Towards Resilient Chapel

Konstantina Panagiotopoulou   Hans-Wolfgang Loidl

[kp167 [1], H.W.Loidl [2]] @hw.ac.uk

Heriot-Watt University

EASC 2015
21st - 23rd April 2015, Edinburgh

Overview

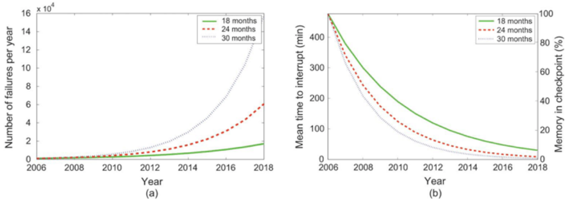
Resilience

Resilience : the ability of a system to **maintain state awareness** and an **accepted level of operational normalcy** in response to disturbances

# of components in modern High Performance Computing (HPC) systems (Tianhe-2 - **3 million cores**, Sequoia **1.5 million cores**)
⇒ challenge on resilience

The Need for Resilience

today's HPC systems ⇒ without failure handling strategies ⇒
Mean Time Between Failure is deteriorating



⇒ significant **waist of their capacity** on failure

- molecular dynamics algorithms
- safety critical systems
- simulation algorithms that require precise results

⇒ **multiple failures during execution**

* Schroeder, Bianca, and Garth A. Gibson. "Understanding failures in petascale computers." Journal of Physics:

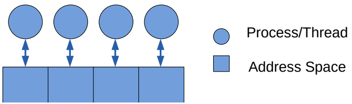
Objectives

- address hardware failure (on one or multiple nodes) during execution of a Chapel program in a distributed setup
- ensure **program termination** and **execution of all tasks**
- **complete transparency** and **automatic task adoption** $\Rightarrow$ no compiler changes $\Rightarrow$ no extra programming effort

The PGAS programming model

Partitioned Global Address Space (PGAS) languages
distributed memory hardware ⇒
programming with PGAS ⇒
globally shared memory

virtual global address space (one-sided message passing library e.g GASNet)

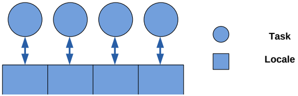


Asynchrony :

- each node executes multiple tasks from a task pool
- nodes can invoke work on other nodes

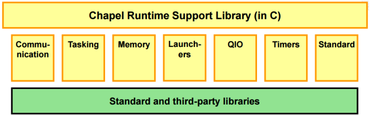
Chapel's Locales and Tasks

Locale: an abstract unit of the target architecture with storage and execution capabilities (e.g. a multi-core processor)
Multi-locale programs start on Locale 0 and scale out



Task: Wrapper of a computation that may execute in parallel

Chapel's Runtime System



- **GASNet** - instantiation of the Communication layer
- **endCounts** - internal module for tracking parallel task completion

Resilient Design

Node failure: anything that prevents nodes in the system from communicating

In Chapel we assume:
a distributed setup where all locales may fail
computation starts by default on Locale 0 and scales out
node failure = locale failure *flat locale model

⇒ target: task migration constructs

Language Constructs : on

on ⇒ task migration

- explicit to the Chapel programmer
- control over locality of the task
- logical continuation of the initial task on a different locale
- **blocking operation on the parent's side**
- explicit synchgronisation point

Example:

```
writeln("start on locale 0");
on Locales[1] do
writeln("now on locale 1");
writeln("on locale 0 again");
```

Chapel's Design Principles

Locality control :**on** construct ⇒ task migration
Task parallelism :**begin, cobegin, coforall** constructs ⇒ task creation
(unstructured, block-structured, loop-structured)

Parallelism and locality are orthogonal ⇒
all constructs can be combined arbitrarily
Fork operations are distinguished in *blocking* and *non-blocking* based on the combinations of the above

Data Redundancy

- where? - **resilient storage**
- in what form? - **data structures**
- what kind of information?
  - copies of the evaluation context (body of migrated tasks)
  - status information on Locales

Assumptions

- **locale 0 is failure free** and acts as resilient storage
- resilience is only supported during program execution - errors during initialisation are fatal
- a **failing locale notifies for its failure** – can be replaced in the future by a hardware notification mechanism
- tasks will execute till completion or not at all

*we aim to weaken these assumptions later on

Specifications

- **Version** : 1.8.0
- **Platform** :64-bit Linux
- **Compiler** :GNU compiler suite
- **LocaleModel** : flat
- **Communication conduit** : gasnet
- **Tasks** : fifo (over POSIX threads)
- **Memory** : default(standard C malloc/free commands)
- **Atomics** : intrinsics
- **Launcher** :amudprun

Data Structures

`failed_table` [array of length # Locales]
stored on Locale 0
records failed nodes(tuple of node id's and status variables)
updates via `FAIL` signals

`transit_msg_list` & `transit_arg_list` [linked lists]
stored on Locale 0
records in-transit fork operations
used for recovery in the non-blocking case
updates via `IN_TRANSIT` and `IN_TRANSIT_DEL` signals

Communication Functions- Forking Operations

Base Idea:
Extend the current GASNet implementation to support resilience

`on` is implemented with a remote fork

- switches to a different locale to execute the task
- can express nested parallelism
- leverages shared data to reduce memory copying

Fork Operations Examples

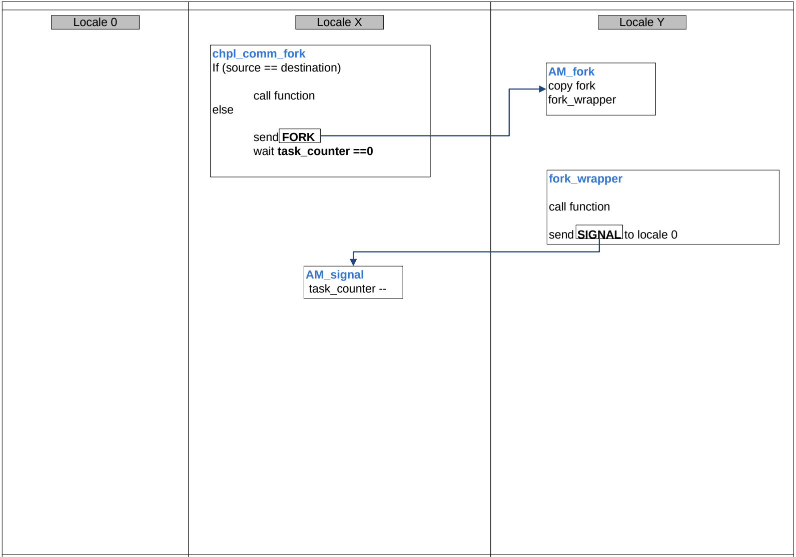
```
//on Locale 0
on Locales[1] do
computation();
//back on Locale 0
```

Listing 1: *Distributed serial Chapel program*

```
//on Locale 0
begin on Locales[1] do
computation();
on Locales[2] do begin
computation();
//back to Locale 0
```

Listing 2: *Distributed parallel Chapel program*

Blocking Fork



Failure Detection

Currently, receiving active notification from the node

- `TIMEOUT` signal - **src**: failing locale **dst**: parent locale

The parent handles the **recovery of the remote task**

*we avoid setting a timeout period on the parent locale due to the asynchronous nature of UDP messages and the overhead

Locale Status Updates

- `FAIL` - **src**: parent locale **dst**: Locale 0
  The parent has discovered a failure (of a child)
  Locale 0 updates the *failed_table*

- `FAIL_UPDATE_REQUEST` - **src**: parent locale **dst**: Locale 0
  `FAIL_UPDATE_REPLY` - **src**: Locale 0 **dst**: parent locale
  The parent requests an update *before launching a new fork* and block-waits on reply from Locale 0

Failure Recovery

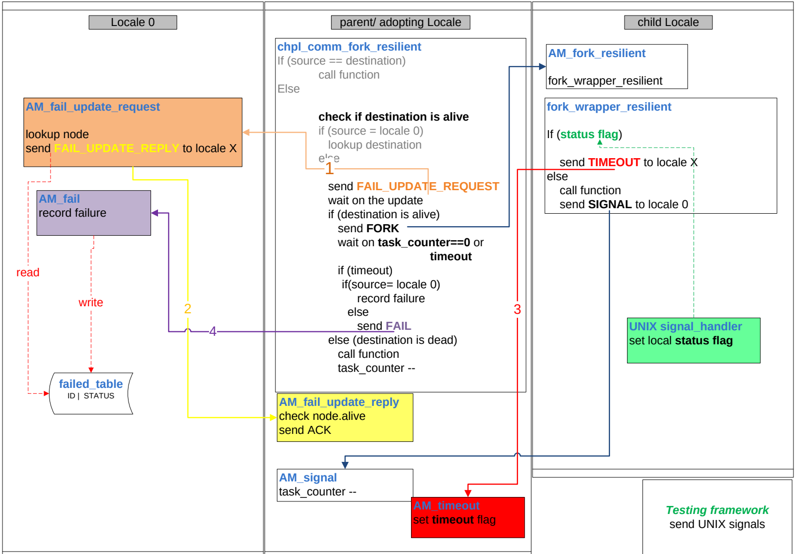
Recovery is handled locally **on the parent locale**, since this is a blocking operation

copies of the evaluation context
are available on the parent locale

Failure of the parent locale is handled on the closest living ancestor (as a destination failure) unless this is Locale 0, which we assume to be failure free and is root of the Locale tree

*such re-launchable functions are free of side-effects that you can't undo; this has to be ensured by an external mechanism and is outside the scope of this work

Resilient Blocking Fork

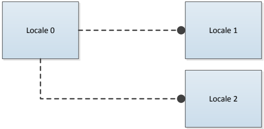


Key implementation aspects

- four additional AM signals and handlers
- one UNIX signal handler to signal failure on locale
- array of failed locales

Constructed Programs - Resilient Blocking Fork - On

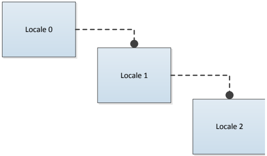
simpleons




```
on Locales[1] do
//computation();
on Locales[2] do
//computation();
```

Constructed Programs - Resilient Blocking Fork - On

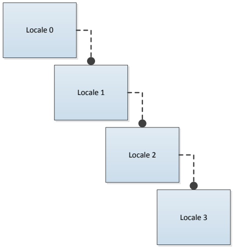
simpleontest




```
on Locales[1] do {
//computation();
on Locales[2] do
//computation();
}
```

Constructed Programs - Resilient Blocking Fork - On

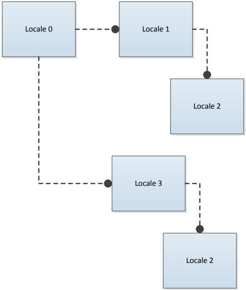
three on




```
on Locales[1] do {
//computation();
on Locales[2] do{
//computation();
on Locales[3] do
//computation();
}
}
```

Constructed Programs - Resilient Blocking Fork - On

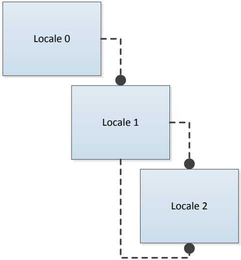
two two




```
on Locales[1] do {
//computation();
on Locales[2] do
//computation();
}
on Locales[3] do{
//computation();
on Locales[2] do
//computation();
}
```

Constructed Programs - Resilient Blocking Fork - On

back




```
on Locales[1] do {
//computation();
on Locales[2] do{
//computation();
on Locales[1] do
//computation();
}
}
```

Testing Framework

- signal-based (overriding default GASNet handlers)
- flexibly simulates node failures for small scale experiments.
- assesses functionality of the prototype implementation
- based on python scripts

2 testing modes
all: all locales (but Locale 0) fail [**stress test**]
rand: random number of locales fail

Limitation:

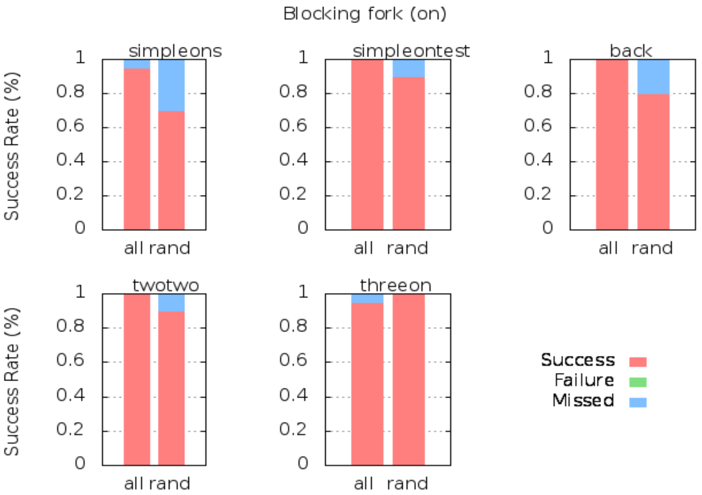
- does not simulate failures at different times during program execution

Testing Platform

Experiments were performed on a 32-node Beowulf cluster (256 cores in total), connected via a Gigabit ethernet network Each machine consists of:

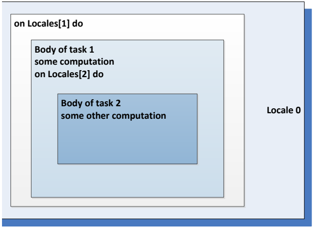
- two quad-core Xeon E5506 2.13GHz
- 12GB of main memory
- three layered cache memory topology (256kB L2 and 4MB shared L3 cache)

Functionality Results- Resilient Blocking Fork - On



Nested Blocking Fork Overhead

The body of each migrated task is a **black box** for the parent locale



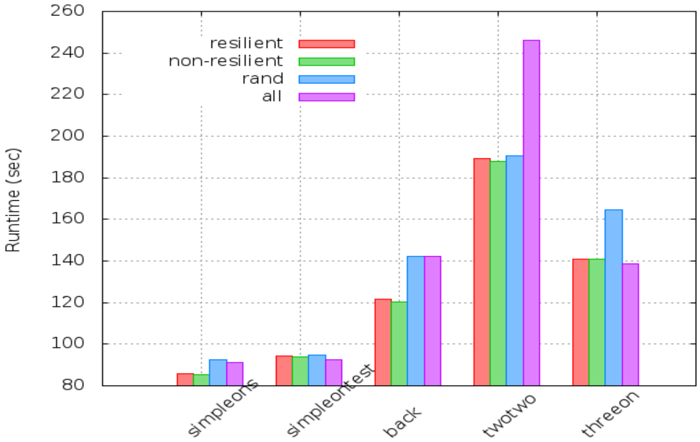
In the figure above, on failure of Locale 1

- locale 0 adopts the task
- reaches the nested on statement & launches the fork

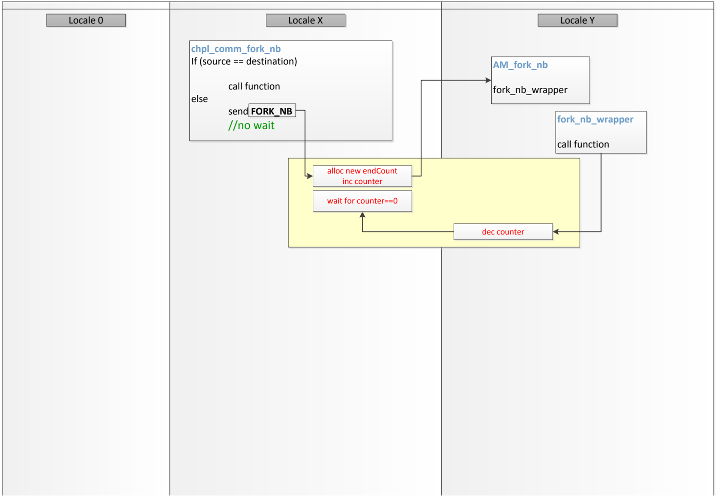
with failures on every two adjacent locales

⇒ recovery on the parent leads to **balanced executions**

Performance Results - On



Ongoing Work - Non-Blocking Fork



Chapel uses **EndCount** objects for each synchronised block to track the completion of parallel tasks

In-Transit Message (non-blocking)

- IN_TRANSIT signal - **src**: parent locale **dst**: Locale 0
- IN_TRANSIT_DEL signal - **src**: parent locale **dst**: Locale 0

Locale 0 keeps track of fork messages in-transit and uses them for recovery The messages and arguments are stored in linked lists

```
struct chpl_comm_transitMsg{
chpl_fn_int_t fid;
int mid;
int src;
int dst;
void* ack;
int  arg_size;
char* data;
chpl_comm_transitMsg_p next;
};
```

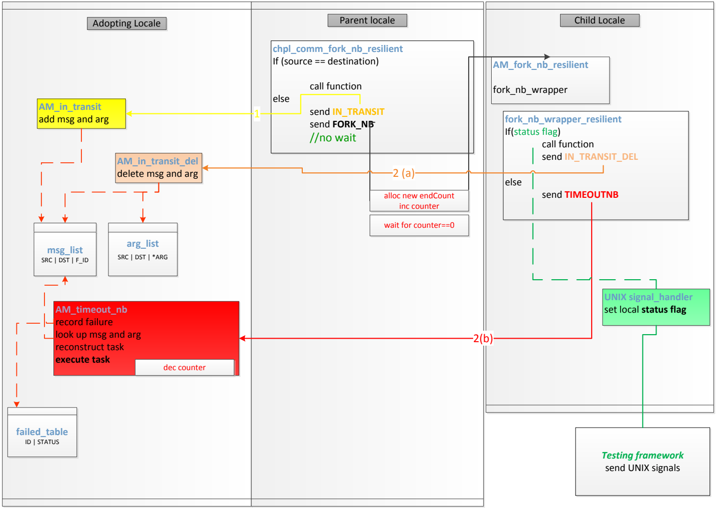
```
struct chpl_comm_transitArg{
int aid;
int arg_size;
char arg[0];
chpl_comm_transitArg_p next;
};
```

Failure Notification(non-blocking)

- `TIMEOUTNB` signal - **src**: failing locale **dst**: Locale 0

Locale 0 handles the **recovery of the remote task**
since the parent locale has exited (**"fire and forget"** nature of non-blocking fork)
*we avoid recovery on the parent as this requires storing information on a locale that might fail

Resilient Non-Blocking Fork



Key implementation aspects

- three additional signals and handlers
- two linked lists for in-transit messages and arguments
- one UNIX signal handler to update the locale's status
- array of failed locales

this is not enough..

The **lost task is recovered** on Locale 0 but ..

Locale 0 cannot decrement the counter of the endCount object

$\Rightarrow$ the execution block-waits on the counter to become zero
$\Rightarrow$ gasnet timeout

Strategies

- fork operation on the failing locale **but**
  - memory is not accessible
  - infinite recovery
    until gasnet timeout
- decrement the counter on the failing locale on detection **but**
  - cannot get a handle to the endCount from the communication layer; statically allocated by the compiler
  - counters may become zero before finishing recovery
- override the endCounts mechanism on the runtime level & extend task adoption policy for endCount's
  $\Rightarrow$ **requires compiler changes**

Limitations

GASNet applies a policy of graceful exit on the event of node crashes prohibits the exclusion of a node from the bootstrapped group
$\Rightarrow$ we can only prevent nodes from communicating with a failed node and recover the task spawned on a failed node

Conclusion

We have presented an **initial design** and **prototype implementation** of
resilience for the PGAS language **Chapel**

Main features:

- **completely transparent implementation**
- **automatic task adoption**
- **data redundancy** and extra inter-locale communication
- **changes only affect the runtime system**, not the compiler
- initial results (with constructed programs) demonstrate low overheads
- minimal set of assumptions

Future Work

Future work focuses on:

- non-blocking fork operations
- distributed task adoption strategies; integration with Chapel's default data distributions
- evolving systems; nodes resurrect or become available at a later point in the execution

References

- Parallel Programmability and the Chapel Language
  Chamberlain, Bradford L., David Callahan, and Hans P. Zima.
  International Journal of High Performance Computing Applications 21.3 (2007): 291-312.
- The Chapel Parallel Programming Language
  -http://chapel.cray.com/
- Resilient control systems: next generation design research
  Rieger, Craig G., David I. Gertman, and Miles A. McQueen.
  Human System Interactions, 2009. HSI'09. 2nd Conference on. IEEE, 2009.
- Evolution of supercomputers
  Xie, Xianghui, et al.
  Frontiers of Computer Science in China 4.4 (2010): 428-436.
- An empirical performance study of chapel programming language
  Dun, Nan, and Kenjiro Taura
  Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2012 IEEE 26th International. IEEE, 2012.

Thank you !

Contigency

... some extra slides ...

The Communication Layer

GASNet – the default instantiation of the communication layer on Linux-based systems; a communication interface for the Global Address Space languages

- network-independent and language-independent
- highly portable
- Active Message(AM) interface on top of UDP
- logically paired request and reply operations
- Core functions return zero on success
- **on fatal error** GASNet terminates the remaining nodes to prevent creation of orphaned processes

EndCounts (ChapelBase module)

Chapel uses **EndCount** objects to track the completion of parallel tasks.
An endCount is allocated for each synchronised block.
The main function itself is governed by an endCount object

```
class _EndCount {
    var i: atomic int,
    taskCnt: taskCntType,
    taskList: _task_list = _nullTaskList;
}
//functions
proc _endCountAlloc();
proc _endCountFree(e: _EndCount);
proc _upEndCount(e: _EndCount);
proc _downEndCount(e: _EndCount);
proc _waitEndCount(e: _EndCount);
```

Language Constructs : cobegin

cobegin ⇒ block-structured task creation

- creates a new task for each statement in the block
- **blocking operation on the parent's side**
- heterogeneous tasks

Example:

```
cobegin {
consumer(1);
consumer(2);
producer();
}
```

Language Constructs : begin

begin ⇒ unstructured parallelism

- launch a new task on a new thread
- **continue with the next statement**
- join: explicit sync block or the implicit sync of the main function
- **fire and forget**

Example:

```
begin writeln("hello world");
writeln("good bye");
```

Output:

hello world good bye	or	good bye hello world

Language Constructs : coforall

coforall ⇒ loop-structured task invocation

- creates a new task for each iteration
- **blocking operation on the parent's side**
- homogenous tasks

Example:

```
begin producer();
coforall i in 1..numConsumers {
consumer(i);
}
```

endCount Allocation

```
def someFunction() {
        begin coFunction(1);
        begin coFunction(2);
}

def coFunction(n) {
    writeln("arg=",n);
}
```

Above: *Chapel code using* `begin` *to spawn a task to exectute a statement.*

The picture was taken from :

"Task Parallel Constructs in Chapel", Thom Haddow, MSc dissertation, The University of Edinburgh, 2008

endCount Allocation

Below: *Abridged mapping into C.*

```
/* begin.chpl:5 */
void someFunction(_EndCount _endCount) {
  // ... (Handling of _endCount from main(), as discussed)
  _class_locals_begin_fn_1 _args_for_begin_fn_1 =
            CHPL_ALLOC_PERMIT_ZERO(sizeof(__class_locals_begin_fn_1),
                        "instance of class _class_locals_begin_fn_1",
                         6, "begin.chpl");
  // ...
  chpl_add_to_task_list( wrap_begin_fn_1, _args_for_begin_fn_1,
                  &((_args_for_begin_fn_1)->_1__endCount->taskList),
                  (_localeID), true);

  // ... ( Defininition of args_for_begin_fn_2 and similar call to
  //        chpl_add_to_task_list(). )

  return;
}
/* begin.chpl:6 */
void wrap_begin_fn_1(_class_locals_begin_fn_1 c) {
  // ...
  chpl_thread_init();
  // ...
  _begin_fn_1(c->_1_endCount);
  chpl_free(c, 6, "begin.chpl"); c = NULL;
  return;
}
/* begin.chpl:6 */
void _begin_fn_1(_EndCount _endCount) {
  coFunction(1);
  chpl___downEndCount_11673(_endCount, 6, "begin.chpl");
  return;
}
// ... (Similar for _begin_fn_2)
```