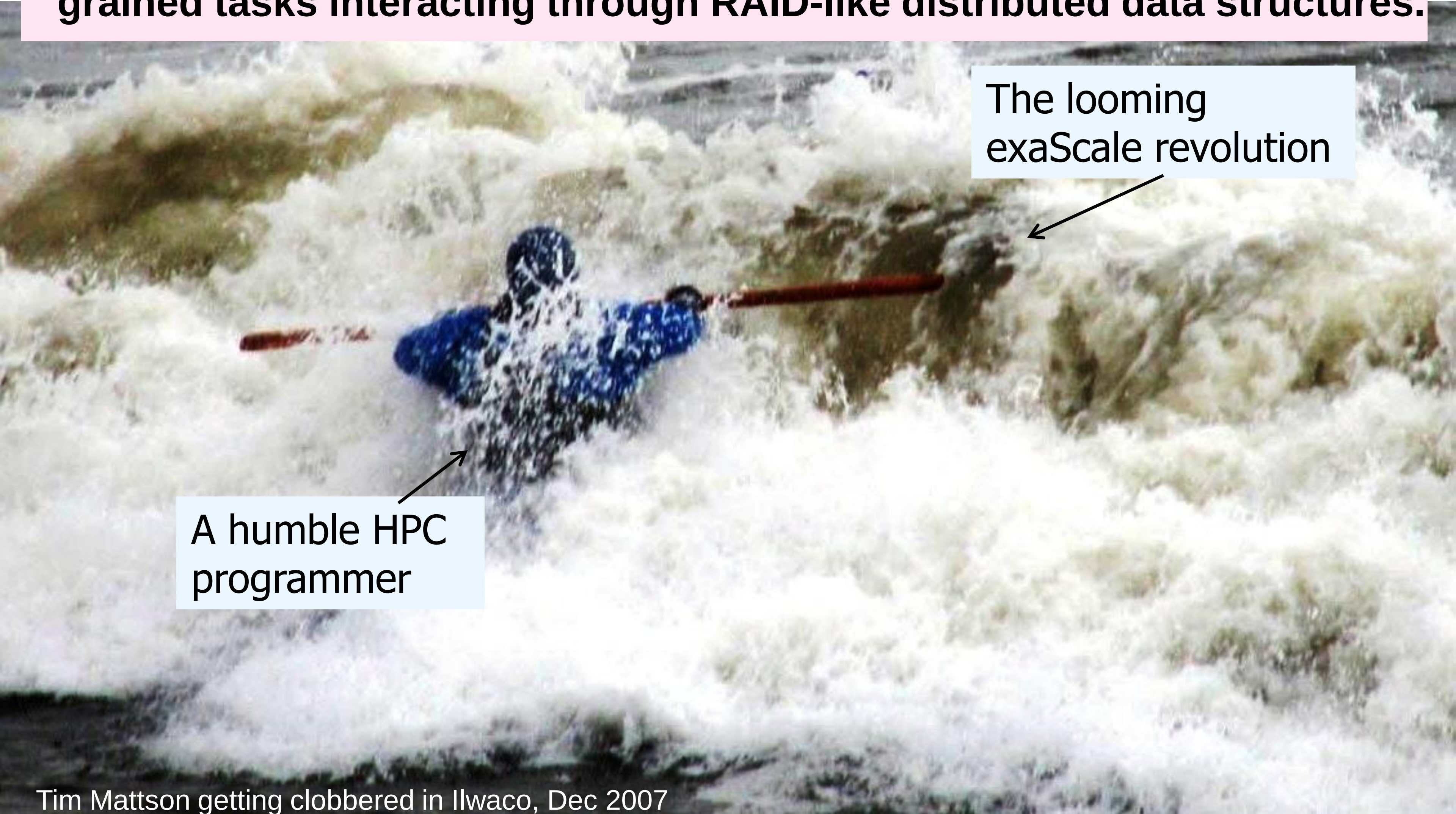
*Third party names are the property of their owners.

## The challenge that scares me: Algorithms

- Exascale algorithms can not depend on checkpoint restart.
  - Silent Errors … you'll get them and not even know it.
- Need algorithms that make progress and converge to the right answer even when faults occurs.
  - Many machine learning algorithms map onto a masterless-map-reduce pattern and can tolerate faults.
  - Some classes of linear algebra algorithms can progress around faults if subsets of the computation can be made reliable (by replicating tasks).
  - Stochastic algorithms
- Research question:
  - Can we fine fault resilient algorithms for the problems we care about for exascale systems?

## Conclusion/Summary

- **Intelligent design won't work for a 20 Mwatt system in 2018 .. We must embrace "unintelligent design".**
- **The past can guide us … self-organizing resilient systems with fine grained tasks interacting through RAID-like distributed data structures.**



Tim Mattson getting clobbered in Ilwaco, Dec 2007