Intel TBB Package

21 messages

JJ Allaire <jj@rstudio.com> Sat, Jan 4, 2014 at 2:16 PM
To: Simon Urbanek <simon.urbanek@r-project.org>, Dirk Eddelbuettel <edd@debian.org>

Hi Simon,

I've been experimenting to see if I can create an R package that makes Intel TBB (https://www.threadingbuildingblocks.org/) available to other packages.

Unfortunately TBB can only be built as a shared library (they do this so that there is only one piece of code per process managing compute resources, noting that static linking of OpenMP was a constant source of contention and related headaches). This seems like it might be a confounding limitation, as Writing R Extensions seems to imply that there's no way to do this robustly in a portable fashion:

http://cran.r-project.org/doc/manuals/R-exts.html#Linking-to-other-packages

This seems like it might be showstopper. Is there any way out of this box? One thing I've considered is having the tbb package do an explicit dyn.load of e.g. libtbb.dylib as part of it's .onLoad routine. No idea if this is in in-bounds or if it would even work.

Any suggestions you might have greatly appreciated!

J.J.

Simon Urbanek <simon.urbanek@r-project.org> Sun, Jan 5, 2014 at 3:30 PM
To: JJ Allaire <jj@rstudio.com>
Cc: Dirk Eddelbuettel <edd@debian.org>

[Quoted text hidden]
There are two main issues: a) so portability b) path management.

I'm not 100% sure what is the status on a) - the main problem used to be Windows, since DLLs can be linked in different ways and the participants have to agree. In addition, the way Windows resolves DLL dependencies is a horrible mess, it typically involves PATH modification gymnastics. On other platforms potential issues exist with two-level namespaces - but currently R disables them on OS X (although it is the default), so if you are looking for a pragmatic solution, it's not as bad.

b) is more of a problem - both R and packages are relocatable and it's very commonly used feature (e.g. OS X binaries get built in separate path and packages can be installed = unpacked anywhere and still have to work). Hence it is pretty much impossible to locate the dependent library if it is inside another package. Pre-loading (i.e. explicit dlopen with RTLD_GLOBAL before) only works with single-level namespaces and I have not tested it recently to see how portable it is. Currently, we defer symbol resolution to run-time on OS X, so it should work at the moment, but that is rather unusual in general.

So the above may work if you are willing to make assumptions that may change in the future. (E.g., R has been using two-level namespaces on OS X a while ago where none of the above would work).

The other option is to auto-generate the bindings. LinkingTo: works around the issues above by using dlopen explicitly. I don't know what it entails for TBB, but technically I think you should be able to simply have an init call that populates all exported symbols - you can still use regular TBB headers to actually use it, you just generate the forwarding table using R API. It will look ugly, since TBB is C++ so you will be generating mangled names, but technically, it should work. It may be an interesting exercise, since this could be used for any library

in general. I may have a look sometime next week just to see what it would look like.

Cheers,
Simon

JJ Allaire <jj@rstudio.com> Mon, Jan 6, 2014 at 8:18 AM
To: Simon Urbanek <simon.urbanek@r-project.org>
Cc: Dirk Eddelbuettel <edd@debian.org>

> The other option is to auto-generate the bindings. LinkingTo: works around the issues above by using dlopen explicitly. I don't know what it entails for TBB, but technically I think you should be able to simply have an init call that populates all exported symbols - you can still use regular TBB headers to actually use it, you just generate the forwarding table using R API. It will look ugly, since TBB is C++ so you will be generating mangled names, but technically, it should work. It may be an interesting exercise, since this could be used for any library in general. I may have a look sometime next week just to see what it would look like.

That option is definitely the most attractive since it has extremely straightforward usage semantics and fully "colors between the lines" of the R dynamic loading system. Just to be sure I understand this 100%, this would look something like:

(1) Within the tbb R library (as distinct from the core tbb library provided by Intel) use R_RegisterCCallable to register extern "C" versions of all the tbb entry points (these entry points are in turn looked up by passing the mangled C++ names to dlsym).

(2) Modify the tbb headers to bind to functions looked up by R_GetCCallable rather than using direct references to the underlying C++ symbols.

There are a couple hundred C++ entry points so this wouldn't be trivial, but it seems like it would most certainly work.

J.J.

Dirk Eddelbuettel <edd@debian.org> Mon, Jan 6, 2014 at 8:36 AM
To: JJ Allaire <jj@rstudio.com>
Cc: Simon Urbanek <simon.urbanek@r-project.org>, Dirk Eddelbuettel <edd@debian.org>

> On 6 January 2014 at 08:18, JJ Allaire wrote:
> | The other option is to auto-generate the bindings. LinkingTo: works around
> | the issues above by using dlopen explicitly. I don't know what it entails
> | for TBB, but technically I think you should be able to simply have an init
> | call that populates all exported symbols - you can still use regular TBB
> | headers to actually use it, you just generate the forwarding table using R
> | API. It will look ugly, since TBB is C++ so you will be generating mangled
> | names, but technically, it should work. It may be an interesting exercise,
> | since this could be used for any library in general. I may have a look
> | sometime next week just to see what it would look like.

I considered this for something recently and but now I can't even remember anymore what it was ...

Don't we have to fall back to C identifiers to remain portable across compilers etc? Or can we work around by letting the compiler create identifiers for "both ends" ?

> | That option is definitely the most attractive since it has extremely
> | straightforward usage semantics and fully "colors between the lines" of the R

> | dynamic loading system. Just to be sure I understand this 100%, this would look
> | something like:
> |
> | (1) Within the tbb R library (as distinct from the core tbb library provided by
> | Intel) use R_RegisterCCallable to register extern "C" versions of all the tbb
> | entry points (these entry points are in turn looked up by passing the mangled
> | C++ names to dlsym).
> |
> | (2) Modify the tbb headers to bind to functions looked up by R_GetCCallable
> | rather than using direct references to the underlying C++ symbols.
> |
> | There are a couple hundred C++ entry points so this wouldn't be trivial, but it
> | seems like it would most certainly work.
>
> Certainly worth a tester.
>
> "Eventually" this may also help us making Boost Headers a not-just-headers
> library, no?
>
> And R will turn into a full-blown OS, or virtual machine :)
>
> Dirk, not amused about the -14 F outside
>
> --
> Dirk Eddelbuettel | edd@debian.org | http://dirk.eddelbuettel.com

---

JJ Allaire <jj@rstudio.com> Mon, Jan 6, 2014 at 9:43 AM
To: Dirk Eddelbuettel <edd@debian.org>
Cc: Simon Urbanek <simon.urbanek@r-project.org>

> Looking at this further, I noticed that TBB has an implementation of a dynamic loader available as a "community preview" feature:
>
> http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm
>
> In this case the client links against a static library that implements the dynamic loading. At first glance this looks like it should work with an R package. I'll investigate further and let you both know what I find out....
>
> J.J.
> [Quoted text hidden]

JJ Allaire <jj@rstudio.com> Mon, Jan 6, 2014 at 10:28 AM
To: Dirk Eddelbuettel <edd@debian.org>
Cc: Simon Urbanek <simon.urbanek@r-project.org>

> False alarm, it looks like the dynamic loader is only implemented for Windows and even at that doesn't support mingw :-(
>
> J.J.
> [Quoted text hidden]

Simon Urbanek <simon.urbanek@r-project.org> Mon, Jan 6, 2014 at 10:39 AM
To: JJ Allaire <jj@rstudio.com>
Cc: Dirk Eddelbuettel <edd@debian.org>

On Jan 6, 2014, at 8:18 AM, JJ Allaire <jj@rstudio.com> wrote:

> The other option is to auto-generate the bindings. LinkingTo: works around the issues above by using dlopen explicitly. I don't know what it entails for TBB, but technically I think you should be able to simply have an init call that populates all exported symbols - you can still use regular TBB headers to actually use it, you just generate the forwarding table using R API. It will look ugly, since TBB is C++ so you will be generating mangled names, but technically, it should work. It may be an interesting exercise, since this could be used for any library in general. I may have a look sometime next week just to see what it would look like.
>
> That option is definitely the most attractive since it has extremely straightforward usage semantics and fully "colors between the lines" of the R dynamic loading system. Just to be sure I understand this 100%, this would look something like:
>
> (1) Within the tbb R library (as distinct from the core tbb library provided by Intel) use R_RegisterCCallable to register extern "C" versions of all the tbb entry points (these entry points are in turn looked up by passing the mangled C++ names to dlsym).
>

Yes

> (2) Modify the tbb headers to bind to functions looked up by R_GetCCallable rather than using direct references to the underlying C++ symbols.
>

No, that's not needed. You would just add an extra .c file that includes a generated header from the TBB R package that populates all symbols. That header would be only used once in that file so that file would be providing all symbols, the actual code in C++ files would use regular C++ headers directly, the trick here is that the symbols would be populated by that init function independent of the C++ code. The only thing that I'm not sure if it would work properly are static C++ initializers, since we'd have to make sure that our init hack gets called before any other initializers.

> There are a couple hundred C++ entry points so this wouldn't be trivial, but it seems like it would most certainly work.
>

The nice thing about the above (just using a .c shim) is that it can be entirely auto-generated at install time of the TBB package. It is compiler-dependent, though, so both the TBB shim package and the package that uses TBB must be compiled using the same tools, but that's a reasonable requirement anyway.

Cheers,
Simon

JJ Allaire <jj@rstudio.com> Mon, Jan 6, 2014 at 12:23 PM
To: Simon Urbanek <simon.urbanek@r-project.org>
Cc: Dirk Eddelbuettel <edd@debian.org>

I think I'm not quite understanding the entire mechanism (but I think I'm close!). Let's imagine I just want to export a single function (tbb::internal::runtime_warning) using the mechanism you described. Here's my understanding of what the tbb package's init function would be:

```
extern "C" void R_init_tbb(DllInfo *dll) {

   void* libtbb = dlopen("<path-to>/libtbb.dylib", RTLD_LOCAL);

   R_RegisterCCallable(
      "tbb",
      "__ZN3tbb8internal15runtime_warningEPKcz",
      (DL_FUNC)dlsym(libtbb, "__ZN3tbb8internal15runtime_warningEPKcz"));
```

```
}
```

Note that I use RTLD_LOCAL to ensure that we aren't relying on the side effects of RTLD_GLOBAL.

The code using the library would look like this:

```
#include <tbb/tbb_stddef.h>

void foo() {
   tbb::internal::runtime_warning("unexpected state!");
}
```

This could _would not_ define any linking dependency on any library (neither the tbb package nor libtbb.dylib), rather it would link internally to stubs that are looked up via R_GetCCallable. What I'm having trouble with is what these declarations look like.

What I'm having trouble with is the .c file that includes these stubs. Here's a guess for what the stub might look like:

```
extern "C" DL_FUNC __ZN3tbb8internal15runtime_warningEPKcz = R_GetCCallable("tbb",
"__ZN3tbb8internal15runtime_warningEPKcz");
```

Alas when I use this the dyn.load call has this error:

Symbol not found: __ZN3tbb8internal15runtime_warningEPKcz

[Quoted text hidden]

Simon Urbanek <simon.urbanek@r-project.org> Mon, Jan 6, 2014 at 4:07 PM
To: JJ Allaire <jj@rstudio.com>
Cc: Dirk Eddelbuettel <edd@debian.org>

JJ,

you were indeed very close. The only remaining issue are prefix underscores. Symbol "x" is saved as "_x" in binary, so instead of __ZN3tbb8internal15runtime_warningEPKcz you have to use _ZN3tbb8internal15runtime_warningEPKcz and all will be working. nm dumps symbols as-is so with the leading underscores - the actual names are without it.

One mildly annoying issue is that dlsym() sometimes adds the underscore and sometimes doesn't so you may try "x" first and fall back to "_x" if you get NULL -- or have a configure test for that -- but the C code symbol is always without the underscore.

BTW: to maintain consistency, I'd put the extern ... R_GetCCallable(..) forest in a header file inside the tbb package.

Cheers,
Simon
[Quoted text hidden]

JJ Allaire <jj@rstudio.com> Mon, Jan 6, 2014 at 4:53 PM
To: Simon Urbanek <simon.urbanek@r-project.org>
Cc: Dirk Eddelbuettel <edd@debian.org>

Thanks! I don't have it working yet (we crash when attempting to call the C++ function) but hopefully I'm still very close. Here's the code inside the tbb package:

```
#include <R_ext/Rdynload.h>

#include <dlfcn.h>

extern "C" void R_init_tbb(DllInfo *dll) {

   void* libtbb = dlopen("/Library/Frameworks/R.framework/Versions/3.0/
Resources/library/tbb/libs/libtbb.dylib", RTLD_LOCAL);

   R_RegisterCCallable(
      "tbb",
      "_ZN3tbb8internal15runtime_warningEPKcz",
      (DL_FUNC)dlsym(libtbb, "_ZN3tbb8internal15runtime_warningEPKcz"));
}
```

Here's the code inside a package that depends on the tbb package:

```
#include <R_ext/Rdynload.h>
extern "C" DL_FUNC _ZN3tbb8internal15runtime_warningEPKcz = R_GetCCallable("tbb",
"_ZN3tbb8internal15runtime_warningEPKcz");

#include <tbb/tbb_stddef.h>

// [[Rcpp::export]]
void testTbb() {
   tbb::internal::runtime_warning("unexpected condition");
}
```

Everything builds and links fine and the depending package loads. The call to R_GetCCallable also definitely works (it's not in the code above but I checked that it's returning a real function pointer). When I call the C++ function it's bye-bye though.

Any thoughts on what else might be amiss? (if you want to punt on helping out at this point that's fine with me as I realize we are quite a ways afield from a mainstream loading/linking scenario ;-)

Here's the crash info:

```
Exception Type:  EXC_BAD_ACCESS (SIGBUS)
Exception Codes: KERN_PROTECTION_FAILURE at 0x0000000105d3e7a0

VM Regions Near 0x105d3e7a0:
    __TEXT                 0000000105d3a000-0000000105d3e000 [   16K] r-x/rwx
SM=COW  /Library/Frameworks/R.framework/Versions/3.0/Resources/library/TbbTest/
libs/TbbTest.so
```

```
--> __DATA                  0000000105d3e000-0000000105d3f000 [    4K] rw-/rwx
SM=COW  /Library/Frameworks/R.framework/Versions/3.0/Resources/library/TbbTest/
libs/TbbTest.so
    __LINKEDIT              0000000105d3f000-0000000105d44000 [   20K] r--/rwx
SM=COW  /Library/Frameworks/R.framework/Versions/3.0/Resources/library/TbbTest/
libs/TbbTest.so

Thread 0 Crashed:: Dispatch queue: com.apple.main-thread
0   ???                             0x0000000105d3e7a0 tbb::internal::runtime_
warning(char const*, ...) + 0
1   libR.dylib                      0x00000001053fb5c0 do_dotcall + 30912
(dotcode.c:578)
2   libR.dylib                      0x000000010541c2ad Rf_eval + 1181
(eval.c:642)
3   libR.dylib                      0x0000000105425bf4 Rf_evalList + 132
(eval.c:2001)
4   libR.dylib                      0x000000010541c1cd Rf_eval + 957
(eval.c:631)
5   libR.dylib                      0x0000000105428a70 do_begin + 384
(eval.c:1573)
6   libR.dylib                      0x000000010541c11f Rf_eval + 783
(eval.c:614)
7   libR.dylib                      0x000000010542635e Rf_applyClosure + 1422
(eval.c:1019)
8   libR.dylib                      0x000000010541c34e Rf_eval + 1342
(eval.c:661)
9   libR.dylib                      0x0000000105449fcf Rf_ReplIteration + 879
(main.c:258)
10  libR.dylib                      0x000000010544a643 run_Rmainloop + 195
(main.c:307)
```

[Quoted text hidden]

JJ Allaire <jj@rstudio.com> Fri, Jan 17, 2014 at 10:08 PM
To: Hadley Wickham <hadley@rstudio.com>, Romain Francois <romain@r-enthusiasts.com>

Forwarded conversation

Subject: **Intel TBB Package**

From: **JJ Allaire** <jj@rstudio.com>
Date: Sat, Jan 4, 2014 at 2:16 PM
To: Simon Urbanek <simon.urbanek@r-project.org>, Dirk Eddelbuettel <edd@debian.org>

Hi Simon,

I've been experimenting to see if I can create an R package that makes Intel TBB (https://www.threadingbuildingblocks.org/) available to other packages.

Unfortunately TBB can only be built as a shared library (they do this so that there is only one piece of code per process managing compute resources, noting that static linking of OpenMP was a constant source of contention and related headaches). This seems like it might be a confounding limitation, as Writing R Extensions seems to imply that there's no way to do this robustly in a portable fashion:

http://cran.r-project.org/doc/manuals/R-exts.html#Linking-to-other-packages

This seems like it might be showstopper. Is there any way out of this box? One thing I've considered is having the tbb package do an explicit dyn.load of e.g. libtbb.dylib as part of it's .onLoad routine. No idea if this is in in-bounds or if it would even work.

Any suggestions you might have greatly appreciated!

J.J.

From: **Simon Urbanek** <simon.urbanek@r-project.org>
Date: Sun, Jan 5, 2014 at 3:30 PM
To: JJ Allaire <jj@rstudio.com>
Cc: Dirk Eddelbuettel <edd@debian.org>

There are two main issues: a) so portability b) path management.

I'm not 100% sure what is the status on a) - the main problem used to be Windows, since DLLs can be linked in different ways and the participants have to agree. In addition, the way Windows resolves DLL dependencies is a horrible mess, it typically involves PATH modification gymnastics. On other platforms potential issues exist with two-level namespaces - but currently R disables them on OS X (although it is the default), so if you are looking for a pragmatic solution, it's not as bad.

b) is more of a problem - both R and packages are relocatable and it's very commonly used feature (e.g. OS X binaries get built in separate path and packages can be installed = unpacked anywhere and still have to work). Hence it is pretty much impossible to locate the dependent library if it is inside another package. Pre-loading (i.e. explicit dlopen with RTLD_GLOBAL before) only works with single-level namespaces and I have not tested it recently to see how portable it is. Currently, we defer symbol resolution to run-time on OS X, so it should work at the moment, but that is rather unusual in general.

So the above may work if you are willing to make assumptions that may change in the future. (E.g., R has been using two-level namespaces on OS X a while ago where none of the above would work).

The other option is to auto-generate the bindings. LinkingTo: works around the issues above by using dlopen explicitly. I don't know what it entails for TBB, but technically I think you should be able to simply have an init call that populates all exported symbols - you can still use regular TBB headers to actually use it, you just generate the forwarding table using R API. It will look ugly, since TBB is C++ so you will be generating mangled names, but technically, it should work. It may be an interesting exercise, since this could be used for any library in general. I may have a look sometime next week just to see what it would look like.

Cheers,
Simon

From: **JJ Allaire** <jj@rstudio.com>
Date: Mon, Jan 6, 2014 at 8:18 AM
To: Simon Urbanek <simon.urbanek@r-project.org>
Cc: Dirk Eddelbuettel <edd@debian.org>

> The other option is to auto-generate the bindings. LinkingTo: works around the issues above by using dlopen explicitly. I don't know what it entails for TBB, but technically I think you should be able to simply have an init call that populates all exported symbols - you can still use regular TBB headers to actually use it, you just generate the forwarding table using R API. It will look ugly, since TBB is C++ so you will be generating mangled names, but technically, it should work. It may be an interesting exercise, since this could be used for any library in general. I may have a look sometime next week just to see what it would look like.

That option is definitely the most attractive since it has extremely straightforward usage semantics and fully "colors between the lines" of the R dynamic loading system. Just to be sure I understand this 100%, this would look something like:

(1) Within the tbb R library (as distinct from the core tbb library provided by Intel) use R_RegisterCCallable to register extern "C" versions of all the tbb entry points (these entry points are in turn looked up by passing the mangled C++ names to dlsym).

(2) Modify the tbb headers to bind to functions looked up by R_GetCCallable rather than using direct references to the underlying C++ symbols.

There are a couple hundred C++ entry points so this wouldn't be trivial, but it seems like it would most certainly work.

J.J.

From: **Dirk Eddelbuettel** <edd@debian.org>
Date: Mon, Jan 6, 2014 at 8:36 AM
To: JJ Allaire <jj@rstudio.com>
Cc: Simon Urbanek <simon.urbanek@r-project.org>, Dirk Eddelbuettel <edd@debian.org>

On 6 January 2014 at 08:18, JJ Allaire wrote:
| The other option is to auto-generate the bindings. LinkingTo: works around
| the issues above by using dlopen explicitly. I don't know what it entails
| for TBB, but technically I think you should be able to simply have an init
| call that populates all exported symbols - you can still use regular TBB
| headers to actually use it, you just generate the forwarding table using R
| API. It will look ugly, since TBB is C++ so you will be generating mangled
| names, but technically, it should work. It may be an interesting exercise,
| since this could be used for any library in general. I may have a look
| sometime next week just to see what it would look like.

I considered this for something recently and but now I can't even remember anymore what it was ...

Don't we have to fall back to C identifiers to remain portable across compilers etc? Or can we work around by letting the compiler create identifiers for "both ends" ?

| That option is definitely the most attractive since it has extremely
| straightforward usage semantics and fully "colors between the lines" of the R
| dynamic loading system. Just to be sure I understand this 100%, this would look
| something like:
|
| (1) Within the tbb R library (as distinct from the core tbb library provided by
| Intel) use R_RegisterCCallable to register extern "C" versions of all the tbb
| entry points (these entry points are in turn looked up by passing the mangled
| C++ names to dlsym).
|
| (2) Modify the tbb headers to bind to functions looked up by R_GetCCallable
| rather than using direct references to the underlying C++ symbols.
|
| There are a couple hundred C++ entry points so this wouldn't be trivial, but it
| seems like it would most certainly work.

Certainly worth a tester.

"Eventually" this may also help us making Boost Headers a not-just-headers
library, no?

And R will turn into a full-blown OS, or virtual machine :)

Dirk, not amused about the -14 F outside

--
Dirk Eddelbuettel | edd@debian.org | http://dirk.eddelbuettel.com

---
From: **JJ Allaire** <jj@rstudio.com>
Date: Mon, Jan 6, 2014 at 9:43 AM
To: Dirk Eddelbuettel <edd@debian.org>
Cc: Simon Urbanek <simon.urbanek@r-project.org>

Looking at this further, I noticed that TBB has an implementation of a dynamic loader available as a "community preview" feature:

http://software.intel.com/sites/products/documentation/doclib/tbb_sa/help/index.htm

In this case the client links against a static library that implements the dynamic loading. At first glance this looks like it should work with an R package. I'll investigate further and let you both know what I find out....

J.J.

From: **JJ Allaire** <jj@rstudio.com>
Date: Mon, Jan 6, 2014 at 10:28 AM
To: Dirk Eddelbuettel <edd@debian.org>
Cc: Simon Urbanek <simon.urbanek@r-project.org>

False alarm, it looks like the dynamic loader is only implemented for Windows and even at that doesn't support mingw :-(

J.J.

From: **Simon Urbanek** <simon.urbanek@r-project.org>
Date: Mon, Jan 6, 2014 at 10:39 AM
To: JJ Allaire <jj@rstudio.com>
Cc: Dirk Eddelbuettel <edd@debian.org>

On Jan 6, 2014, at 8:18 AM, JJ Allaire <jj@rstudio.com> wrote:

> The other option is to auto-generate the bindings. LinkingTo: works around the issues above by using dlopen explicitly. I don't know what it entails for TBB, but technically I think you should be able to simply have an init call that populates all exported symbols - you can still use regular TBB headers to actually use it, you just generate the forwarding table using R API. It will look ugly, since TBB is C++ so you will be generating mangled names, but technically, it should work. It may be an interesting exercise, since this could be used for any library in general. I may have a look sometime next week just to see what it would look like.
>
> That option is definitely the most attractive since it has extremely straightforward usage semantics and fully "colors between the lines" of the R dynamic loading system. Just to be sure I understand this 100%, this would look something like:
>

> (1) Within the tbb R library (as distinct from the core tbb library provided by Intel) use R_RegisterCCallable to register extern "C" versions of all the tbb entry points (these entry points are in turn looked up by passing the mangled C++ names to dlsym).
>

Yes

> (2) Modify the tbb headers to bind to functions looked up by R_GetCCallable rather than using direct references to the underlying C++ symbols.
>

No, that's not needed. You would just add an extra .c file that includes a generated header from the TBB R package that populates all symbols. That header would be only used once in that file so that file would be providing all symbols, the actual code in C++ files would use regular C++ headers directly, the trick here is that the symbols would be populated by that init function independent of the C++ code. The only thing that I'm not sure if it would work properly are static C++ initializers, since we'd have to make sure that our init hack gets called before any other initializers.

> There are a couple hundred C++ entry points so this wouldn't be trivial, but it seems like it would most certainly work.
>

The nice thing about the above (just using a .c shim) is that it can be entirely auto-generated at install time of the TBB package. It is compiler-dependent, though, so both the TBB shim package and the package that uses TBB must be compiled using the same tools, but that's a reasonable requirement anyway.

Cheers,
Simon

From: **JJ Allaire** <jj@rstudio.com>
Date: Mon, Jan 6, 2014 at 12:23 PM
To: Simon Urbanek <simon.urbanek@r-project.org>
Cc: Dirk Eddelbuettel <edd@debian.org>

I think I'm not quite understanding the entire mechanism (but I think I'm close!). Let's imagine I just want to export a single function (tbb::internal::runtime_warning) using the mechanism you described. Here's my understanding of what the tbb package's init function would be:

```
extern "C" void R_init_tbb(DllInfo *dll) {

   void* libtbb = dlopen("<path-to>/libtbb.dylib", RTLD_LOCAL);

   R_RegisterCCallable(
      "tbb",
      "__ZN3tbb8internal15runtime_warningEPKcz",
      (DL_FUNC)dlsym(libtbb, "__ZN3tbb8internal15runtime_warningEPKcz"));

}
```

Note that I use RTLD_LOCAL to ensure that we aren't relying on the side effects of RTLD_GLOBAL.

The code using the library would look like this:

```
#include <tbb/tbb_stddef.h>
```

```
void foo() {
   tbb::internal::runtime_warning("unexpected state!");
}
```

This could _would not_ define any linking dependency on any library (neither the tbb package nor libtbb.dylib), rather it would link internally to stubs that are looked up via R_GetCCallable. What I'm having trouble with is what these declarations look like.

What I'm having trouble with is the .c file that includes these stubs. Here's a guess for what the stub might look like:

```
extern "C" DL_FUNC __ZN3tbb8internal15runtime_warningEPKcz = R_GetCCallable("tbb",
"__ZN3tbb8internal15runtime_warningEPKcz");
```

Alas when I use this the dyn.load call has this error:

Symbol not found: __ZN3tbb8internal15runtime_warningEPKcz

From: **Simon Urbanek** <simon.urbanek@r-project.org>
Date: Mon, Jan 6, 2014 at 4:07 PM
To: JJ Allaire <jj@rstudio.com>
Cc: Dirk Eddelbuettel <edd@debian.org>

JJ,

you were indeed very close. The only remaining issue are prefix underscores. Symbol "x" is saved as "_x" in binary, so instead of __ZN3tbb8internal15runtime_warningEPKcz you have to use _ZN3tbb8internal15runtime_warningEPKcz and all will be working. nm dumps symbols as-is so with the leading underscores - the actual names are without it.

One mildly annoying issue is that dlsym() sometimes adds the underscore and sometimes doesn't so you may try "x" first and fall back to "_x" if you get NULL -- or have a configure test for that -- but the C code symbol is always without the underscore.

BTW: to maintain consistency, I'd put the extern ... R_GetCCallable(..) forest in a header file inside the tbb package.

Cheers,
Simon

From: **JJ Allaire** <jj@rstudio.com>
Date: Mon, Jan 6, 2014 at 4:53 PM
To: Simon Urbanek <simon.urbanek@r-project.org>
Cc: Dirk Eddelbuettel <edd@debian.org>

[Quoted text hidden]

Hadley Wickham <hadley@rstudio.com> Fri, Jan 17, 2014 at 10:17 PM
To: JJ Allaire <jj@rstudio.com>

Nice. Looks like you're pretty close!!
Hadley
[Quoted text hidden]

--
http://had.co.nz/

Romain Francois <romain@r-enthusiasts.com> Sat, Jan 18, 2014 at 7:03 AM
To: JJ Allaire <jj@rstudio.com>
Cc: Hadley Wickham <hadley@rstudio.com>

Pardon my french, but that looks like an awful lot of mess just to get threading. Even if you succeed at making a tab package, you'll end up with something that is really hard to maintain.

I've only briefly read tab documentation but I don't think the developer experience is great. Could you share some pointers on what you find compelling about TBB.

At the end of the day, I will use whatever you choose as the best, but I sternly believe that C++11 threads are there best tool in town.

Le 18 janv. 2014 à 04:08, JJ Allaire <jj@rstudio.com> a écrit :
[Quoted text hidden]

JJ Allaire <jj@rstudio.com> Sat, Jan 18, 2014 at 7:11 AM
To: Romain Francois <romain@r-enthusiasts.com>
Cc: Hadley Wickham <hadley@rstudio.com>

It's not just threading, it's automatic parallelization of algorithms
in a C++ friendly fashion. They also have a more sophisticated
scheduler that allocates work dynamically to least busy threads, etc.
Basically, I think they take a bunch of low level concurrency
housekeeping and make it auto-magic.

That said, I agree that if we know what we are doing and don't mind
putting in the time we can achieve approximately the same benefit with
our own threading code. Also definitely agree that maintenance looks
problematic. I've given up the ghost on this one for now :-)
[Quoted text hidden]

Romain Francois <romain@r-enthusiasts.com> Sat, Jan 18, 2014 at 9:05 AM
To: JJ Allaire <jj@rstudio.com>
Cc: Hadley Wickham <hadley@rstudio.com>

Le 18 janv. 2014 à 13:11, JJ Allaire <jj@rstudio.com> a écrit :

> It's not just threading, it's automatic parallelization of algorithms
> in a C++ friendly fashion. They also have a more sophisticated
> scheduler that allocates work dynamically to least busy threads, etc.
> Basically, I think they take a bunch of low level concurrency

> housekeeping and make it auto-magic.

The book I use as my reference for threading (C++ Concurrency in Action by Anthony Williams) essentially says that these auto magic methods (this is similar to what openmp does IIUC) are somewhat limited.

For what I anticipate about dplyr and fastread, I don't think an automatic method will do.

> That said, I agree that if we know what we are doing and don't mind
> putting in the time we can achieve approximately the same benefit with
> our own threading code. Also definitely agree that maintenance looks
> problematic. I've given up the ghost on this one for now :-)

Low level control with threads, muteness, futures, conditions etc gives us all the tools to take the most advantage of the available hardware concurrency.

There are dragons, R is notoriously not thread safe, so whenever we want to use concurrency and R we have to be extra careful about synchronizing. And thread unsafety can be hidden in the most basic functions. Take Rf_mkChar for example, if the string is already in the cache, it could be harmless, but if not then it could open the door to hard to debug race conditions on the hash map that R uses internally to cache the strings.

In dplyr, there are cases where we don't need the R interpreter at all, thanks to hybrid. This is the best case scenario and will be easy to take advantage of threading. When we do need the R interpreter to be involved, perhaps we can synchronise access to it through mutexes so that only that part os synchronized and the rest of the algorithm can evolve in parallel. We'll have to measure to be sure threading helps. But I don't think automatic methods such as openmp or tab will help. But again, I've only briefly considered them.
[Quoted text hidden]

Hadley Wickham <hadley@rstudio.com> Mon, Jan 20, 2014 at 9:31 AM
To: Romain Francois <romain@r-enthusiasts.com>
Cc: JJ Allaire <jj@rstudio.com>

>> It's not just threading, it's automatic parallelization of algorithms
>> in a C++ friendly fashion. They also have a more sophisticated
>> scheduler that allocates work dynamically to least busy threads, etc.
>> Basically, I think they take a bunch of low level concurrency
>> housekeeping and make it auto-magic.
>
> The book I use as my reference for threading (C++ Concurrency in Action by Anthony Williams) essentially says that these auto magic methods (this is similar to what openmp does IIUC) are somewhat limited.
>
> For what I anticipate about dplyr and fastread, I don't think an automatic method will do.

I'm a little surprised that you think this, because I was thinking
that most of the dplyr problems (e.g. grouped mutate and summarise)
you have a big loop over the groups, and you just want to assign
different elements of the loop to difference processors. That seems
like a good fit to OpenMP to me.

>> That said, I agree that if we know what we are doing and don't mind
>> putting in the time we can achieve approximately the same benefit with
>> our own threading code. Also definitely agree that maintenance looks
>> problematic. I've given up the ghost on this one for now :-)
>
> Low level control with threads, muteness, futures, conditions etc gives us all the tools to take the most advantage of the available hardware concurrency.

Agreed, but equally we don't want to have to rewrite all the tools we
need from scratch. It's nice to be able to draw on a well designed set
of thread-safe mutable data structures, or immutable data structures

that we can generate within each thread and then efficiently fuse
together at the end.

But it seems like like std::thread may be the way forward at the
moment. It will only work (easily) on OS X and ubuntu, but it's an
easy way to get up and running with threaded computation. Once we have
a better idea of exactly what tools we need, we can invest more time
in a solution that works well with CRAN and on (e.g.) RHEL. Does that
sound reasonable?

Hadley

--
http://had.co.nz/

Romain Francois <romain@r-enthusiasts.com> Mon, Jan 20, 2014 at 10:16 AM
To: Hadley Wickham <hadley@rstudio.com>
Cc: JJ Allaire <jj@rstudio.com>

Le 20 janv. 2014 à 15:31, Hadley Wickham <hadley@rstudio.com> a écrit :

>>> It's not just threading, it's automatic parallelization of algorithms
>>> in a C++ friendly fashion. They also have a more sophisticated
>>> scheduler that allocates work dynamically to least busy threads, etc.
>>> Basically, I think they take a bunch of low level concurrency
>>> housekeeping and make it auto-magic.
>>
>> The book I use as my reference for threading (C++ Concurrency in Action by Anthony Williams) essentially says that these auto magic methods (this is similar to what openmp does IIUC) are somewhat limited.
>>
>> For what I anticipate about dplyr and fastread, I don't think an automatic method will do.
>
> I'm a little surprised that you think this, because I was thinking
> that most of the dplyr problems (e.g. grouped mutate and summarise)
> you have a big loop over the groups, and you just want to assign
> different elements of the loop to difference processors. That seems
> like a good fit to OpenMP to me.

Perhaps. threading is new to me and I have not investigated too much openmp or TBB, …
I'm confortable with something I'm in control of, and I don't quite understand what these automatic things are doing. I have a better understanding of what goes on with regular threads from the standards.

In the background I'm learning these things, although I have focused for now on material from the book I mentioned before. I will find the time to look at what openmp does, ...

One source of pain is that e.g. there is no open mp with stock compilers on OSX, so we'd have to roll our own, or fall back to gcc from brew, ... not sure this would be available on Simon's machine, we'd have to communicate to users how to install it, etc .. IMO this matches the difficulty of requiring C++11 compiler.

But I'm keeping an open mind about this.

>>> That said, I agree that if we know what we are doing and don't mind
>>> putting in the time we can achieve approximately the same benefit with
>>> our own threading code. Also definitely agree that maintenance looks
>>> problematic. I've given up the ghost on this one for now :-)
>>
>> Low level control with threads, muteness, futures, conditions etc gives us all the tools to take the most advantage of the available hardware concurrency.
>

> Agreed, but equally we don't want to have to rewrite all the tools we
> need from scratch. It's nice to be able to draw on a well designed set
> of thread-safe mutable data structures, or immutable data structures
> that we can generate within each thread and then efficiently fuse
> together at the end.

For something like summarise, we can allocate the data before the threads are spawned, then let each thread fill its part. The premise is that each thread knows where to fill the data and that does not overlap with other threads business.

> But it seems like like std::thread may be the way forward at the
> moment. It will only work (easily) on OS X and ubuntu, but it's an
> easy way to get up and running with threaded computation. Once we have
> a better idea of exactly what tools we need, we can invest more time
> in a solution that works well with CRAN and on (e.g.) RHEL. Does that
> sound reasonable?

Fine by me. We need practice and data.

Perhaps I can try a few things using std::thread and openmp () to see:
- how they perform
- what we prefer writing

Perhaps trying different algorithms.

Beyond the obvious things in summarise, I'd be interested to see how group_by could take advantage of parallel. At its heart, it is just training a hash map. I can see how to deal with this in threads, essentially use as many maps as there are threads to avoid data contention and re-serialization. Then merge the data somehow. I'm not sure how to do this in openmp. But again, I don't know much about openmp.

Also there can be several ways to split the work between the threads. For example, in the scenario where we create two variable in summarise, we could have one thread doing one variable and another do the other. We can one column after the other, but each thread takes part of the rows, we can do a combination of the approaches.

Consider a problem where we can easily hybrid one column and not the other:

summarise(df, a = foo(b,c), d = mean(a))

if we work by splitting the rows, computing a will probably not be more efficient with threads, because we will have to synchronise access to the R interpreter anyway.

but if we allow another thread to calculate d for a chunk once a is ready, then the game is on because mean can be calculated independently from the R interpreter ...

> Hadley
>
> --
> http://had.co.nz/

Hadley Wickham <hadley@rstudio.com> Mon, Jan 20, 2014 at 10:37 AM
To: Romain Francois <romain@r-enthusiasts.com>
Cc: JJ Allaire <jj@rstudio.com>

> One source of pain is that e.g. there is no open mp with stock compilers on OSX, so we'd have to roll our own, or fall back to gcc from brew, ... not sure this would be available on Simon's machine, we'd have to communicate to users how to install it, etc .. IMO this matches the difficulty of requiring C++11 compiler.

Ok, makes sense.

>> Agreed, but equally we don't want to have to rewrite all the tools we
>> need from scratch. It's nice to be able to draw on a well designed set
>> of thread-safe mutable data structures, or immutable data structures
>> that we can generate within each thread and then efficiently fuse
>> together at the end.
>
> For something like summarise, we can allocate the data before the threads are spawned, then let each thread fill its part. The premise is that each thread knows where to fill the data and that does not overlap with other threads business.

This assumes that there are many more groups than threads, but that's
a totally reasonable assumption. There should be relatively little
locking overhead because we can preallocate the output vector and
every thread can modify the same structure, because we know that they
won't be writing in the same place.

>> But it seems like like std::thread may be the way forward at the
>> moment. It will only work (easily) on OS X and ubuntu, but it's an
>> easy way to get up and running with threaded computation. Once we have
>> a better idea of exactly what tools we need, we can invest more time
>> in a solution that works well with CRAN and on (e.g.) RHEL. Does that
>> sound reasonable?
>
> Fine by me. We need practice and data.
>
> Perhaps I can try a few things using std::thread and openmp () to see:
> - how they perform
> - what we prefer writing
>
> Perhaps trying different algorithms.
>
> Beyond the obvious things in summarise, I'd be interested to see how group_by could take advantage of parallel. At its heart, it is just training a hash map. I can see how to deal with this in threads, essentially use as many maps as there are threads to avoid data contention and re-serialization. Then merge the data somehow. I'm not sure how to do this in openmp. But again, I don't know much about openmp.

I think the two approaches are:

1. use a thread safe hashmap. All threads are writing to the same
hashmap, so it has to be thread safe, but the existing code would
change very little.

2. use an immutable hashmap. All threads write to separate hashmaps
which are joined together at the end.

My feeling is that approach 2 is what's generally favoured these days
because it's so much simpler. But maybe people have spent enough time
figuring out how to write efficient thread safe hashmaps that 1. would
be faster.

> Also there can be several ways to split the work between the threads. For example, in the scenario where we create two variable in summarise, we could have one thread doing one variable and another do the other. We can one column after the other, but each thread takes part of the rows, we can do a combination of the approaches.
>
> Consider a problem where we can easily hybrid one column and not the other:
>
> summarise(df, a = foo(b,c), d = mean(a))
>
> if we work by splitting the rows, computing a will probably not be more efficient with threads, because we will

have to synchronise access to the R interpreter anyway.
>
> but if we allow another thread to calculate d for a chunk once a is ready, then the game is on because mean can be calculated independently from the R interpreter ...

Have you read the riposte paper?
http://www.justintalbot.com/wp-content/uploads/2012/10/pact080talbot.pdf

Hadley

--
http://had.co.nz/

Hadley Wickham <hadley@rstudio.com> Mon, Jan 20, 2014 at 12:51 PM
To: Romain Francois <romain@r-enthusiasts.com>
Cc: JJ Allaire <jj@rstudio.com>

The spark paper (http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf) may also have interesting ideas. But what we want is substantially simpler since it's on one machine and doesn't need to be fault tolerant.

Hadley
[Quoted text hidden]
--
http://had.co.nz/

Romain Francois <romain@r-enthusiasts.com> Mon, Jan 20, 2014 at 1:26 PM
To: Hadley Wickham <hadley@rstudio.com>
Cc: JJ Allaire <jj@rstudio.com>

Le 20 janv. 2014 à 18:51, Hadley Wickham <hadley@rstudio.com> a écrit :

> The spark paper
> (http://www.cs.berkeley.edu/~matei/papers/2012/nsdi_spark.pdf) may
> also have interesting ideas. But what we want is substantially simpler
> since it's on one machine and doesn't need to be fault tolerant.

Thanks. I'll have a look.
[Quoted text hidden]

Romain Francois <romain@r-enthusiasts.com> Mon, Jan 20, 2014 at 1:44 PM
To: Hadley Wickham <hadley@rstudio.com>
Cc: JJ Allaire <jj@rstudio.com>

Le 20 janv. 2014 à 16:37, Hadley Wickham <hadley@rstudio.com> a écrit :

>> One source of pain is that e.g. there is no open mp with stock compilers on OSX, so we'd have to roll our own, or fall back to gcc from brew, ... not sure this would be available on Simon's machine, we'd have to communicate to users how to install it, etc .. IMO this matches the difficulty of requiring C++11 compiler.
>
> Ok, makes sense.

And Ripley is making progress yet again on C++11. (from Dirk's tweet):
http://developer.r-project.org/blosxom.cgi/R-devel/2014/01/19#n2014-01-20

Sent him an email to Ripley this morning asking question about C++11 and pointing that his claims :

> Note that the default compiler for OS X is based on GCC 4.2.1 and has no
> support for anything other than the GNU version of C++98 and GNU
> extensions (which include TR1)

are not true on OSX Mavericks.

as usual it was unanswered. It looks like the C++11 support is going to look like having

USE_CXX1X=

on the Makevars and Makevars.win files

>>> Agreed, but equally we don't want to have to rewrite all the tools we
>>> need from scratch. It's nice to be able to draw on a well designed set
>>> of thread-safe mutable data structures, or immutable data structures
>>> that we can generate within each thread and then efficiently fuse
>>> together at the end.
>>
>> For something like summarise, we can allocate the data before the threads are spawned, then let each thread fill its part. The premise is that each thread knows where to fill the data and that does not overlap with other threads business.
>
> This assumes that there are many more groups than threads, but that's
> a totally reasonable assumption. There should be relatively little
> locking overhead because we can preallocate the output vector and
> every thread can modify the same structure, because we know that they
> won't be writing in the same place.

Yes. We eventually probably will resort to use some sort of thread pool with a fixed number of threads grabbing work from a queue.

>>> But it seems like like std::thread may be the way forward at the
>>> moment. It will only work (easily) on OS X and ubuntu, but it's an
>>> easy way to get up and running with threaded computation. Once we have
>>> a better idea of exactly what tools we need, we can invest more time
>>> in a solution that works well with CRAN and on (e.g.) RHEL. Does that
>>> sound reasonable?
>>
>> Fine by me. We need practice and data.
>>
>> Perhaps I can try a few things using std::thread and openmp () to see:
>> - how they perform
>> - what we prefer writing
>>
>> Perhaps trying different algorithms.
>>
>> Beyond the obvious things in summarise, I'd be interested to see how group_by could take advantage of parallel. At its heart, it is just training a hash map. I can see how to deal with this in threads, essentially use as many maps as there are threads to avoid data contention and re-serialization. Then merge the data somehow. I'm not sure how to do this in openmp. But again, I don't know much about openmp.
>
> I think the two approaches are:
>
> 1. use a thread safe hashmap. All threads are writing to the same
> hashmap, so it has to be thread safe, but the existing code would
> change very little.

Conceptually I think a thread safe hash map would have to block a lot, so locking at too small granularity and would essentially reserialize because of that. But perhaps there are some existing thread safe hash maps

available.

> 2. use an immutable hashmap. All threads write to separate hashmaps
> which are joined together at the end.
>
> My feeling is that approach 2 is what's generally favoured these days
> because it's so much simpler. But maybe people have spent enough time
> figuring out how to write efficient thread safe hashmaps that 1. would
> be faster.

Given in that case the end game is not to create a hash map, but to use the data to create a list of integer vectors, I think I'd use 2 and try to be smart on how to create the list of vectors. But it might be worth looking around for thread safe hash maps.

>> Also there can be several ways to split the work between the threads. For example, in the scenario where we create two variable in summarise, we could have one thread doing one variable and another do the other. We can one column after the other, but each thread takes part of the rows, we can do a combination of the approaches.
>>
>> Consider a problem where we can easily hybrid one column and not the other:
>>
>> summarise(df, a = foo(b,c), d = mean(a))
>>
>> if we work by splitting the rows, computing a will probably not be more efficient with threads, because we will have to synchronise access to the R interpreter anyway.
>>
>> but if we allow another thread to calculate d for a chunk once a is ready, then the game is on because mean can be calculated independently from the R interpreter ...
>
> Have you read the riposte paper?
> http://www.justintalbot.com/wp-content/uploads/2012/10/pact080talbot.pdf

Looking at it now.