Towards an Error Model for OpenMP

Michael Wong, Michael Klemm, Alejandro Duran, Tim Mattson, Grant Haab, Bronis R. de Supinski, and Andrey Churbanov

Some of the usual suspects (who have photos)

Current problems with OpenMP 3.0 Error Handling

- **Historically limited to HPC, but need to expand into industrial applications**
- **Limited by the three key requirements:**
  - Must not throw exceptions outside of parallel region
  - Single Entry Single Exit
  - Must not escape structured block
- **We will study examples and work around**
- **Offer a roadmap to design a state of the art exception handling system**
- **Offer specific recommendation for beyond 3.1, and future proposals**

What other popular concurrent languages have done

STATE OF THE ART	1 Kill, Violence is THE answer	2 Don't take NO for an answer	3 Ask politely, accept rejection	4 Set flag, let it poll
What?	Shoot First, ask question later	Fire him, but let him clean his desk	Fire him, but let him get a lawyer	Fire him, by email!
How?	Violence is not the answer because it **Randomly** corrupt states	Interrupt at well-defined points and allow handler (but target can't refuse)	Interrupt at well-defined points, allow handler, can be ignored	Target can check between well-defined points, manually, or as part of #2, #3
Pthreads	pthread_kill, pthread_cancel (async)	Pthread_cancel (deferred mode)	NA	Manual
Java	Thread.destroy, Thread.stop	NA	Thread.interrupt	Manual or Thread.interrupt
.NET	Thread.Abort	NA	Thread.interrupt	Manual or Sleep(0)
C++0x	NA	NA	NA	Manual
Why?	Avoid, unless you know for sure	OK for exception-unaware language	Good, automated for exception-aware languages	Same as #3 but need more cooperative effort

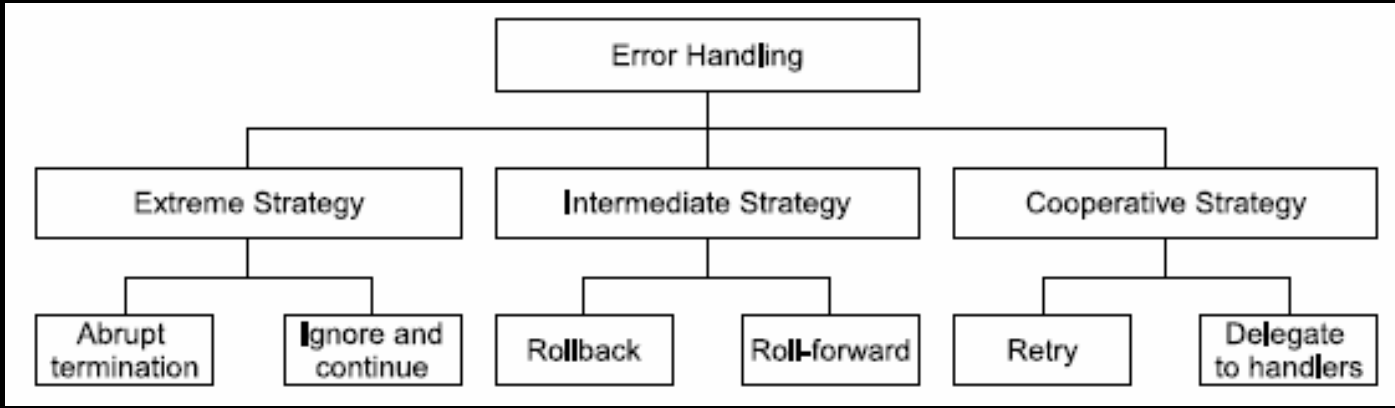
Overview of current problems and workarounds

- **Throwing an exception from a parallel region, some worksharing:**
  - Use an if flag to test for err condition, set the err and flush, record a ptr to the exception, and handle it outside of the parallel region
- **Throwing from a structured block like master directive:**
  - Break out the master directive into an if test
- **Synchronization constructs such as critical**
  - Use RAII or scope locks
- **NO WORKAROUND: tasks, sections and ordered**
- **if you want to throw an exception out of a critical-region in OpenMP - use *guard objects* (scoped locking)**
- **if you want to throw an exception out of a master region in OpenMP - use *if (omp_get_thread_num () == 0)***
- **if you want to throw an exception out of any other scope that was opened by an OpenMP-construct, you are out of luck**

Design Goals of the Exception Handling System

- **Compatible with current and possible future OpenMP base languages**
- **Provide exception handling for all base languages**
  - Exception handling is the state of the art in clean, separation of concerns, error handling
- **Support system-level and user-defined errors**
- **Flexible models that provide the best tools to handle an exception**
- **Backwards compatible with existing code**

Classification of Error Handling Strategies



- **Goal: support Extreme and Cooperative Strategy**
- **Intermediate Strategy: needs Transactional Memory support in OpenMP, and is not in our scope**
  - But is the subject of current and past research, stay tuned!
- **Step 1: provide a construct to support the Abrupt Termination pattern**
  - DONE construct will terminate an OpenMP region
- **Step 2: additionally support Ignore and continue, Retry, Delegate to handlers**
  - Studying an Error code and a Callback proposal

Done Proposal

- **Planned for beyond 3.1**
- **Allow user to Terminate innermost region**
- **Use-case: concurrent search that should stop when the first instance is found by a thread**
- **Syntax:**
  - **#pragma omp done** [clause−l i s t]
  - *clause-list* being one or more of parallel, alltasks, taskgroup
  - binding set of the done construct is the current thread team
  - applies to the innermost enclosing OpenMP construct(s) of the types specified in the clause (i. e., parallel or task).

Throwing exceptions out of parallel region

```
1 void example() {
2     try {
3 #pragma omp parallel
4         {
5 #pragma omp for
6             for (int i = 0; i < N; i++) {
7                 potentially_causes_an_exception();
8             }
9             phase_1();
10 #pragma omp barrier
11             phase_2();
12         }
13     }
14     catch (std::exception *ex) {
15         // handle exception pointed to by ex
16     }
17 }
```

```
1 void example() {
2     std::exception *ex = NULL;
3 #pragma omp parallel shared(ex)
4     {
5         ...
6 #pragma omp for
7         for (int i = 0; i < N; i++) {
8             // if an exception occurred, cease execution of the loop body
9             // (the 'if' effectively prohibits most compiler optimizations)
10 #pragma omp flush
11             if (!ex) {
12                 // catch a potential exception locally
13                 try {
14                     potentially_causes_an_exception();
15                 }
16                 catch (const std::exception *e) {
17                     // remember to handle it after the parallel region
18 #pragma omp critical
19                     ex = e;
20                 }
21             }
22         }
23 #pragma omp flush
24         // if an exception occurred, stop executing the parallel region
25         if (ex) goto termination;
26         phase_1();
27 #pragma omp barrier
28         phase_2();
29     termination:
30         ;
31     }
32     if (ex) {
33         // handle exception pointed to by ex
34     }
35 }
```

```
1 void example() {
2     std::exception *ex = NULL;
3 #pragma omp parallel shared(ex)
4     {
5         ...
6 #pragma omp for
7         for (int i = 0; i < N; i++) {
8             // no 'if' that prevents compiler optimizations
9             try {
10                 causes_an_exception();
11             }
12             catch (const std::exception *e) {
13                 // still must remember exception for later handling
14 #pragma omp critical
15                 ex = e;
16 #pragma omp done parallel for
17             }
18         }
19         phase_1();
20 #pragma omp barrier
21         phase_2();
22     }
23     // continue here if an exception is thrown in the 'for' loop
24     if (ex) {
25         // handle exception stored in ex
26     }
27 }
```

Done Example

```
1  void example() {
2      std::exception *ex = NULL;
3  #pragma omp parallel shared(ex)
4      {
5          ...
6  #pragma omp for
7          for (int i = 0; i < N; i++) {
8              // no 'if' that prevents compiler optimizations
9              try {
10                 causes_an_exception();
11             }
12             catch (const std::exception *e) {
13                 // still must remember exception for later handling
14 #pragma omp critical
15                 ex = e;
16 #pragma omp done parallel for
17             }
18         }
19         phase_1();
20 #pragma omp barrier
21         phase_2();
22     }
23     // continue here if an exception is thrown in the 'for' loop
24     if (ex) {
25         // handle exception stored in ex
26     }
27 }
```

Cancellation Points

- **Immediate termination of regions is not possible**
  - Would lead to inconsistent program state
  - Discouraged by most threading libraries
- **The done construct signals termination at (the next) cancellation point**
  - Threads need to actively check at these CPs for active termination requests
  - Possible cancellation points: barriers

Flavors of the **done** construct

Flavor	Semantics
done	abort inner-most region without restricting the type (e.g. task, for, etc.)
done parallel	terminate inner-most parallel region
done alltasks	Terminate all active and schedule tasks. Executing tasks may not create new tasks.
done taskgroup	Abort all tasks of the current task group. (May be added when OpenMP defines taskgroups.)

Error Code Proposal

- **Similar to posix**
- **Program continues at first statement following end of innermost construct when error occurs inside any OpenMP construct**
- **Any variables created or modified inside construct are undefined**
- **Error is communicated through variable shared between thread team members**
  - omp-error-var variable is of type omp_error_t
  - stores an error code that identifies whether any thread that executed the preceding OpenMP construct or runtime library routine encountered an error
  - If concurrent errors occur, the runtime system may arbitrarily select one error code and store it in the shared variable.

Error Code Proposal query

- query the value of this variable by calling a new OpenMP runtime support routine
  - omp_error_t omp_get_error (**char** * omp_err_string , **int** bufsize)
  - **Return** any value of a set of constants that are defined in the standard OpenMP include file
  - Minimal set which can be added by implementation:
    - • OMP ERR NONE
    - • OMP ERR THREAD CREATION
    - • OMP ERR THREAD FAILURE
    - • OMP ERR STACK OVERFLOW
    - • OMP ERR RUNTIME LIB
  - Also returns an implementation-defined, zero terminated string in the memory area pointed to by omp_err_string

Error Code Example

```
1 #include "omp.h"
2
3 #define BUFSIZE ...
4
5 void error_res(omp_error_t perr, char *err_str) {
6   // User-written function to clean up, report,
7   // and otherwise respond to the error
8 }
9
10 int main {
11     omp_error_t perr;
12     char * err_str;
13     int terminate = 0, nths = 16;
14     while (!terminate) {
15 #pragma omp parallel numthreads(nths)
16         {
17             ... // The body of the region
18         }
19         if ((perr = omp_get_error(err_str, BUFSIZE)) != OMP_ERR_NONE) {
20             error_res(perr, err_str);
21             if(perr == OMP_ERR_THREAD_CREATE) {
22                 nths  = (nths > 1) ? (nths -1) : 1;
23             }
24             else {
25                 printf("unrecoverable error\n");
26                 terminate = 1;
27             }
28         }
29     }
30 }
```

Callback Proposal

- **Based on previous IWOMP proposal by Duran et al, but expanded based on our discussion**
- **Use** *callback* notifications and supports both exception-aware and exception-unaware languages
- Adds an *onerror* clause that overrides OpenMP's default error-handling behavior
- handler can take any necessary actions and notify the OpenMP runtime about how to proceed with execution
- a set of default handlers that the program can specify with the *onerror* clause to implement common error responses.
- the *context* directive associates error classes and error handlers with sequential code regions to support errors that arise in OpenMP runtime routines.
- Users are not required to define any callbacks in which case the implementation will provide backward compatibility with the current *best effort* approach

Callback extensions

- This proposal extends the *onerror* proposal to meet our OpenMP error handling model requirements
- add the error class OMP USER CANCEL to associate error handlers with termination requests of done constructs
- provide the error class OMP EXCEPTION RAISED, so that error handlers can catch and handle C++ exceptions, either locally or globally by re-throwing
- exploring extensions such as specifying a default handler with an environment variable so that applications can take appropriate actions for errors that occur during initialization of the OpenMP runtime or from invalid states of internal control variables

Callback example

```
1 omp_error_action_t savedata(omp_err_info_t *error, my_data_t *data) {
2     /* save computed data */
3     return OMP_ABORT; // notify the resolution to the error
4 }
5
6 void f() {
7     my_data_t data;
8 #pragma parallel onerror(OMP_ERR_SEVERE,OMP_ERR_FATAL: savedata, &data)
9     {
10        /* parallel code */
11    }
12 }
```

Further Committee discussions since publication

- **Cancellation points**
  - Implementation defined
  - Minimal set: entry, exit of regions, critical section, loop chunk completion, runtime calls
- **Orphaned DONE and barriers?**
  - Add NoCancellation clause to Parallel region to improve optimization
- **Cancel any parallel region, by name?**
- **SHOULD NOT allow listing parallel, worksharing and task at the same time, but only one of them - outermost among those we want to terminate.**