The Parallel Patterns Library in Visual Studio 2010

Stephan T. Lavavej
Visual C++ Libraries Developer
stl@microsoft.com

Concurrency Is Hard (Let's Go Shopping)

- Problems
  - Correctness is hard
    - Bugs are notoriously difficult to avoid, find, and fix
  - Performance is hard
    - Efficiency on your hardware
    - Scalability to bigger hardware
  - Elegance is hard
    - When you have low-level primitives and nothing else
  - There is no escape
    - The future is more cores, not (many) more hertz
- Solution: Do the hard parts in libraries

Parallel Computing Platform

- Four major parts
  - Concurrency Runtime (ConcRT, "concert")
  - Parallel Patterns Library (PPL, "people")
  - Asynchronous Agents Library
  - Parallel Debugging and Profiling
- Implemented within msvcr100.dll/msvcp100.dll and libcmt.lib/libcpmt.lib
  - Nothing more to link against or redistribute
  - x86 DLLs are currently 735 + 416 = 1151 KB
  - In VS 2008 SP1 (VC9 SP1), 641 + 560 = 1201 KB
  - PPL and Agents are mostly header-only

Concurrency Runtime (ConcRT)

- What happens when a single process contains independent concurrent components?
- ConcRT…
  - Arbitrates between multiple requests for computing resources within a single process
  - Can reclaim resources when its cooperative blocking mechanisms are used
  - Is aware of locality (e.g. cores sharing caches)
  - Is aware of Windows 7 User Mode Scheduled Threads
  - Isn't used by application developers directly
  - Is used by library developers to build programming models
    - PPL, Agents, Boost 1.42.0?

Parallel Patterns Library (PPL)

- Namespace
  - `Concurrency`
- Tasks
  - `task_handle`
  - `task_group`
  - `structured_task_group`
- Algorithms
  - `parallel_invoke()`
  - `parallel_for()`
  - `parallel_for_each()`
- Containers
  - `combinable`
  - `concurrent_queue`
  - `concurrent_vector`
- Synchronization Primitives
  - `critical_section`
  - `event`
  - `reader_writer_lock`

Tasks

- Task (`task_handle`)
  - Stores a given functor
  - Represents a sub-computation that can be executed concurrently with other tasks
- Task Group (`task_group`)
  - Stores a given bunch of tasks
  - Performs a computation by executing its tasks concurrently and waiting for them to finish
- A task can use a task group to execute nested tasks
- Structured Task Group (`structured_task_group`)
  - Less overhead, more restrictions

Tasks Example: Serial Recursion

```
void quicksort(vector<int>::iterator first,
vector<int>::iterator last) {
  if (last - first < 2) { return; }
  int pivot = *first;
  auto mid1 = partition(first, last,
    [=](int elem) { return elem < pivot; });
  auto mid2 = partition( mid1, last,
    [=](int elem) { return elem == pivot; });
  quicksort(first, mid1);
  quicksort(mid2, last);
}
```

Tasks Example: Parallel Recursion

```
void quicksort(vector<int>::iterator first,
vector<int>::iterator last) {
  if (last - first < 2) { return; }
  int pivot = *first;
  auto mid1 = partition(first, last,
   [=](int elem) { return elem < pivot; });
  auto mid2 = partition( mid1, last,
   [=](int elem) { return elem == pivot; });
  task_group g;
  g.run([=] { quicksort(first, mid1); });
  g.run([=] { quicksort(mid2, last); });
  g.wait();
}
```

Tasks Example: Performance

- Intel Core 2 Quad Q9450 (Yorkfield 2.66 GHz)
- 50,000,000 elements
- Serial: 2939.85 ms
- Parallel: 1308.12 ms
- Speedup: 2.247

Algorithms: parallel_invoke()

- Takes 2 to 10 functors
- Executes them concurrently
- Waits for them to finish
- Example:

```
parallel_invoke(
   [=] { quicksort(first, mid1); },
   [=] { quicksort(mid2, last); }
);
```

parallel_invoke() Performance

- Intel Core 2 Quad Q9450 (Yorkfield 2.66 GHz)
- 50,000,000 elements
- Serial: 2939.85 ms
- task_group
  - 1308.12 ms
  - Speedup: 2.247
- parallel_invoke()
  - 1122.9 ms
  - Speedup: 2.618

Algorithms: parallel_for()

- Usage:
  - `parallel_for(first, last, functor);`
  - `parallel_for(first, last, step, functor);`
    - Requires `step > 0`
- Concurrently calls `functor` with each index in `[first, last)`

Containers: combinable

- Collects results from tasks
- Merges them into a final result
- A lock-free alternative to a shared variable
- Notable `combinable<T>` members:
  - `combinable()`
  - `combinable(generator)`
  - `T& local()`
  - `T combine(combiner)`
  - `combine_each(accumulator)`

parallel_for() / combinable Example: Serial Iteration

```
vector<int> v;

for (int i = 2; i < 5000000; ++i) {
    if (is_carmichael(i)) {
        v.push_back(i);
    }
}
```

parallel_for() / combinable Example: Parallel Iteration

```
vector<int> v;

combinable<vector<int>> c;

parallel_for(2, 5000000, [&](int i) {
    if (is_carmichael(i)) {
        c.local().push_back(i);
    }
});

c.combine_each([&](const vector<int>& sub) {
    v.insert(v.end(), sub.begin(), sub.end());
});

sort(v.begin(), v.end());
```

parallel_for() / combinable Example: Performance

- Intel Core 2 Quad Q9450 (Yorkfield 2.66 GHz)
- 4,999,998 indices
- Serial: 8679.61 ms
- Parallel: 2183.43 ms
- Speedup: 3.975

Containers: concurrent_queue And concurrent_vector

- concurrent_queue and concurrent_vector are…
  - Lock-free data structures
    - Aside: shared_ptr is also lock-free
  - Similar to queue and vector
    - NOT IDENTICAL
    - Example: vector is contiguous, concurrent_vector isn't
  - Suspiciously familiar
    - Intel Threading Building Blocks
  - Not in VS 2010 Beta 1 (VC10 Beta 1)
- concurrent_vector<T> can be better than combinable<vector<T>>
  - combinable<int>, etc. is still useful

concurrent_vector Example: Code And Performance

```
concurrent_vector<int> c;

parallel_for(2, 5000000, [&](int i) {
    if (is_carmichael(i)) {
        c.push_back(i);
    }
});

sort(c.begin(), c.end());
```

- combinable: 2183.43 ms
- concurrent_vector: 2181.02 ms

Algorithms: parallel_for_each()

- Usage:
  - `parallel_for_each(first, last, functor);`
- Concurrently calls `functor` with each element in `[first, last)`
- Accepts forward iterators (e.g. `forward_list`)
- Really likes random access iterators (e.g. `vector`)

parallel_for_each() Example: Performance

- Intel Core 2 Quad Q9450 (Yorkfield 2.66 GHz)
- 4,999,998 elements
- `forward_list`
  - Serial: 8681.78 ms
  - Parallel: 2680.05 ms
  - Speedup: 3.239
- `vector`
  - Serial: 8682.49 ms
  - Parallel: 2189.59 ms
  - Speedup: 3.965

Synchronization Primitives

- Provided by `<concrt.h>`, not `<ppl.h>`:
  - `critical_section`
  - `event`
  - `reader_writer_lock`
- These cooperative blocking mechanisms talk to ConcRT and allow it to schedule something else
- Windows API synchronization primitives still work
  - But ConcRT isn't aware of them, so they're less efficient
    - Except on Windows 7, thanks to UMS Threads
      - Still, you should prefer ConcRT's synchronization primitives

Scaling: Quad-Core Vs. Octo-Core Speedups

- Intel Core 2 Quad Q9450 (Yorkfield 2.66 GHz)
- Intel Xeon E5335 (8 cores, Clovertown 2.00 GHz)

Example	Quad	Octo
task_group	2.247	3.455
parallel_invoke()	2.618	3.857
combinable	3.975	7.991
concurrent_vector	3.980	7.988
p_f_e() forward_list	3.239	6.242
p_f_e() vector	3.965	7.965

Questions?

- For more information, see:
  - msdn.com/concurrency
  - channel9.msdn.com/tags/Parallelism
  - blogs.msdn.com/vcblog
- stl@microsoft.com

Bonus Slides!

Tasks Example (1/5)

```
C:\Temp>type quicksort.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
#include <ppl.h>
#include <windows.h>
using namespace std;
using namespace Concurrency;

long long counter() {
    LARGE_INTEGER li;
    QueryPerformanceCounter(&li);
    return li.QuadPart;
}

long long frequency() {
    LARGE_INTEGER li;
    QueryPerformanceFrequency(&li);
    return li.QuadPart;
}
```

Tasks Example (2/5)

```
void quicksort(vector<int>::iterator first, vector<int>::iterator last) {
    if (last - first < 2) {
        return;
    }

    int pivot = *first;

    auto mid1 = partition(first, last, [=](int elem) { return elem < pivot; });
    auto mid2 = partition( mid1, last, [=](int elem) { return elem == pivot; });

#ifdef USE_PPL
    task_group g;

    g.run([=] { quicksort(first, mid1); });
    g.run([=] { quicksort(mid2, last); });

    g.wait();
#else
    quicksort(first, mid1);
    quicksort(mid2, last);
#endif
}
```

Tasks Example (3/5)

```
int main() {
    vector<int> v;

    for (int k = 1, n = 1; v.size() < 50000000; ) {
        v.push_back(n);

        if (n == 1 || n > 700000000) {
            n = ++k;
        } else if (n % 2 == 0) {
            n /= 2;
        } else {
            n = 3 * n + 1;
        }
    }
```

Tasks Example (4/5)

```
    long long start = counter();

    quicksort(v.begin(), v.end());

    long long finish = counter();

    cout << (finish - start) * 1000.0 / frequency()
        << " ms" << endl;

    if (is_sorted(v.begin(), v.end())) {
        cout << "SUCCESS" << endl;
    } else {
        cout << "FAILURE" << endl;
    }
}
```

Tasks Example (5/5)

```
C:\Temp>cl /EHsc /nologo /W4 /MT /O2 /GL /D_ITERATOR_DEBUG_LEVEL=0
    /Fequicksort_serial.exe quicksort.cpp
quicksort.cpp
Generating code
Finished generating code

C:\Temp>cl /EHsc /nologo /W4 /MT /O2 /GL /D_ITERATOR_DEBUG_LEVEL=0
    /DUSE_PPL /Fequicksort_parallel.exe quicksort.cpp
quicksort.cpp
Generating code
Finished generating code

C:\Temp>quicksort_serial
2939.85 ms
SUCCESS

C:\Temp>quicksort_parallel
1308.12 ms
SUCCESS
```

parallel_for() / combinable Example (1/5)

```
C:\Temp>type carmichael.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <vector>
#include <ppl.h>
#include <windows.h>
using namespace std;
using namespace Concurrency;

long long counter() {
    LARGE_INTEGER li;
    QueryPerformanceCounter(&li);
    return li.QuadPart;
}

long long frequency() {
    LARGE_INTEGER li;
    QueryPerformanceFrequency(&li);
    return li.QuadPart;
}
```

parallel_for() / combinable Example (2/5)

```
bool is_carmichael(const int n) {
    if (n < 2) { return false; }

    int k = n;

    for (int i = 2; i <= k / i; ++i) {
        if (k % i == 0) {
            if ((k / i) % i == 0) { return false; }
            if ((n - 1) % (i - 1) != 0) { return false; }
            k /= i;
            i = 1;
        }
    }

    return k != n && (n - 1) % (k - 1) == 0;
}
```

parallel_for() / combinable Example (3/5)

```
int main() {
    vector<int> v;

    long long start = counter();

#ifdef USE_PPL
    combinable<vector<int>> c;

    parallel_for(2, 5000000, [&](int i) {
        if (is_carmichael(i)) {
            c.local().push_back(i);
        }
    });

    c.combine_each([&](const vector<int>& sub) {
        v.insert(v.end(), sub.begin(), sub.end());
    });

    sort(v.begin(), v.end());
#else
```

parallel_for() / combinable Example (4/5)

```
    for (int i = 2; i < 5000000; ++i) {
        if (is_carmichael(i)) {
            v.push_back(i);
        }
    }
#endif

    long long finish = counter();

    cout << (finish - start) * 1000.0 / frequency() << " ms" << endl;

    cout << v.size() << " Carmichael numbers found." << endl;

    cout << "First five: ";
    for_each(v.begin(), v.begin() + 5, [](int i) { cout << i << " "; });
    cout << endl;

    cout << "Last five: ";
    for_each(v.end() - 5, v.end(), [](int i) { cout << i << " "; });
    cout << endl;
}
```

parallel_for() / combinable Example (5/5)

```
C:\Temp>cl /EHsc /nologo /W4 /MT /O2 /GL /D_ITERATOR_DEBUG_LEVEL=0
    /Fecarmichael_serial.exe carmichael.cpp
carmichael.cpp
Generating code
Finished generating code

C:\Temp>cl /EHsc /nologo /W4 /MT /O2 /GL /D_ITERATOR_DEBUG_LEVEL=0 /DUSE_PPL
    /Fecarmichael_parallel.exe carmichael.cpp
carmichael.cpp
Generating code
Finished generating code

C:\Temp>carmichael_serial
8679.61 ms
74 Carmichael numbers found.
First five: 561 1105 1729 2465 2821
Last five: 4335241 4463641 4767841 4903921 4909177

C:\Temp>carmichael_parallel
2183.43 ms
74 Carmichael numbers found.
First five: 561 1105 1729 2465 2821
Last five: 4335241 4463641 4767841 4903921 4909177
```

concurrent_vector Example (1/4)

```
C:\Temp>type convector.cpp
#include <algorithm>
#include <iostream>
#include <ostream>
#include <concurrent_vector.h>
#include <ppl.h>
#include <windows.h>
using namespace std;
using namespace Concurrency;

long long counter() {
    LARGE_INTEGER li;
    QueryPerformanceCounter(&li);
    return li.QuadPart;
}

long long frequency() {
    LARGE_INTEGER li;
    QueryPerformanceFrequency(&li);
    return li.QuadPart;
}
```

concurrent_vector Example (2/4)

```
bool is_carmichael(const int n) {
    if (n < 2) { return false; }

    int k = n;

    for (int i = 2; i <= k / i; ++i) {
        if (k % i == 0) {
            if ((k / i) % i == 0) { return false; }
            if ((n - 1) % (i - 1) != 0) { return false; }
            k /= i;
            i = 1;
        }
    }

    return k != n && (n - 1) % (k - 1) == 0;
}
```

concurrent_vector Example (3/4)

```
int main() {
    long long start = counter();

    concurrent_vector<int> c;

    parallel_for(2, 5000000, [&](int i) {
        if (is_carmichael(i)) {
            c.push_back(i);
        }
    });

    sort(c.begin(), c.end());

    long long finish = counter();

    cout << (finish - start) * 1000.0 / frequency() << " ms" << endl;

    cout << c.size() << " Carmichael numbers found." << endl;
```

concurrent_vector Example (4/4)

```
    cout << "First five: ";
    for_each(c.begin(), c.begin() + 5, [](int i) { cout << i << " "; });
    cout << endl;

    cout << "Last five: ";
    for_each(c.end() - 5, c.end(), [](int i) { cout << i << " "; });
    cout << endl;
}

C:\Temp>cl /EHsc /nologo /W4 /MT /O2 /GL /D_ITERATOR_DEBUG_LEVEL=0
    convector.cpp
convector.cpp
Generating code
Finished generating code

C:\Temp>convector
2181.02 ms
74 Carmichael numbers found.
First five: 561 1105 1729 2465 2821
Last five: 4335241 4463641 4767841 4903921 4909177
```

parallel_for_each() Example (1/5)

```
C:\Temp>type pfe.cpp
#include <algorithm>
#include <forward_list>
#include <iostream>
#include <numeric>
#include <ostream>
#include <vector>
#include <concurrent_vector.h>
#include <ppl.h>
#include <windows.h>
using namespace std;
using namespace Concurrency;

long long counter() {
    LARGE_INTEGER li;
    QueryPerformanceCounter(&li);
    return li.QuadPart;
}

long long frequency() {
    LARGE_INTEGER li;
    QueryPerformanceFrequency(&li);
    return li.QuadPart;
}
```

parallel_for_each() Example (2/5)

```
bool is_carmichael(const int n) {
    if (n < 2) { return false; }

    int k = n;

    for (int i = 2; i <= k / i; ++i) {
        if (k % i == 0) {
            if ((k / i) % i == 0) { return false; }
            if ((n - 1) % (i - 1) != 0) { return false; }
            k /= i;
            i = 1;
        }
    }

    return k != n && (n - 1) % (k - 1) == 0;
}
```

parallel_for_each() Example (3/5)

```
int main() {

#ifdef FORWARD
    forward_list<int> src(4999998);
#endif

#ifdef RANDOM
    vector<int> src(4999998);
#endif

    iota(src.begin(), src.end(), 2);

    long long start = counter();
```

parallel_for_each() Example (4/5)

```
#ifdef SERIAL
    vector<int> dest;

    for_each(src.begin(), src.end(),
        [&](int i) {
            if (is_carmichael(i)) {
                dest.push_back(i);
            }
        });
#endif

#ifdef PARALLEL
    concurrent_vector<int> dest;

    parallel_for_each(src.begin(), src.end(),
        [&](int i) {
            if (is_carmichael(i)) {
                dest.push_back(i);
            }
        });

    sort(dest.begin(), dest.end());
#endif
```

parallel_for_each() Example (5/5)

```
    long long finish = counter();

    cout << (finish - start) * 1000.0 / frequency() << " ms" << endl;

    cout << dest.size() << " Carmichael numbers found." << endl;

    cout << "First five: ";
    for_each(dest.begin(), dest.begin() + 5, [](int i) { cout << i << " "; });
    cout << endl;

    cout << "Last five: ";
    for_each(dest.end() - 5, dest.end(), [](int i) { cout << i << " "; });
    cout << endl;
}

cl /EHsc /nologo /W4 /MT /O2 /GL /D_ITERATOR_DEBUG_LEVEL=0 pfe.cpp /DFORWARD /DSERIAL
    /Fepfe_forward_serial.exe
cl /EHsc /nologo /W4 /MT /O2 /GL /D_ITERATOR_DEBUG_LEVEL=0 pfe.cpp /DFORWARD /DPARALLEL
    /Fepfe_forward_parallel.exe
cl /EHsc /nologo /W4 /MT /O2 /GL /D_ITERATOR_DEBUG_LEVEL=0 pfe.cpp /DRANDOM /DSERIAL
    /Fepfe_random_serial.exe
cl /EHsc /nologo /W4 /MT /O2 /GL /D_ITERATOR_DEBUG_LEVEL=0 pfe.cpp /DRANDOM /DPARALLEL
    /Fepfe_random_parallel.exe
```

Ultrasort Example (1/7)

```
C:\Temp>type ultrasort.cpp
#include <stddef.h>
#include <algorithm>
#include <iostream>
#include <memory>
#include <ostream>
#include <tuple>
#include <vector>
#include <ppl.h>
#include <windows.h>
using namespace std;
using namespace Concurrency;

long long counter() {
    LARGE_INTEGER li;
    QueryPerformanceCounter(&li);
    return li.QuadPart;
}

long long frequency() {
    LARGE_INTEGER li;
    QueryPerformanceFrequency(&li);
    return li.QuadPart;
}
```

Ultrasort Example (2/7)

```
void ultrasort(vector<int>::iterator first, vector<int>::iterator last) {
    size_t elems = last - first;

    unsigned int procs = CurrentScheduler::Get()
        ->GetNumberOfVirtualProcessors();

    if (elems < procs) {
        sort(first, last);
        return;
    }

    size_t slice = elems / procs;

    typedef tuple<vector<int>::iterator, vector<int>::iterator,
        shared_ptr<event>> range_t;

    vector<range_t> ranges;

    task_group tasks;
```

Ultrasort Example (3/7)

```
for (unsigned int i = 0; i < procs; ++i) {
    auto a = first + slice * i;
    auto b = i + 1 < procs
        ? first + slice * (i + 1)
        : last;

    auto e = make_shared<event>();

    ranges.push_back(make_tuple(a, b, e));

    tasks.run([=] {
        sort(a, b);
        e->set();
    });
}
```

Ultrasort Example (4/7)

```
while (ranges.size() > 1) {
    vector<range_t> fused;

    for (size_t i = 0; i + 1 < ranges.size(); i += 2) {
        auto a0 = get<0>(ranges[i]);
        auto b0 = get<1>(ranges[i]);
        auto e0 = get<2>(ranges[i]);

        auto b1 = get<1>(ranges[i + 1]);
        auto e1 = get<2>(ranges[i + 1]);

        auto e = make_shared<event>();

        fused.push_back(make_tuple(a0, b1, e));

        tasks.run([=] {
            event * both[] = { e0.get(), e1.get() };
            event::wait_for_multiple(both, 2, true);
            inplace_merge(a0, b0, b1);
            e->set();
        });
    }
```

Ultrasort Example (5/7)

```
        if (ranges.size() % 2 != 0) {
            fused.push_back(ranges.back());
        }

        ranges.swap(fused);
    }

    tasks.wait();
}

int main() {
    vector<int> v;

    for (int k = 1, n = 1; v.size() < 50000000; ) {
        v.push_back(n);

        if (n == 1 || n > 700000000) {
            n = ++k;
```

Ultrasort Example (6/7)

```
        } else if (n % 2 == 0) {
            n /= 2;
        } else {
            n = 3 * n + 1;
        }
    }

    long long start = counter();

#ifdef USE_PPL
    ultrasort(v.begin(), v.end());
#else
    sort(v.begin(), v.end());
#endif

    long long finish = counter();
```

Ultrasort Example (7/7)

```
    cout << (finish - start) * 1000.0 / frequency()
        << " ms" << endl;

    if (is_sorted(v.begin(), v.end())) {
        cout << "SUCCESS" << endl;
    } else {
        cout << "FAILURE" << endl;
    }
}

C:\Temp>cl /EHsc /nologo /W4 /MT /O2 /GL /D_ITERATOR_DEBUG_LEVEL=0
    /Feultrasort_serial.exe ultrasort.cpp
ultrasort.cpp
Generating code
Finished generating code

C:\Temp>cl /EHsc /nologo /W4 /MT /O2 /GL /D_ITERATOR_DEBUG_LEVEL=0 /DUSE_PPL
    /Feultrasort_parallel.exe ultrasort.cpp
ultrasort.cpp
Generating code
Finished generating code
```

Ultrasort Example: Scaling (Or Lack Thereof)

- Intel Core 2 Quad Q9450 (Yorkfield 2.66 GHz)
  - `parallel_invoke()`: 2.618 speedup
  - `ultrasort()`: 2.416 (3286.28 ms to 1360.12 ms)
- Intel Xeon E5335 (8 cores, Clovertown 2.00 GHz)
  - `parallel_invoke()`: 3.857 speedup
  - `ultrasort()`: 3.192 (4340.89 ms to 1359.94 ms)
- Find the inefficiency!