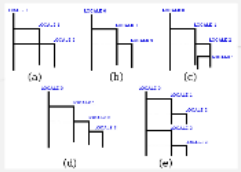
Transparently Resilient Task Parallelism for Chapel

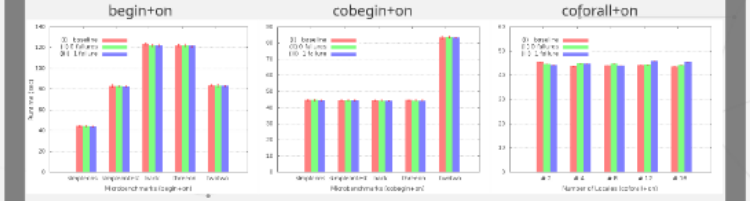
Konstantina Panagiotopoulou, Hans-Wolfgang Loidl

Heriot-Watt University, Edinburgh, UK

Evaluation Results



constructed micro-benchmarks



*System specs:16 nodes of a 32-node Beowulf cluster connected via Gigabit ethernet network, with each node consisting of two quad-core Xeon E5506 2.13GHz, 12GB of main memory and three-layered cache memory topology.

Resilience

Fault tolerance property that enables a system to continue operating properly in the event of failure

Resilience integrated design strategy to enable the fault tolerance property of the system

Transparent Resilience for Chapel

GOAL
- allow distributed long-running Chapel applications to terminate successfully in the presence of hardware failure of participating nodes
- achieve **transparent** resilience without requiring user-assistance or source code modifications

Buddy Locales

Resilient Non-Blocking Fork

Non-Blocking Fork

Failure rates

Chapel's Task Parallel constructs

```
// create a fire-and-forget task
begin async_task();
reg_task();

// create one task per statement
cobegin {
 producer(1);
 producer(2);
 consumer(1);
} // implicit join

// create a task per iteration
coforall t in 0..#numTasks {
  async_task();
} // implicit join
```

on construct

"invoke work on remote locales"

```
on Locales[1] do
  remote_task(); //on Loc1
local_task(); // on Loc0
```

translates to a blocking fork operation

```
begin on Locales[1] do
  async_remote_task(); //on Loc1
local_task(); // on Loc0
cobegin on...
coforall on...
```

translate to non-blocking fork operations

Resilient Store

maintains critical data & survives Failures

Blocking Fork

parent locale waits on completion of the remote task on the child locale

simplest case for recovery

data already available on the parent -> no need for replication

Blocking Fork

recovery from failure is handled on the parent locale
- data retrieved from local memory

locales branch out, forming the LocaleTree with Locale 0 as root

parent failures can be dealt with as child failures one level higher on the tree

- failure of Locale 0 is unrecoverable

Failure Detection

assumption: a node explicitly notifies on its failure

active research on hardware-assisted prediction
- failure prediction/avoidance
  - health metrics
  - external factors (eg. temperature)

GASNet communication API applies strict system-wide abort policy on failures

failure injection/detection needs to be transparent to GASNet

State maintenance

carrying the history of past computations

2 common state-recovery techniques
- checkpointing and forward error recovery

obtain a new correct state and resume execution

use of message logging mechanisms, cost proportional to the number of failures

Limitations

Resilient Blocking Fork[1]

AM_timeout: function performing recovery on the parent using locally available data

AM_fail(_update) : functions to query and update centrally stored (Loc 0) information on the status of nodes

[1] "Towards Resilient Chapel: Design and Implementation of a Non-Blocking Fork Mechanism for Chapel", K. Panagiotopoulou and H.-W. Loidl, 27th International Conference on Parallel Computing and Distributed Systems (PDCS'15), University of Edinburgh, Edinburgh, Scotland, UK, 2015

Ongoing Work

support for Chapel's data parallel constructs

distributions (distributed memory): Block,Cyclic, BlockCyclic etc.

layouts (shared memory)

Focus : **parallel iterators** [*forall loop on blocked data*]

- buddy locales for redundancy and task re-execution
- under-the-hood remote memory operations

https://github.com/KonstantinosP/chapel

Future Work

- TESTING
- Optimizations
  - cost of remote operations
  - minimize added communication
- Performance improvements
  - replace linked list to tackle O(N) cost of serial lookup operations
  - use of async operations on the data structures
- Investigate portability to other predefined distributions

Longer term goals
- cost model of resilience across multiple configurations
- load balancing policy

Thank you!

Questions?

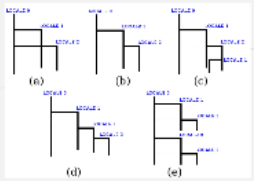
CHIUW'16, IPDPS'16, 23-27 May, Chicago

Transparently Resilient Task Parallelism for Chapel

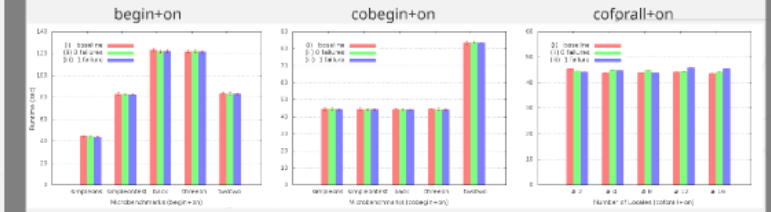
Konstantina Panagiotopoulou, Hans-Wolfgang Loidl

Heriot-Watt University, Edinburgh, UK

Evaluation Results



constructed micro-benchmarks



*System specs:16 nodes of a 32-node Beowulf cluster connected via Gigabit ethernet network, with each node consisting of two quad-core Xeon E5506 2.13GHz, 12GB of main memory and three-layered cache memory topology.

Resilience

Fault tolerance property that enables a system to continue operating properly in the event of failure

Resilience integrated design strategy to enable the fault tolerance property of the system

Transparent Resilience for Chapel

GOAL
- allow distributed long-running Chapel applications to terminate successfully in the presence of hardware failure of participating nodes
- achieve **transparent** resilience without requiring user-assistance or source code modifications

Buddy Locales

Resilient Non-Blocking Fork

Non-Blocking Fork

Failure rates

failure rates grow proportionally to the number of processor chips, despite hardware improvements

Mean Time Between Failures (MTBF)
between 1 day and 1 month
80% hardware-related factors

1.75 to 8.85 annualized disk failure ratios

Chapel's Task Parallel constructs

```
// create a fire-and-forget task
begin async_task();
reg_task();

// create one task per iteration
coforall t in 0..#numTasks {
    async_task();
} // implicit join
```

on construct

"invoke work on remote locales"

```
on Locales(1) do
  remote_task(); //on Loc1
local_task(); // on Loc0
```

translates to a blocking fork operation

```
begin on Locales(1) do
  async_remote_task(); //on Loc1
local_task(); // on Loc0
cobegin on....
coforall on....
```

translate to non-blocking fork operations

Resilient Store

manages critical data & survives failures

Blocking Fork

parent locale waits on completion of the remote task on the child locale

simplest case for recovery
data already available on the parent => no need for replication

Blocking Fork

recovery from failure is handled on the parent locale
- data retrieved from local memory

Locales branch out forming the LocaleTree with Locale 0 as root.

parent failures can be dealt with as child failures one level higher on the tree

•failure of Locale 0 is unrecoverable

Failure Detection

assumption
- a node explicitly notifies on its failure

active research on hardware-assisted prediction
- failure prediction/avoidance
  - health metrics
  - (external) factors (eg. temperature)

GASNet communication API applies strict system-wide abort policy on failures
→ failure injection/detection needs to be transparent to GASNet

State maintenance

carrying the history of past computations

2 common state-recovery techniques

- checkpointing and **forward error recovery**

obtain a new correct state and resume execution
use of **message logging** mechanisms
cost proportional to the number of failures

Limitations

Chapel's decoupled communication and tasking layers → partial ahead tracking of task completion on the parent's side

complexity of distinguishing tasks with side-effects (compiler-assisted)

currently we can perform recovery on side-effect-free programs only

Resilient Blocking Fork[1]

AM_timeout: function performing recovery on the parent using locally available data
AM_fail(_update) : functions to query and update centrally stored (Loc 0) information on the status of nodes

(1) "Towards Resilient Chapel: Design and implementation of a transparent resilience mechanism for Chapel", K. Panagiotopoulou and H.W. Loidl, 2015, In Proceedings of the Institute Autonomic Computing and Software (EASE '15), University of Edinburgh, Edinburgh, Scotland, UK, 16-17

Ongoing Work

support for Chapel's data parallel constructs

distributions (distributed memory)
Block,Cyclic,
BlockCyclic etc.

layouts (shared memory)

Focus : **parallel iterators** [*forall loop on blocked data*]

- buddy locales for redundancy and task re-execution
- under-the-hood remote memory operations

https://github.com/konsP/chapel

Future Work

- TESTING
- Optimizations
  - cost of remote operations
  - minimize added communication
- Performance improvements
  - replace linked list to tackle O(N) cost of serial lookup operations
  - use of async operations on the data structures
- Investigate portability to other predefined distributions

Longer term goals
- cost model of resilience across multiple configurations
- load balancing policy

Thank you!

Questions?

CHIUW'16, IPDPS'16, 23-27 May, Chicago

Resilience

Fault tolerance property that enables a system to continue operating properly in the event of failure

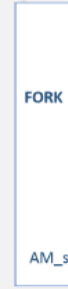
Resilience integrated design strategy to enable the fault tolerance property of the system

Transparent Resilience for Chapel

GOAL

- allow distributed long-running Chapel applications to terminate successfully in the presence of hardware failure of participating nodes
- achieve **transparent** resilience without requiring user-assistance or source code modifications

[1] *"Exascale Computing Study: Technology Challenges in Achieving Exascale Systems,"* P. Kogge, K. Bergman et al., 2008.
[2] *"The International Exascale Software Project: A Call to Cooperative Action by the Global High-Performance Community,"* J. Dongarra, P. Beckman, et al., International Journal of High Performance Computing Applications, 2009.
[3] *"High-End Computing Resilience: Analysis of Issues Facing the HEC Community and Path-Forward for Research and Development,"* NNSA, DOE, DARPA, and DOD, cleared by DARPA on 20-01-2010.



Failure rates

failure rates grow proportionally to the number of processor chips, despite hardware improvements

analysis of failure data by Los Alamos National Laboratories [1]

Mean Time Between Failures (MTBF)
between 1 day and 1 month
60% hardware-related factors *PlanetLab, SIGMETRICS 008*

1.7% to **8.6%** annualized disk failure rates

Google's disk failure analysis [2]

a good place to begin
USENIX: The Computer Failure Data Repository (CFDR)
www.usenix.org/cfdr

[1] "A large-scale study of failures in High-Performance Computing systems", B. Schroeder, G. Gibson et al., Dependable and Secure Computing, IEEE Transactions , vol. 7, no. 4, pp. 337–350, 2010

Predictions for exascale

MTBF ~ 20mins for exascale[1]

system-wide MTBF so small that more th
of the application's **execution time wi**
spent writing checkpoints and **recoveri**
failures[2]

ter Failure Data

dr

ms", B.
nsactions ,

Predictions for exascale

MTBF ~ 20mins for exascale[1]

system-wide MTBF so small that more than **50%** of the application's **execution time will be spent writing checkpoints** and **recovering** from **failures**[2]

[1]"*Facilitating codesign for extreme-scale systems through lightweight simulation*".Engelmann C and Lauer Fr. In Workshop on Application/Architecture Co-design for Extreme-scale Computing (AACEC), 2010.
[2] "*Modeling the impact of checkpoints on next-generation systems*". Oldfield Ron A., Arunagiri S et al., In 24th IEEE Conference on Mass Storage Systems and Technologies, pages 30-46, 2007.

Chapel's Task Parallel constructs

```
// create a fire-and-forget task
begin async_task();
reg_task();
```

```
// create one task per
statement
cobegin {
 producer(1);
 producer(2);
 consumer(1);
} // implicit join
```

```
// create a task per iteration
coforall t in 0..#numTasks {
    async_task();
} // implicit join
```

th

on construct

"invoke work on remote locales"

on Locales[1] do
    remote_task(); //on Loc1
local_task(); // on Loc0

distributed
serial

translates to a blocking fork operation

the 

factor

begin on Locales[1] do
    async_remote_task(); //on Loc1
local_task(); // on Loc0

distributed
parallel

cobegin on...
coforall on...
    translate to non-blocking fork operations

Resilient Store

maintains critical data &
survives failures

external file system
(eg.ZooKeeper)

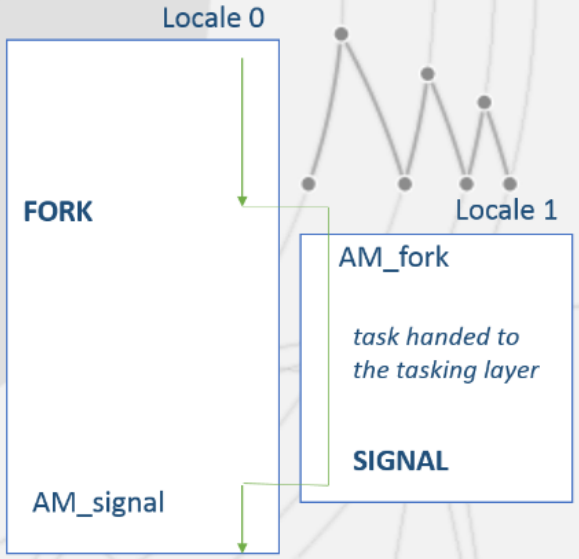
- depends on external components
- optimized existing frameworks

in-memory replication
(eg.RDD's)

- memory overhead
- management of data structures within the RTS
- faster than disk storage
- self-contained applications
- transparent to the programmer

here:
track/store **in-transit messages** on every locale
simple DS (linked lists) for task descriptors,
populated during execution

Blocking Fork



parent locale **waits on completion** of the remote task on the child locale

simplest case for recovery

data already available on the parent -> no need for replication

Blocking Fork

recovery from failure is handled on the **parent locale**

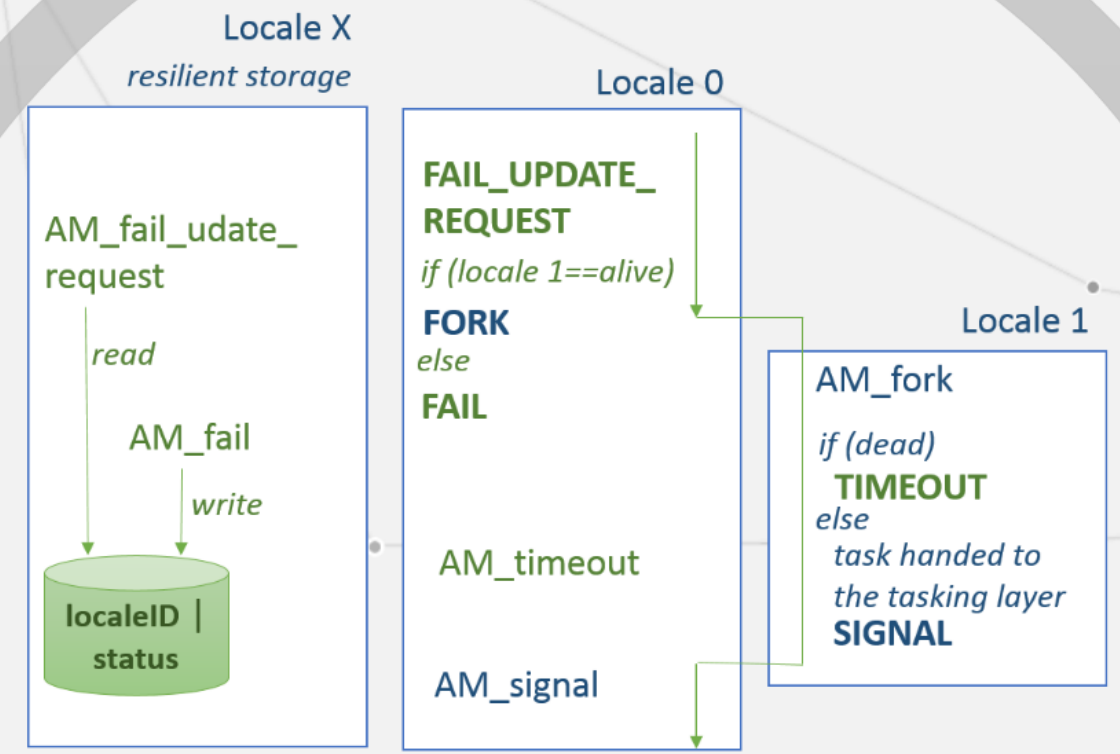
- data retrieved from local memory

locales branch out forming the LocaleTree with Locale 0 as root

parent failures can be dealt with as child failures one level higher on the tree

*failure of Locale 0 is unrecoverable

Resilient Blocking Fork[1]



AM_timeout: function performing recovery on the parent using locally available data

AM_fail(_update) : functions to query and update centrally stored (Loc 0) information on the status of nodes

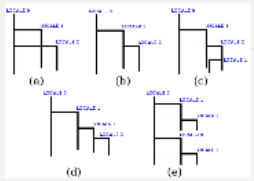
[1] "Towards Resilient Chapel: Design and implementation of a transparent resilience mechanism for Chapel", K. Panagiotopoulou and H.W. Loidl,2015, In Proceedings of the 3rd International Conference on Exascale Applications and Software (EASC '15), University of Edinburgh, Edinburgh, Scotland, UK, 86-91.

Transparently Resilient Task Parallelism for Chapel

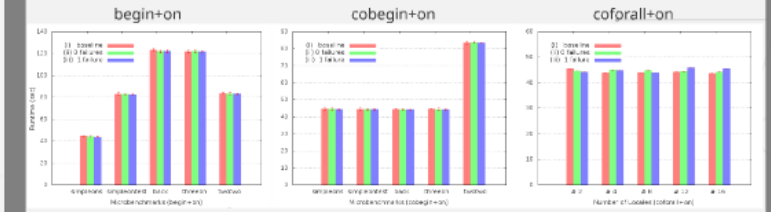
Konstantina Panagiotopoulou, Hans-Wolfgang Loidl

Heriot-Watt University, Edinburgh, UK

Evaluation Results



constructed micro-benchmarks



*System specs:16 nodes of a 32-node Beowulf cluster connected via Gigabit ethernet network, with each node consisting of two quad-core Xeon E5506 2.13GHz, 12GB of main memory and three-layered cache memory topology.

Resilience

Fault tolerance property that enables a system to continue operating properly in the event of failure

Resilience integrated design strategy to enable the fault tolerance property of the system

Transparent Resilience for Chapel

GOAL
- allow distributed long-running Chapel applications to terminate successfully in the presence of hardware failure of participating nodes
- achieve **transparent** resilience without requiring user-assistance or source code modifications

Buddy Locales

Resilient Non-Blocking Fork

Non-Blocking Fork

Failure rates

failure rates grow proportionally to the number of processor chips, despite hardware improvements

Mean Time Between Failures (MTBF)
between 1 day and 1 month
80% hardware-related factors

1.75 to 8.85 annualized disk failure ratios

Chapel's Task Parallel constructs

```
// create a fire-and-forget task
begin async_task();
reg_task();

// create one task per iteration
coforall t in 0..#numTasks {
    async_task();
} // implicit join
```

on construct

"invoke work on remote locales"

```
on Locales(1) do
  remote_task(); //on Loc1
local_task(); // on Loc0
```

translates to a blocking fork operation

```
begin on Locales(1) do
  async_remote_task(); //on Loc1
local_task(); // on Loc0
cobegin on....
coforall on....
```

translate to non-blocking fork operations

Resilient Store

manages critical data & survives failures

Blocking Fork

parent locale waits on completion of the remote task on the child locale

simplest case for recovery
data already available on the parent => no need for replication

Blocking Fork

recovery from failure is handled on the parent locale
- data retrieved from local memory

Locales branch out forming the LocaleTree with Locale 0 as root.

parent failures can be dealt with as child failures one level higher on the tree

•failure of Locale 0 is unrecoverable

Failure Detection

assumption
- a node explicitly notifies on its failure

active research on hardware-assisted prediction
- failure prediction/avoidance
  - health metrics
  - (external) factors (eg. temperature)

GASNet communication API applies strict system-wide abort policy on failures
→ failure injection/detection needs to be transparent to GASNet

State maintenance

carrying the history of past computations

2 common state-recovery techniques

- checkpointing and **forward error recovery**

obtain a new correct state and resume execution
use of **message logging** mechanisms
cost proportional to the number of failures

Limitations

Chapel's decoupled communication and tasking layers → partial ahead tracking of task completion on the parent's side

complexity of distinguishing tasks with side-effects (compiler-assisted)

currently we can perform recovery on side-effect-free programs only

Resilient Blocking Fork[1]

AM_timeout: function performing recovery on the parent using locally available data
AM_fail(_update) : functions to query and update centrally stored (Loc 0) information on the status of nodes

(1) "Towards Resilient Chapel: Design and implementation of a transparent resilience mechanism for Chapel", K. Panagiotopoulou and H.W. Loidl, 2015, In Proceedings of the Institute Autonomic Computing and Software (EASE '15), University of Edinburgh, Edinburgh, Scotland, UK, 16-17

Ongoing Work

support for Chapel's data parallel constructs

distributions (distributed memory)
Block,Cyclic,
BlockCyclic etc.

layouts (shared memory)

Focus : **parallel iterators** [*forall loop on blocked data*]

- buddy locales for redundancy and task re-execution
- under-the-hood remote memory operations

https://github.com/konsP/chapel

Future Work

- TESTING
- Optimizations
  - cost of remote operations
  - minimize added communication
- Performance improvements
  - replace linked list to tackle O(N) cost of serial lookup operations
  - use of async operations on the data structures
- Investigate portability to other predefined distributions

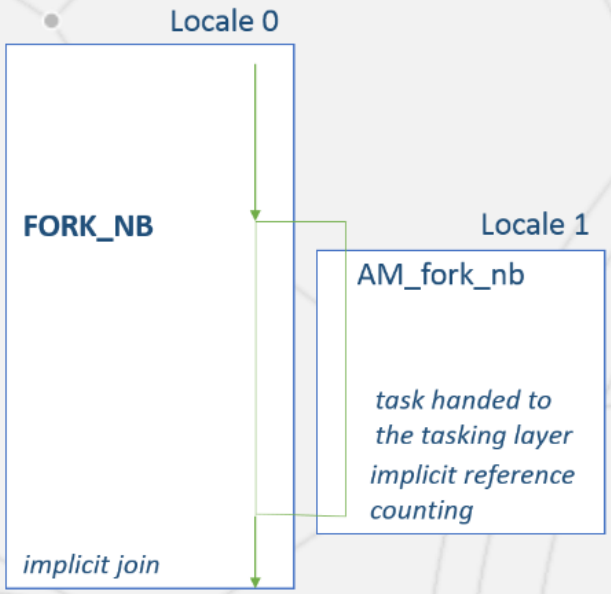
Longer term goals
- cost model of resilience across multiple configurations
- load balancing policy

Thank you!

Questions?

CHIUW'16, IPDPS'16, 23-27 May, Chicago

Non-Blocking Fork



Locale 0 (parent) may EXIT before the failure on Locale 1 (child) occurs

task descriptors on the parent's memory become **inaccessible**

→ need to store task descriptors in advance

UK

Buddy Locales[1]

idea: replicate essential data in the memory of other nodes in the system

parent-buddy communication / data sent to buddies before launching the fork operation

➡ *added communication*

buddy locale is responsible for **adoption** and **recovery**

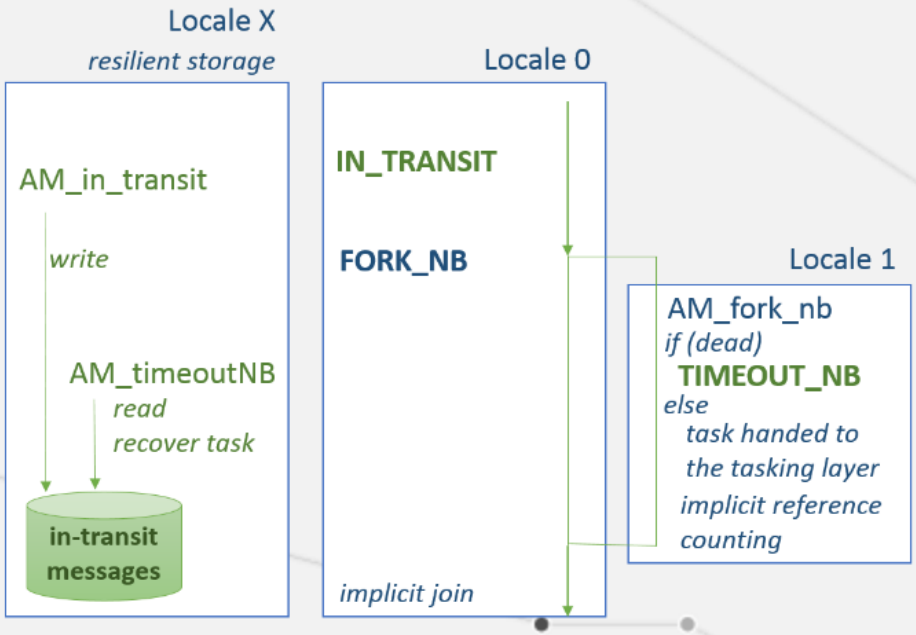
➡ *potentially unbalanced loads*

# of buddies per locale
buddy selection algorithm

➡ configurable

[1]"*Fault-tolerance techniques for high-performance computing*". Herault T, and Robert Y. Springer, 2015.

Resilient Non-Blocking Fork



IN_TRANSIT parent → buddy locale(s)
buddy stores task descriptor
before launching remote fork

TIMEOUT_NB child → buddy locale(s)
buddy retrieves task descriptor &
launches recovery

Failure Detection

assumption

a node explicitly notifies on its failure

active research [1] on hardware-assisted prediction

- failure prediction/avoidance
  - health metrics
  - external factors (eg.temperature)

such HW capability can be represented by a failure notification message on the communication layer

GASNet communication API applies strict system-wide abort policy on failures

→ failure injection/detection needs to be transparent to GASNET

[1] "*Critical Event Prediction for Proactive Management in Large-scale Computer Clusters*," R. K. Sahoo, A. J. Oliner et al., in Proceedings of the Ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. ACM, 2003, pp. 426–435.

State maintenance

carrying the history of past computations

2 common state-recovery techniques

- checkpointing and **forward error recovery**

obtain a new correct state and resume execution
use of **message logging** mechanisms
cost proportional to the number of failures

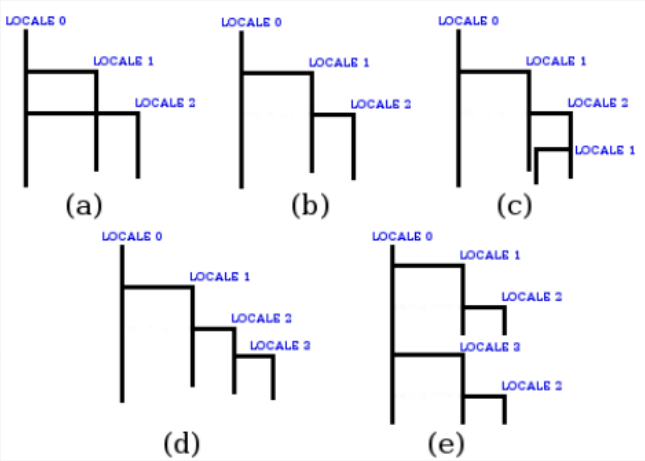
Limitations

Chapel's decoupled communication and tasking layers -> do not allow **tracking of task completion** on the parent's side

complexity of distinguishing tasks with side effects (compiler-assisted)

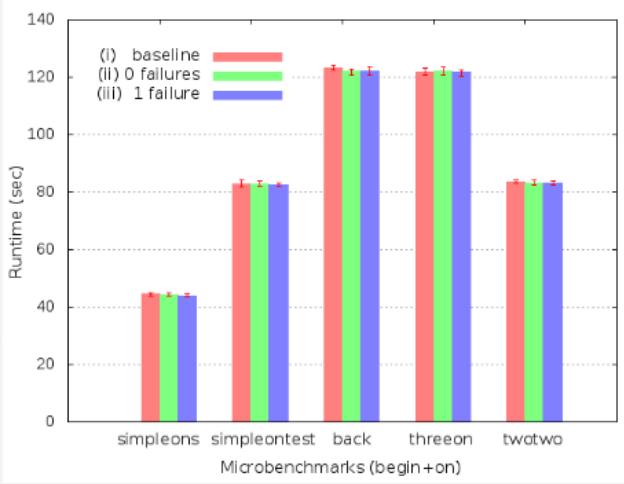
currently we can perform recovery on side-effect-free programs only

Evaluation Results

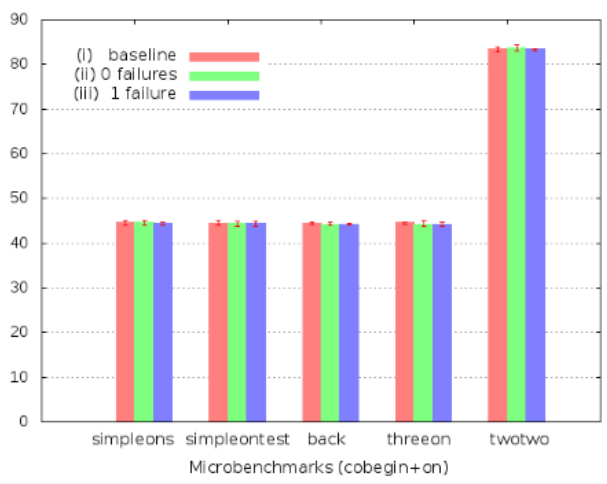


constructed micro-benchmarks

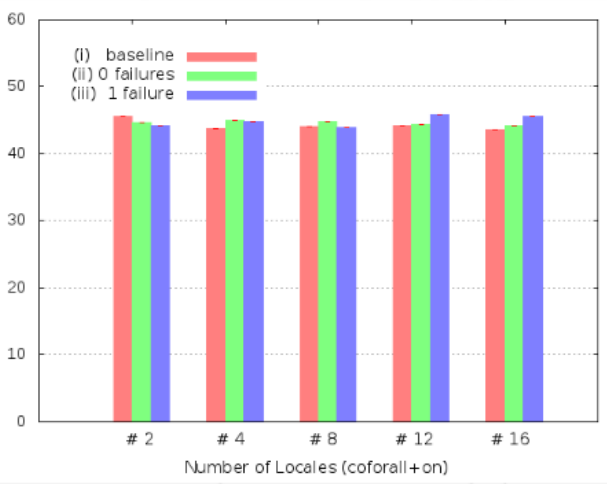
begin+on



cobegin+on

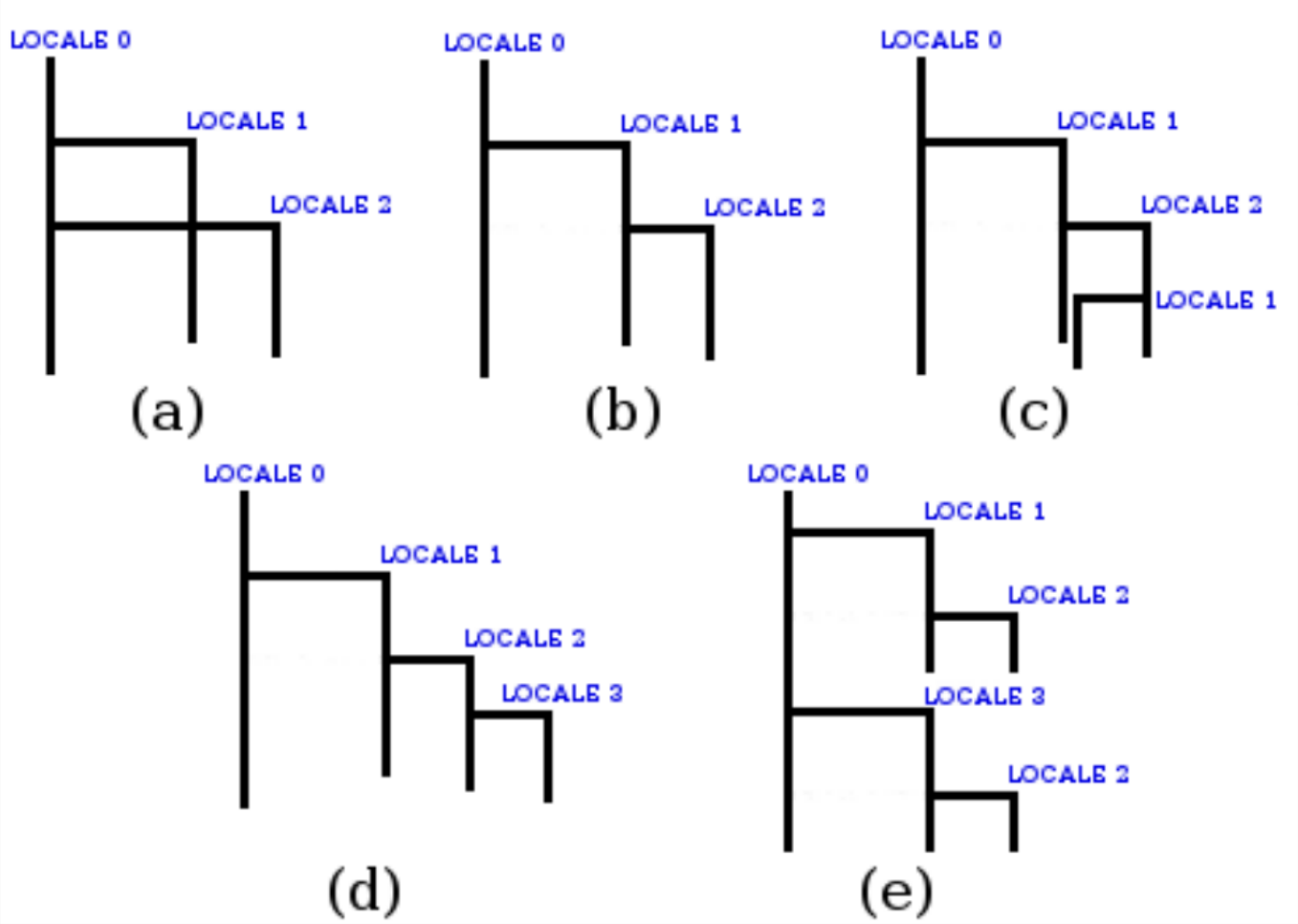


coforall+on



*System specs:16 nodes of a 32-node Beowulf cluster connected via Gigabit ethernet network, with each node consisting of two quad-core Xeon E5506 2.13GHz, 12GB of main memory and three-layered cache memory topology.

Evaluation Resul

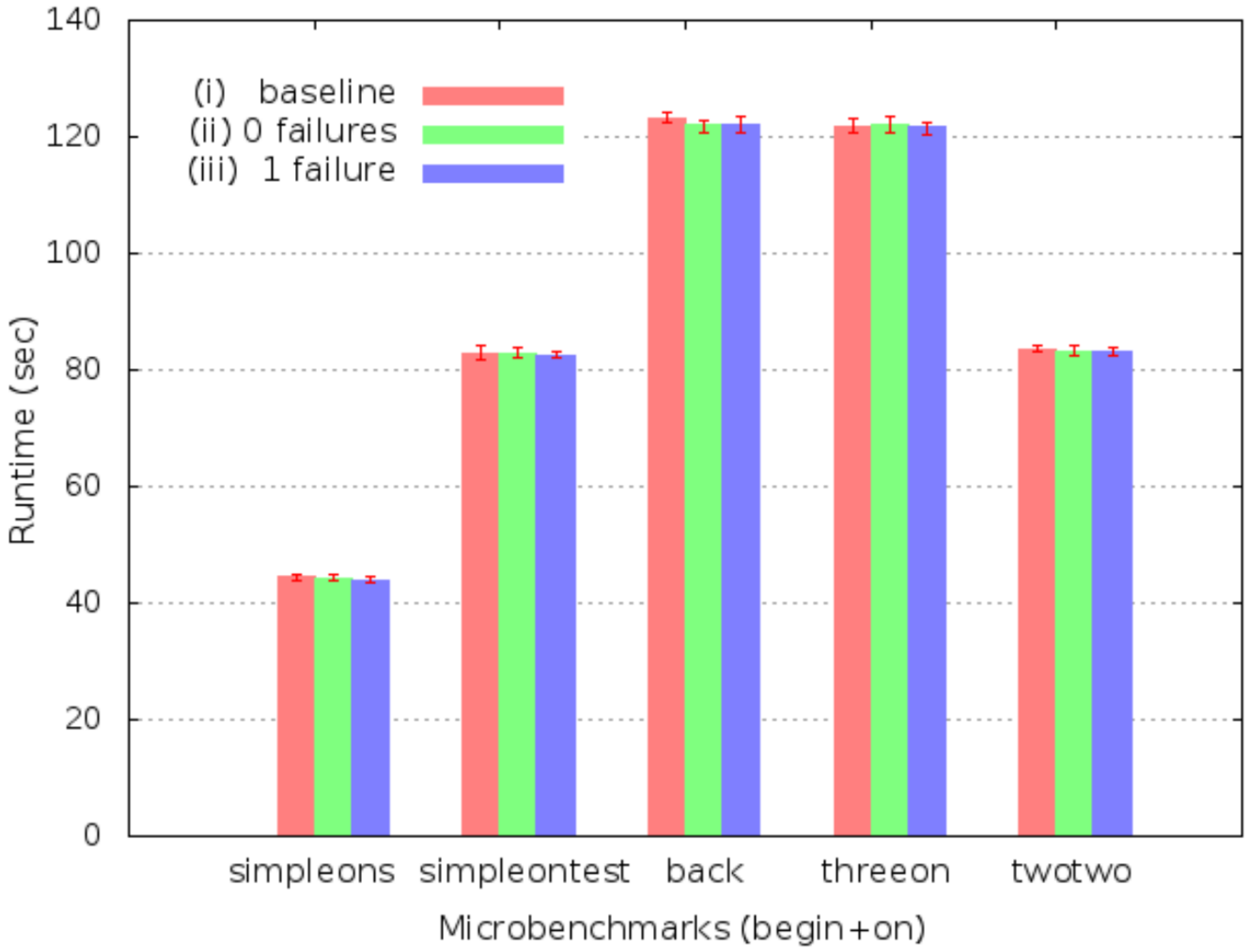


constructed
micro-benchmarks

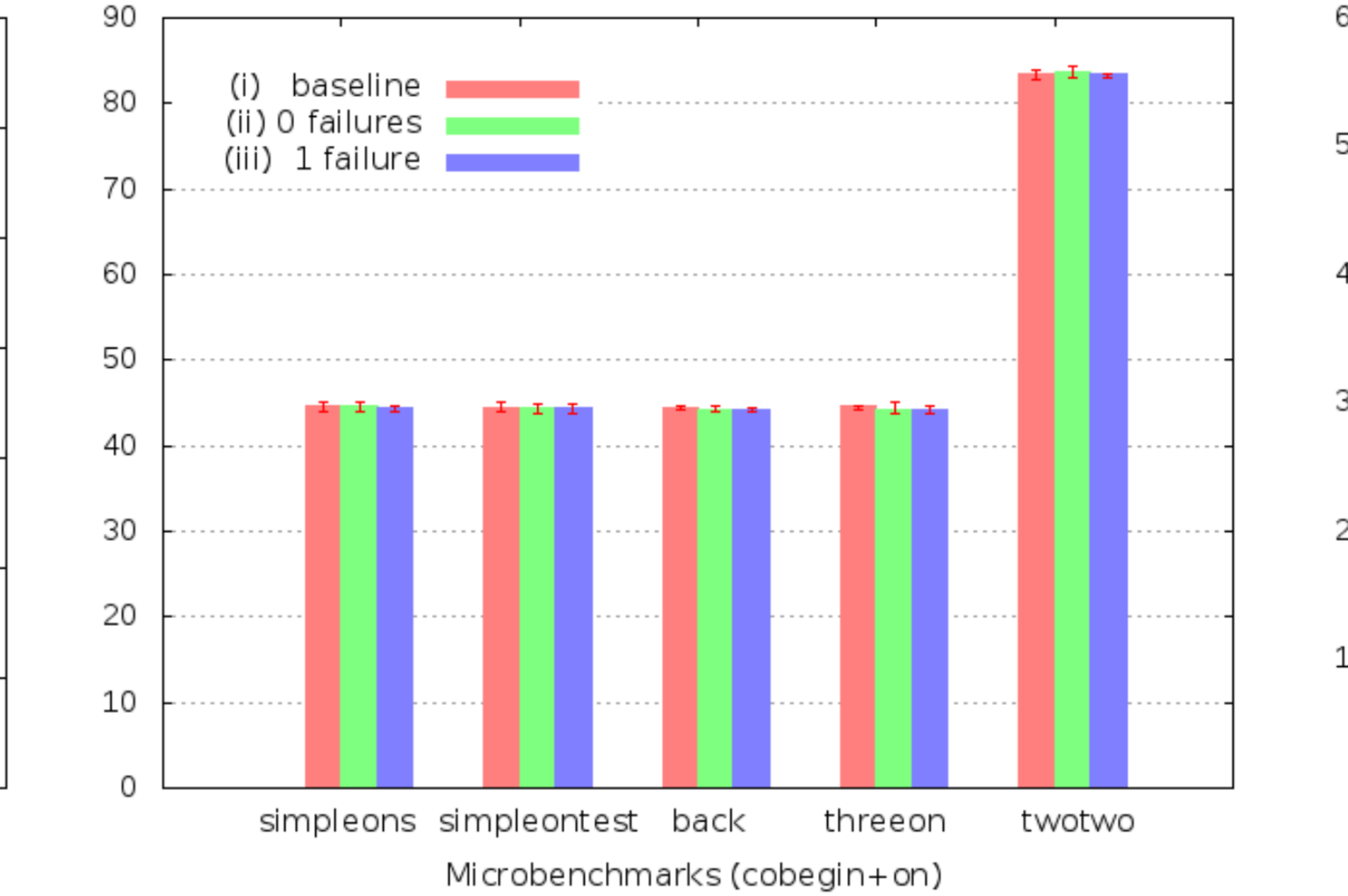
begin+on

cobegin+c

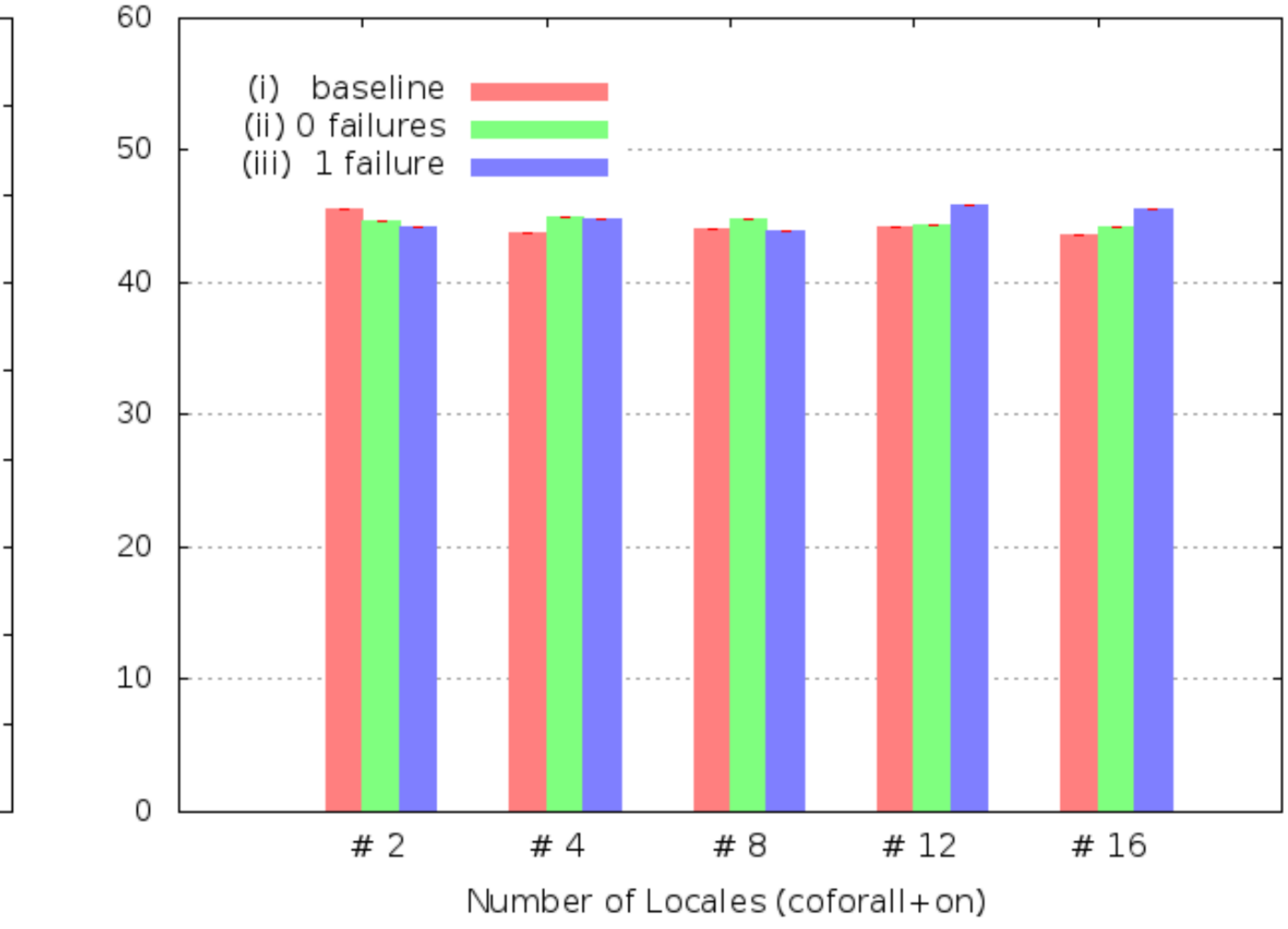
begin+on


140
120
100
80
60
40
20
0
Runtime (sec)
(i) baseline
(ii) 0 failures
(iii) 1 failure
simpleons
simpleontest
back
threeon
twotwo
Microbenchmarks (begin+on)

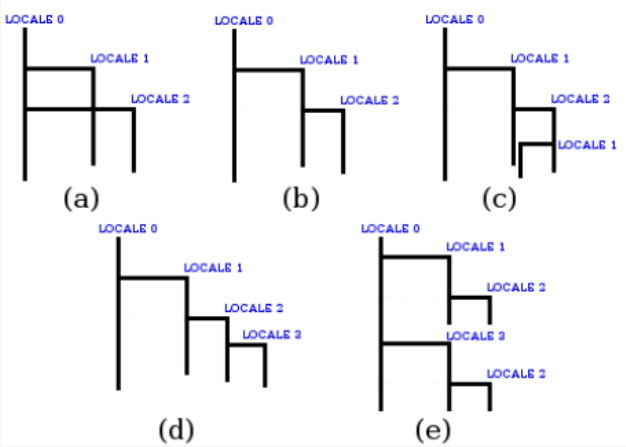
cobegin+on


(i) baseline
(ii) 0 failures
(iii) 1 failure
90
80
70
60
50
40
30
20
10
0
simpleons
simpleontest
back
threeon
twotwo
Microbenchmarks (cobegin+on)

coforall+on

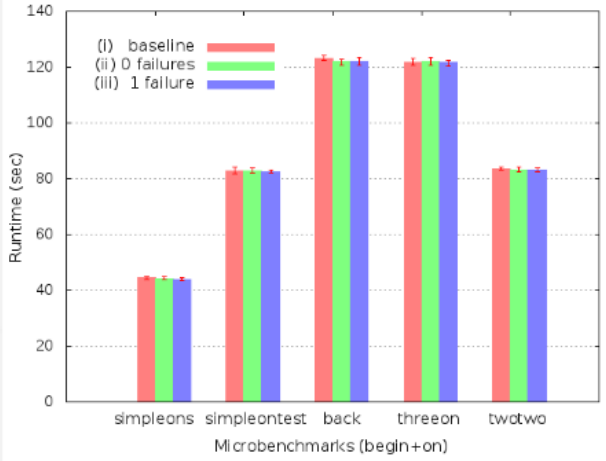

60
50
40
30
20
10
0
(i) baseline
(ii) 0 failures
(iii) 1 failure
2
4
8
12
16
Number of Locales (coforall+on)

Evaluation Results

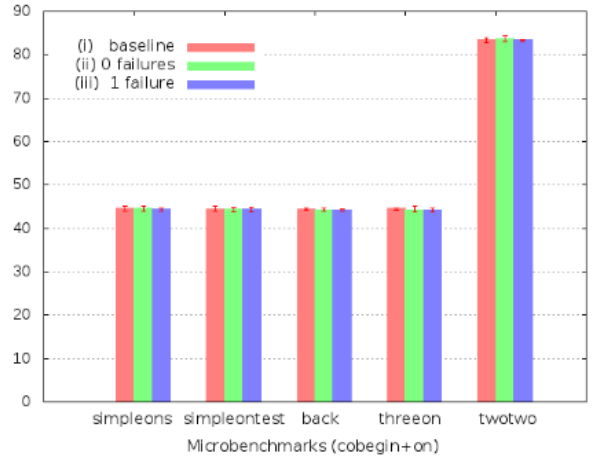


constructed micro-benchmarks

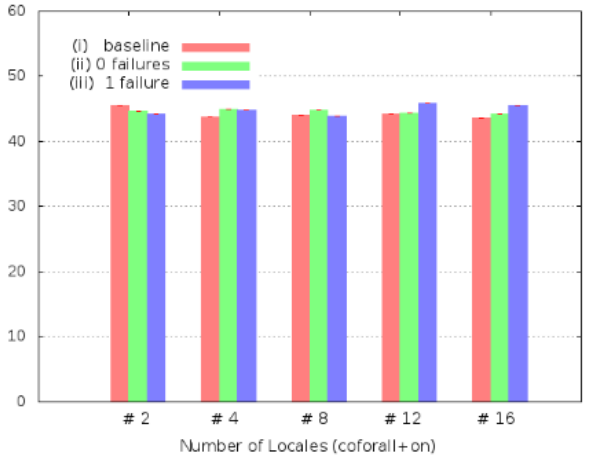
begin+on



cobegin+on



coforall+on



*System specs:16 nodes of a 32-node Beowulf cluster connected via Gigabit ethernet network, with each node consisting of two quad-core Xeon E5506 2.13GHz, 12GB of main memory and three-layered cache memory topology.

Ongoing Work

support for Chapel's data parallel constructs

distributions (distributed memory)
Block,Cyclic, BlockCyclic etc.

layouts (shared memory)

Focus : **parallel iterators** [***forall*** *loop on blocked data*]

- buddy locales for redundancy and task re-execution
- under-the-hood remote memory operations

https://github.com/konsP/chapel

Future Work

- T E S T I N G
- Optimizations
  - cost of remote operations
  - minimize added communication
- Performance improvements
  - replace linked list to tackle O(N) cost of serial lookup operations
  - use of async operations on the data structures
- Investigate portability to other predefined distributions

Longer term goals

- cost model of resilience across multiple configurations
- load balancing policy

Thank you!

Questions?

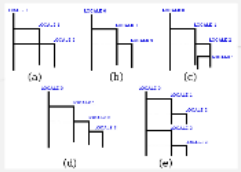
kp167@hw.ac.uk

Transparently Resilient Task Parallelism for Chapel

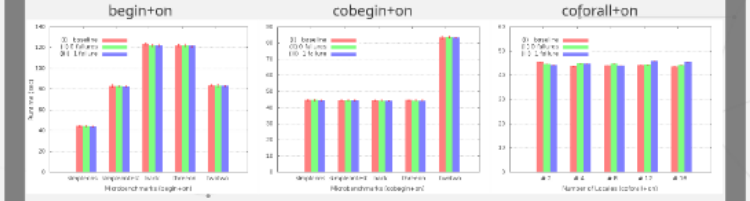
Konstantina Panagiotopoulou, Hans-Wolfgang Loidl

Heriot-Watt University, Edinburgh, UK

Evaluation Results



constructed micro-benchmarks



*System specs:16 nodes of a 32-node Beowulf cluster connected via Gigabit ethernet network, with each node consisting of two quad-core Xeon E5506 2.13GHz, 12GB of main memory and three-layered cache memory topology.

Resilience

Fault tolerance property that enables a system to continue operating properly in the event of failure

Resilience integrated design strategy to enable the fault tolerance property of the system

Transparent Resilience for Chapel

GOAL
- allow distributed long-running Chapel applications to terminate successfully in the presence of hardware failure of participating nodes
- achieve **transparent** resilience without requiring user-assistance or source code modifications

Buddy Locales

Resilient Non-Blocking Fork

Non-Blocking Fork

Failure rates

Chapel's Task Parallel constructs

```
// create a fire-and-forget task
begin async_task();
reg_task();

// create one task per statement
cobegin {
 producer(1);
 producer(2);
 consumer(1);
} // implicit join

// create a task per iteration
coforall t in 0..#numTasks {
  async_task();
} // implicit join
```

on construct

"invoke work on remote locales"

```
on Locales[1] do
  remote_task(); //on Loc1
local_task(); // on Loc0
```

translates to a blocking fork operation

```
begin on Locales[1] do
  async_remote_task(); //on Loc1
local_task(); // on Loc0
cobegin on...
coforall on...
```

translate to non-blocking fork operations

Resilient Store

maintains critical data & survives Failures

Blocking Fork

parent locale waits on completion of the remote task on the child locale

simplest case for recovery

data already available on the parent -> no need for replication

Blocking Fork

recovery from failure is handled on the parent locale
- data retrieved from local memory

locales branch out, forming the LocaleTree with Locale 0 as root

parent failures can be dealt with as child failures one level higher on the tree

- failure of Locale 0 is unrecoverable

Failure Detection

assumption: a node explicitly notifies on its failure

active research on hardware-assisted prediction
- failure prediction/avoidance
  - health metrics
  - external factors (eg. temperature)

GASNet communication API applies strict system-wide abort policy on failures

failure injection/detection needs to be transparent to GASNet

State maintenance

carrying the history of past computations

2 common state-recovery techniques
- checkpointing and forward error recovery

obtain a new correct state and resume execution

use of message logging mechanisms, cost proportional to the number of failures

Limitations

Resilient Blocking Fork[1]

AM_timeout: function performing recovery on the parent using locally available data

AM_fail(_update) : functions to query and update centrally stored (Loc 0) information on the status of nodes

[1] "Towards Resilient Chapel: Design and Implementation of a Non-Blocking Fork Mechanism for Chapel", K. Panagiotopoulou and H.-W. Loidl, 27th International Conference on Parallel Computing and Distributed Systems (PDCS'15), University of Edinburgh, Edinburgh, Scotland, UK, 2015

Ongoing Work

support for Chapel's data parallel constructs

distributions (distributed memory): Block,Cyclic, BlockCyclic etc.

layouts (shared memory)

Focus : **parallel iterators** [*forall loop on blocked data*]

- buddy locales for redundancy and task re-execution
- under-the-hood remote memory operations

https://github.com/KonstantinosP/chapel

Future Work

- TESTING
- Optimizations
  - cost of remote operations
  - minimize added communication
- Performance improvements
  - replace linked list to tackle O(N) cost of serial lookup operations
  - use of async operations on the data structures
- Investigate portability to other predefined distributions

Longer term goals
- cost model of resilience across multiple configurations
- load balancing policy

Thank you!

Questions?

CHIUW'16, IPDPS'16, 23-27 May, Chicago