MPI-3 Survey Data

Question 1

Did you attend the MPI Forum BOF at SC09?

No	1028
Yes	32

Question 2

Which of the following best describes you?

User of MPI applications	159
MPI application developer	303
Library / middleware developer (that uses MPI)	104
MPI implementer	54
Academic educator researcher	295
Student	103
Project / program / general management	31
Other	25

Show/Hide Open Answers

administrator
Advanced user support
Application Benchmarker
Beginner
benchmarker in HPC-industry
Compiler developer
computer architect
consultant
general user support
HPC Support
HPC team lead
Industry researcher
MPI implementer (beginner)
OS
performance tools
PMPI user
Q/A engineer of one of the MPI implementations
scientific computing staff
Several of above
support
systems administrator
technical marketing
tool developer (that targets MPI)
Um was geht's eigentlich?!?!?Gibt's das auch auf deutsch?

Question 3

Rate your expertise with the MPI standard.

I am not familiar at all with the MPI standard	42
I am knowledgeable about basic MPI functionality	347
I have a good understanding of some parts of the MPI standard	492
I deeply understand most of the MPI standard	174
I am an expert on the entire MPI standard	17

Question 4

Think of an MPI application that you run frequently. What is the typical number of MPI processes per job that you run? (Select all that apply)

1-16 MPI processes	472
17-64 MPI processes	495
65-512 MPI processes	466
513-2048 MPI processes	224
2049 MPI processes or more	174
I don't know	38

Question 5

Using the same MPI application from the previous question, what is the typical number of MPI processes that you run per node?(Select all that apply)

1 MPI process	358
2-3 MPI processes	323
4-57 MPI processes	476
8-15 MPI processes	322
16 MPI processes or more	133
I don't know	55

Question 6

Using the same MPI application from the previous question, what is the typical number of MPI processes that you run per node?(Select all that apply)

32 bit	361
64 bit	886
I don't know	41

Show/Hide Open Answers

Other	10

64-bit integer
both
depends on platform
Häh?
ia64
IA-64
mixed
PPC

Question 7

I expect to be able to upgrade to an MPI-3 implementation and still be able to run my legacy MPI applications *without recompiling*.

Strongly Disagree	257
Disagree	372
Undecided	198
Agree	114
Strongly Agree	59

Show/Hide Open Answers

1/ Need to be trained with the new MPI3 standard 2/Need to access to a MPI3 library
Allowing an application to run without recompiling is too constraining. It would prevent interesting evolution.
Allowing the implementation changed in details is convenience and tended to better performance.
Although recompiling should not be necessary -- recompiling is minimally intrusive.
*A shift to a major new version of *any* library normally requires more than just a recompile.*
At first, I thought this meant recompile and run. But the next question asks about recompiling. So then I assume here we are not recompiling. On systems that don't support shared libraries, I fully expect that an old MPI2 executable would run fine after making mpi3 the default.
AUGH. AUGH AUGH AUGH. AUGH AUGH AUGH AUGH AUGH. I want an MPI implementation that works with compilers to be optimized well. Yes, that breaks the binary compatibility layer ISVs want, but it would let me write natural code rather than the bizarre contortions necessary to pack messages by hand, etc.
Avoidance to recompilation should not be an obstacle for innovation.
Backwards compatibility should not impede progress
Binary backwards compatibility hampers progress of the standard, and a recompilation is easily performed.
Binary compatibility is not necessary in my view. The old libraries can be kept for applications that are not recompiled.
Cannot see how this question makes sense. If I've a statically linked application why would I have to recompile it just because I updated MPI libraries. Unless this question is referring to mpiexec. Ambiguous.
clean it up!
Compiling doesn't hurt
Do you mean via a shared library implementation??? MPI needs to standardize the MPI header files (may not be possible at this point but at least MPI3 should have a standardized set of header files)
'Expect' is an ambiguous word in this context. I interpret it to mean 'want/need', and since I don't want or need this to be the case, I disagree with the statement. I

continue to use that definition in the following.
For me it's not an important feature, as I recompile all my application whenever I make a major change on my code
For me the answer depends on the scope of 'upgrading' (the whole computer system / OS vs. source code of a program). I'd expect a system to be able to run both MPI-2 and MPI-3 applications using different libraries, i.e. one program linking to MPI-2, the next job using MPI-3. Upgrading a system to MPI-3 capabilities should not prevent legacy binaries from running using an MPI-2 library. Otherwise the upgrade probably will not take place at all and development will not progress towards MPI-3.
Ha, that's funny.
Have no clue.
Hell, I have to recompile when switching MPI implementations most of the time. (Which (ABI interop) is something I'd love to see changed in MPI3)
High performance computing doesn't require ABI compatibilities.
Hopefully new functionality will be presented via new functions or descriptors. So the old code will run, and if I'd like to I'll be able to modify part which are forth to benefit from the new standart.
I actually don't care about this.
I am used to recompile frequently, thus, for me having to recompile is no problem. But for e.g. commercial software cannot be recompiled by the user.
I compile my program all the time
I compile statically. So I assume the library change.
I'd expect to recompile for any new MPI implementation.
I don't care to recompile, as long as I don't have to change the source code.
I don't expect to be able to upgrade any library or subsystem without requiring apps to relink or recompile.
I don't mind having to recompile even for large codes. *It would be particularly convenient not to have to recode.*
I don't mind recompiling
I don't mind recompiling.
I expect to be able to upgrade to an MPI-3 implementation
I expect to recompile whenever I change MPI implementations, regardless of any version change ...

with ditching backwards compatibility (as long as I can install MPI2 and MPI3 alongside each other).
If only recompilation is required, I do not view that as a problem.
If performance benefits make it worth, great.
If progress is desired, a price must be paid. This price should be minimized, but will not be zero.
If recompilation is necessary to support better performance, it would be ok.
If recompilation is the price for improved performance and features, why not ?
I frequently recompile for different platforms, for scalar or MPI code versions or to describe different physical problems. I don't care about recompiling,
i frequently recompile the code anyways
I have no problem recompiling my applications.
IMHO it seems not necessary to be backward compatible on such a level since it is an major upgrade.
Implementations could simply ship the MPI2 libraries, or support a runtime 'hint' that would assert the application is MPI2 compliant. Either option would allow existing applications to continue running, without constraining the MPI3 standard to be compatible with the MPI2 standard. The reality is that most implementations will continue to support MPI2 for a very long time - to support existing customers, and to take advantage of existing toolsets. The transtion from MPI2 to MPI3 will probably take at least as long as the transition from MPI1 to MPI2.
im willing to have some advantages of recompiling with mpi3
In most of my MPI using applications, the code is distributed as source and, in one, input decks are compiled at run-time into platform specific optimized binaries. As such, my users have no expectation of reusing an ancient binary on modern systems and such capabilities within an MPI-3 have no utility for me.
In my opinion, a 'new' MPI should be able to live with some 'old' MPI side-by-side on some system, e.g. in an MPI2.dll and MPI3.dll
I prefer performance and flexibility over backward compatibility.
I recompile anyway on an almost daily basis
I recompile my MPI-based applications quite frequently during developing them, so it doesn't matter to me that MPI-3 breaks the run-time compatibility.

Isn't that kind of a ridiculous constraint to put on MPI developers? I am not an MPI developer, but some features may require more information from the user environment or may factor a problem differently, it is silly to sacrifice future performance gains for the sake of saving a recompile. If people don't have the source or something, let them keep their old executables and old libraries. Or rewrite it.
it does not matter for me if I have to recompile
I think that If very usefull changes will be there, then recompiling will not difficult.
I think this is too much a restriction for MPI-3 implementors.
It is clear that one has to recompile to include new features.
it is dangerous to even think about this option, or do you want MPI to become a dinosaur?
It is not hard to type make.
It is not important to me, I can recompile them.
It is reasonable to expect application developers to recompile as new libraries become available.
Its a big problem, that most MPI implementations are not binary compatible.
It should be possible for legacy MPI programs and MPI-3 programs to coexist, but they don't need to linkt ot the same libraries
Its no problem to recompile my application, but the API should be the same so that there are no patches necessary to recompile properly.
I want a clean MPI3 without the burden of old mistakes.
I will have to link the new libraries to the application
**I* would not expect being able to run or even link against a library when the major version number has changed. This is the philosophy of version numbers using major.minor.patch*
I would not like to see any changes to the already existing APIs. Please do not make the mistake those dumb idiots of the HDF group made when they moved from HDF4 -> HDF5 and, yet again, when they moved from HDF5 version 1.6 to HDF5 version 1.8. I would vote, strongly, in favor of backward compatibility. All MPI-1.1 and MPI-2 APIs should work as is in MPI-3. *All that said, it would be ridiculous to*

*expect legacy MPI applications to run *without recompiling**
Jumping from major to major release is not a big issue, but it would be preferably to jump between differennt MPI-3 implementations without recompiling. I know that ABI's has been discussed before, and propably has been voted as not important, but for a provider of commercial software like us it would beneficial
Kann man das essen?
Major version changes normally require recompilation.
Most ISVs probably would like this, but this may make some changes difficult. So I would have no problems breaking this restriction. But do so only once.
MPI-3 should not be binary compatible with previous versions of the standard.
No need to port obsolete routines into the next generation
No objections to compile providing the source code can remain unchanged.
no problem to recompile
not applicable, I am a developer
Not realistic...
not sure why without recompiling is in quotes - is something different meant than the obvious?
not to be forced to recompile is a matter of convenience, but nothing essential at all
One usually has to recompile with every update of the MPI library on the systems anyways...
only if just the implementation has changed. usualla there'l some more changes such like with the compilers . .
Performance portability is a hopeless effort with MPI.
recomiling is fine, as long as the previous API remains supported
Recompilation is a non-issue
Recompilation is no issue at all for any of my applications, they are regulary recompiled anyway.
Recompilation should not be a problem especially if the new standard brings new features. One can always #ifdef MPI3 versus older versions for preserving portability.
recompilation would be fine with me if it's smooth.
- recompile is fine
Recompiling, even in case of very huge programs, should be acceptable if it needs to be done a single time.

each new simulation setup.
Recompiling is negligible compared to runtime.
recompiling is no problem at all.
recompiling is no problem at all, and freqently done anyway
Recompiling is not a problem.
Relinking will be required obviously.
Requiring recompile makes transition between versions difficult, but could probably live with it.
Since I use the programs on super computers like juropa with special architectures, I anyway recompile the program with respect to the given super computer; therefore it seems not to be a problem to recompile shortly the program.
Since our (academic use) code is recompiled almost every time it is run, this is not particularly relevant for us.
Source code available - recompilation is not an issue
Support for old binary execution in my opinion is not mandatory: if needed by the evolution of the standard, we should be able to change binary support. Compatibility may still be provided for old binary by means of library wrappers or virtual execution.
That's ridiculous.
There is no binary compatibility between different MPI2 implementations today, anyway.
This assumes a shared library environment. I generally don't run in an environment that supports shared libraries so it isn't an expectation at all.
this depends on a large part on the implementation and the stability/quality of the implementation
This is more complicated (more parts) than just the mpi libs, so if there were an 'it depends' option i'd vote for it. ;) This is one of the things that i'd sacrifice if the benefits were compensating.
This is what library versioning is for. If you have an MPI-2 application, link it against MPI-2 libraries. If you have an MPI-3 application, link it against MPI-3.
This of course depends on a lot of details concerning the MPI3 implementation.
This of course presupposes that the calling interfaces for the existing MPI routines stay the same.
This would severely limit the nature of the changes considered for MPI-3. Re-compilation when moving from one major version of a standard to the next is not unreasonable.

Too large of a constraint.
Unless you use DLL code, it is very difficult to change MPI without recompiling. Nevertheless, DLL deployment is a good practice :)
useful to run legacy exec for verification, I would not expect to run as performant w/o recompilation
Usually, recompilation is NOT the problem.
We already run multiple MPI implementations, seems silly to contrain future implementations with compatibility with past libraries.
we don't mind currently because we ask users to compile source codes for the parallel version.
We have wrapper library for different MPI implementation. They will surely have to be recompiled.
We recompile applications when compiler is updated for improved performance because performance fairly important in the area where MPI is used. In the same way, we recompile applications if MPI implementation is updated.
What? Install MPI-3 and not even recompile? Who DOES that???
Why upgrade then?
without recompiling an 'compatible mode' (offering NOT all new, but all OLD functionalities) would be nice ...
Would be fine if it was like that - but I think it's nearly impossible.
Wow, the time spent implementing this feature might best be used elsewhere, don't you think? Given the plethora of MPI implementations and the manners in which they have been implemented, testing this feature would be a nightmare, and ultimately failure oriented.

Question 8

I expect to be able to upgrade to an MPI-3 implementation and only need to recompile my legacy MPI applications *with no source code changes*.

Strongly Disagree	31
Disagree	76
Undecided	154

Show/Hide Open Answers

Agree	394
Strongly Agree	341

additional functionality added to the new standard should add state to the standard, not change the function of what is already there, to the extent humanly possible.
again, no changes to the source code would be convenient, but being forced to modify the sources wouldn't be a show stopper either
although this would be nice, it might limit the possibilities to accomodate new features in a user friendly way. I don't mind source code changes, but they should be e.g. straightforward regexp replacements and nothing that requires a lot of genuinely new coding
An 'upgrade paper' with concrete (!) information on what has to be changed would be brilliant. Your documentation is usually very good, but I'm not familiar with all the details and concepts, so this could potentially save me (and others) a lot of work.
Any change to the MPI API would prevent a new version from becoming widespread.
As long as it is easy to maintain MPI-2 and MPI-3 source compatibility with a minimum effort.
As long as the legacy is not using some function which may be depreciated, I do agree with this.
As long as there is a clear guide to necessary code changes, I don't mind slight modifications to the code. What is definitely a no-go is a change to the interfaces which allows old code to be compiled succesfully yet changes its functionality.
As previous. Code changes are not a problem. despite my code is 3.3million lines, the MPI-part has been isolated under separate Classes/Modules ('jacket routines') and changing that is not a problem. More actually a preferred way!
backward compatibility is the reason for some of the worst library interfaces in the history of software development :)
Backward compatibility on basic routines such as SendRecv or AllReduce should be maintained. Source changes at the level of derived type construction differences between MPI-1 and -2 would be OK
Backward compatibility should be maintained (at least in the first versions)
Backwards compatibility would be quite nice.
changes required in the source code should be only minor.
Changing source code is not a problem

Changing the code is very undesirable. I have to support multiple platforms, now I need to support multiple MPI levels? The best thing about MPI is that it's a standard, not a basket of standards.
clean it up!
compatibility assumed.
Define legacy, please. Our actively developed codes often break when switching implementations. Usually we don't have a problem running with the Cray or SGI libraries but OpenMPI and MVAPICH frequently cause us headaches. I wonder too about implementations of openib and openfabrics. That stuff is out of my realm, but some of our problems could be rooted here instead of with MPI.
Depends on the complexity of the changes, eg if I can use a script (e.g. a name changes), and of course, how extensively I use that which changes.
depends on the cost/benefit ratio
Depends on the specific features. Compatibility is expected of course.
downward compatibilty is essential!
Exceptions might be acceptable for seldom used parts of MPI-1/2
Except maybe some specific non-commonly used MPI routines
For accessing the new features, it is understandable to change the source code
For a given MPI 1.2 / 2 ABI, an upgrade to MPI-3 _must_ maintain backward binary compatibility
For most parts, I expect that I don't have to do source code changes, at least if I don't get some great benefits from it, i.e, not just because minor usability improvements.
For our application (open source scientific code), only simple changes in the source code that could be performed based on autotools detection and preprocessor macros will be acceptable. Otherwise, we would not be able to migrate to MPI-3 until it is available in all possible platforms our users might have access to.
For the most part, yes. However, peta-scale may require substantial enhancements and modifications to truly scale.
From a ScaLAPACK perspective, porting the BLACS is a pain.
General backward compatibility is important, even if for a special topic exception can be considered.
Given my previous comment, it is arguable that at least source-code

compatibility should be maintained. However, minimal changes (e.g. to use a backward compatible version of include files, or to enable a compatible behavior for MPI-3 functions) are quite acceptable.
Given the installed base, a fully backwards compatible mode should be supported to avoid alienating or at least seriously annoying users. It will help speed adoption to have this compatibility mode.
Hopefully, the majority of MPI applications will need no or minimal changes. There should be no problem to modify or improve less-often used features, if that increases useability.
I do not want to have to maintain two versions of the code for hosts that support MPI-2 or MPI-3.
I don't think it's such a big deal to break one or two APIs when releasing a major new version of a lib. When should one clean up old cruft if not at such an occasion. But the work required to port to the new version should be kept reasonable.
I expect some APIs to change, however, most legacy MPI programs should run without major source code changes
I expect source code changes to reflect new possibilities in the MPI protocol.
If changes to the API are necessary to provide a substantial increase in performance, that's OK with me.
If changes to the interface make things better with additional concurrency control recompiling/restructuring my code is fine.
If new features require architectural changes, then they should be made. Users can use MPI2 until they are ready to change.
If performance benefits make it worth, so be it.
If performance can be improved by small changes to the API (e.g. additional parameters like hints; or less parameters by API consolidation) that's fine.
If source code changes can be avoided, that would be nice. But I would not hesitate changing the source code if this brings performance/portability advantages.
If the existing C++-bindings go, that's ok.
*If there is no *stron* need, I expect backward compatibility.*
If there is the possibility to improve e.g. MPI performance on multicore systems, regardless if it would involve a major redesign. Performance should overrule

downward compability in the field of HPC. *However, the legacy interface should remain unchanged.* *(e.g. a new mpi3.h - header and an old mpi.h - header)*
I have to use MPI-1. Even MPI-2 would make difficulties without code changes.
I hope the relevant changes in my source code are as few as possible.
In my applications, 3rd party communications libraries such as MPI are only exposed to the application through thin wrapper library. This has been used many times successfully to permit different libraries, vendor specific extensions and what not to be used in my applications without source code change outside the wrapper implementation. As such, it doesn't matter much to me whether or not source code changes are required to use an MPI-3 implementation; such changes would only affect a tiny part of my code base.
I see no real reason to make old functions obsolete.
'it depends', again.
I think, that changes in source code will not be applicable for some users of MPI applications and a problems may be here.
It is not a big issue for us, but mainly use broadcast,send/receive and have hidden everything below a thin layer so it will not be a big problem if things changes
It is not optimal having to maintain several versions of the same code or to write custom MPI routine wrappers, until MPI-3 is widely deployed
It's a nice-to-have, but hardly a show stopper
it should be clear what changes there are and how to 'quickly' fix issues (maybe sub-optimally, but at least working)
It's preferable to leave existing code unchanged. Small interface changes however are acceptable, since it is still possible to run old code with an MPI-2 implementation.
*It would be nice, but it's not critical. The key issue is if one needs to *rethink* the parallelism in a legacy MPI application, not just make simple text substitutions.*
It would be nice though!
It would be nice to avoid source code changes but I'm not sure if it's that important. Source code management tools make many types of changes like this fairly easy to make.
It would be nice to see very basic functionality (the big 6 - say init, finalize, send, recv, allreduce, sendrecv) not

require code changes.
I use only basic functionality which should stay the same apart from some fringe changes (include files etc).
I would accept smaller local changes that don't affect the communication structure as a whole.
I would expect to have to make source code changes to be able to take advantage of new MPI3 capabilities.
Minimal source code changes would be acceptable.
Minor code changes are also not a problem.
Minor code modifications, or those possible to handle semi-automatically would be fine I think.
Most MPI implementation supports one specific version of MPI standard. So we have to update MPI standard if computing system operator updates MPI implementation. We don't want to modify all application source code.
MPI-3 has to be backward compatible and not change the semantics of any existing MPI calls. Mess with that and you may as well go home.
mpi3 implementations should include as well mpi2 as mpi1.1/1.2
MPI3 is MPI, not a new stuff.
MPI-3 should not be API compatible with previous versions of the standard.
MPI3 should not break any existing MPI2 API's or calling syntax. If the value of the MPI constants need to change, that will be reflected in the header files, and addressed at compile time. Extensions to existing API's are acceptable.
My code uses the mainstream MPI constructs (including MPI-IO).
No clue.
No need to port obsolete routines into the next generation
Not having to change the source is the key point of having standards. Also, the performance should at least not suffer when switching to MPI3.
Obviously development costs shold be minimal as possible. All the big companies runing their cost saving programs now, and new standarts could become below budget.
Of course, if I have to tell configure to use a different library for legacy MPI applications, that is OK
One would expect that minor source code changes are necessary for routins like MPI_Init, but not for most of the message passing subroutines.
Only new features should need source

code change.
(or at least, a little changes)
or least source code changes should be trivial and small.
Or with minor modifications
Otherwise, if there would be an automated conversion process for e.g. (but not limited to= C/C++/Fortran77/Fortran/9x, then changes to the source code could be less unattractive to the average MPI user.
Parallel computers have changed a lot since the introduction of MPI-1. If MPI is not allowed to follow these changes, it will become obsolete. *It is already common to have software in several versions installed on parallel computers. Providing both MPI-2 and MPI-3 libraries to choose from would thus be straightforward.*
Particularly important when relying on 3rd party libraries which would all need to be updated.
Perhaps some sort of backward-compatibility mechanisms can be devised to make legacy applications compile (think of a special header to be included or macro to be defined before including mpi.h) and link (think of a special MPI-2 library wrapping the MPI-3 implementation) against the MPI-2 API. In this case, the MPI-3 would have to freedom to advance in current limitations (like the 2GB entries maximum)
Probably one compile 'directive' could help to tell to MPI library what kind of MPI 'profile' (MPI version) I want to be used.
See above
See above comments.
See last question.
Seems unrealistic to have only one MPI implementation for any large cluster. Again we would run a legacy mpi for a legacy app.
should we let legacy be the driving factor of innovation?
Simple applications should run without change. The changes would have to be for greatly improved scalability/performance.
Small adjustments would be OK if it is necessary for a cleaner standard.
Smaller changes may be acceptable if sufficient benefit may be reached. Strong changes in dogma may be a problem.
Some many codes exist with minimal support that source code changes pose problems, particularly if this means a full QA-cycle is required.

only if - the MPI-3 call preserve the same MPI original names and the tasks operated by MPI-3 are formally identical to MPI
Source changes would be acceptable to me if they provided better performance in the long term, or if they produced other tangible benefits to maintainability or readability of the source code.
Source code changes are understandable if we can get enough advantages from MPI-3.
Source code changes make sense if the result is better than before.
Source code compatibility is absolutely essential for MPI applications to remain sustainable over time. If I develop a simulation in 2009, I want people to still be able to verify and test the program as-is in 2050.
Source level compatiblities would be help.
Such features are called upward compatibility?!
Surely a 'must'?
That would be nice.
That would be nice for a standard to really be backwards compatible. Although changes would probably be minor, I guess...
The MPI-3 API should be backward compatible to MPI-2 in order to allow legacy code to continue running in production. However, I welcome a smaller _alternative_ API for new development >150 methods is too much. One possibility to have both is 'mpi2.h' for the legacy API and 'mpi.h' for the new one or vice versa.
the programme should be able to run under mpi-3 as it did under mpi-2, however I would be willing to change parts of the source code to improve the parallel performance.
There should be compatibility as f90 stands to f77
There should be no changes to existing APIs that would break codes that have used those existing APIs in conformance with the existing standard. MPI-3 might _propose_ alternative APIs and _deprecate_ old ones, but changes should only be forced for good reason (e.g. the change from MPI_Address to MPI_Get_Address deprecated the old 32-bit routine in favour of the 64-bit routine, but didn't force this with a change to the MPI_Address routine for 32-bit applications which didn't need the new functionality. On the other hand, 64-bit applications would typically break if they

	didn't use the new routine, and in this case it would have been reasonable to force the change when compiling to run 64-bit [by not including the 32-bit routine in the 64-bit library]).
	*This is badly worded. I actually think old source should compile clean and work with an MPI-3 library, but I don't mind requiring source changes to access new *features* of the MPI-3 library.* *I guess I would expect any dramatic new features to be either automatic (no source code changes necessary) or optional (if necessary source code changes are not implemented, use the previous and less efficient method.) I don't mind 'paying' for better performance with a source code change.*
	This would be a quite nice feature but it shouldn't include keeping all deprecated stuff with the new standard, so I'm willing to account for source code changes as long as there is a good documentation and maybe a replacement list as a starting point.
	This would be nice ... as an advantage I expect that it would help getting people to switch to the new version - but at the other hand it might prevent some more or less 'radical' changes that might be necessary. *As a tradeoff, maybe it is possible to provide a compatibility library that translates MPI-1/2 calls to MPI-3. This way, old applications could still compile unchanged or with little changes - but probably with a performance impact.*
	This would be the easiest way for me. But I do not expect that the API will never change.
	unless the use of new available functions i would like keep my original source code
	Upgrading with no source code changes is imperative.
	without recoding I assume an 'compatible mode' (offering NOT all new, but all OLD functionalities) ...
	With time everyone get a better understanding about message passing, the MPI library developer included, so it is normal to make small changes, in MPI API or its semantic if it is for a good reason, especially for a major release.
	would be nice...

Question 9

What ONE THING would you like to see added or improved in the MPI standard?

Show/Hide Open Answers

A appropriated multi-thread safe semantics implemented as multi-thread manner for the new thread model to be used in POSIX (the one proposed for the new version of C++)
ABI compatibility on any given platform. Would greatly simplify testing, comparisons, etc.
Ability to include the parameters of interconnection network. Some basic debugging.
ability to work with other programming models such as openMP.
Active messages.
Active messages whose reception is signalled by user-registered callbacks. Callbacks should be allowed to re-enter MPI progress engine to do more communication, possibly for long periods.
add 'const' to arguments of the MPI function if the communication buffer is not modified in the MPI function
adding the following features to one-sided communication: - combining multiple transfers into a single MPI call / network transfer - strided accesses - collective communications
A decent C++ binding.
A dummy MPI module (Fortran). It is useful to be able to run teh code on a scalar workstation for testing, and this may not have MPI installed. Yes, I can use the CPP to comment out every MPI call in every source file, but it would be better if I could simply change one line in a dummy module. If I include this rather than module mpi, then I can compile the code with no further source changes to run on a scalar machine (which may not have MPI). I have since written my own, but it is very rough and ready. The dummy module provides all the MPI subroutines, but they behave exactly as if there was only one node. I can supply a better explanation and my template file if required. Email me at: a.hart@ed.ac.uk
a function to return in a Cartesian grid the rank of the neighboring processes at corners (as needed for Lattice Boltzmann applications), i.e. if a processor has choords (0,0), what is the rank of the process is at (1,1)?
A global timestamp. Please contact me for how it can be implemented (nmm1@cam.ac.uk).
All arguments to MPI calls should be declared as MPI specific entities (handles if you will) so as to enable the use of such things as eight-byte count arguments without having to use different api calls. This would help our fortran users that autopromote variables (yes that's a horrible thing to do yet most of them do).
All arguments to MPI routines are declared with a type defined in an mpi header file so that auto promoting FORTRAN or just increasing functionality by changing types (8 byte counts for example) is managed by modifying one header file.
Allow read access to send buffer between MPI_ISEND and MPI_WAIT
ALL-To-ALL management
a memcpy operation, where the source and destination format can be specified using mpi_datatypes
A more comprehensive C++ interface
An implementation of Master Last (Google for 'Minimizing Startup Costs for Performance-Critical Threading' as presented in Rome, Italy) and/or processor affinity control for tasks, perhaps a core-assignment vector or something like that, to improve performance.
a possibility to check if a node is failing and if yes to switching to another node, i.e. one could run a job on 1026 proc and have 2 backup procs on on which to switch in case one proc fails
A process waiting in MPI_recv should not consume 100% of a CPU (at least this happens in openmpi and seems difficult to circumvent).
A real C++ interface with no pointers and some (basic) support of std containers.
A simple and fast possiblity to do RMA with minimal synchronisation requirements.
a simplified one-sided communication
A standard ABI, please.
A standardized and portable mechanism for inquiring topology-related information at

application level.
A strict limit of memory consumption in each MPI call. For example, the standard should clearly specify that an in-place communication function cannot consume a memory space propotional to the size of user's buffer.
asynch. communication, thread model
Asynchronous collective calls.
asynchronous collectives
Asynchronous collectives *Letting applications deal with process crashes (with MPI returning an error message and adjusting the relevant communicators)*
Asynchronus communication being truly effective
atomic get and accumulate operations for remote memory access
At present I do not see _any_ benefit to using one-sided communications as opposed to MPI_Send/MPI_Recv. I do know of some codes that rather ambitiously decided to use MPI_Put/MPI_Get instead of MPI_Send/MPI_Recv and were surprised to learn (from me) that plain old MPI_Send/Recv works better. *I would also like the MPI-3 forum to take the lead in standardizing the functionality (APIs) of parallel I/O packages like HDF5, netCDF and CGNS. If the HDF5, netCDF and CGNS folks want to continue with their developments, then that's fine with me. But they could still, perhaps, adhere to a common set of APIs.* *One more thing: stay out of the threads model. Its a waste of time. Its unlikely that there will ever be a meeting point between MPI and OpenMP. If the MPI-3 forum is still interested in finding a via media between message passing and shared memory, they'd be better off pursuing a library based approach (as opposed to a compiler based approach). OpenMP is overly conservative w.r.t synchronization.* *Now, I know that with multicore being the latest buzzword, there is considerable interest in getting MPI to interoperate with threads in an 'efficient' manner. I am not sure this approach is the right one. The purported advantages to the thread based approach is outweighed by the problems of concurrency and ensuring that the resulting implementation is deterministic.* *Instead, I'd suggest concentrating your efforts on MPI+OpenCL and MPI+CUDA. Better interoperability here would have higher dividends.*   *To those who berate MPI to be the assembly language of parallel programming my response is: so what? After years of compiler design, we still resort to assembly level programming to get better performance! Ha, Ha!! Those who know me will recognize this comment!!!*
A Waitany() function, *which waits for an arbitrary incoming MPI communication WITHOUT giving it an array of all possible request-handles*
better c++ integration
better compatibility with Fortran
Better control of affinity and handling of multi core. *Maybe it should be possible to have a standarized* *way on how applications should run (i.e. on as few cpus as possible to use the cache, or as spread as possible to get memory troughput on numa system)*
better fault tolerance
better Fortran bindings
Better Fortran compatibility, in particular non-blocking MPI - although it is clear that the Fortran standard itself sets strong limitations for this
better handing of one-sided communications
Better implementation of one-sided communication *(on all machines I use it is unexpectedly slow and sometimes unreliable).*
Better integration with C++
Better integration with multithreading libraries and extensions like posix threads and OpenMP. We have seen huge differences in the asynchronous communication routines

between different implementations. Thread safety is not enough
better interaction with shared memory paradigms like OpenMP
Better one-sided communications, I use that a lot.
Better process creation/destruction like PVM Taking into account specialized hardware/network for all-to-all or broadcast communications
better process management -> simpler batch use within queueing systems
Better semantics
Better specification of RMA behavior (and/or more flexible options, such as preference for batch transfers or as-soon-as-possible).
Better specification of what constitutes one-sided communications. The MPI-2 standard is somewhat vague on this matter and implementors(vendors) can actually avoid providing 'true' one-sided comms.
Better standardization of toolchain (mpirun not named or behaving different in different implementations etc.)
better support for fault tolerance
Better support for fortran 95/2003
Better support for hybrid multi-processing/multi-threading (core pinning, shared cache control).
Better support for inferring language structured types into MIP types (i.e. without explicitly coding the same information twice)
Better support for one-side communication. I am using MPI_lock and MPI_unlock which mean a process waits for all ongoing communication when calling MPI_unlock. It should be possible to wait for a particular communication like with MPI_Isend and MPI_Irecv.
- better support for threads inside an MPI app - MPI_lock(), MPI_unlock(), MPI_condvar, etc - MPI_atomic_add(), etc - machine queries: - int MPI_get_info(int machine, int info_type); where info_type can be sth like NUM_CORES, CPU_SPEED, NET_SPEED, etc - one-way RPC support: - MPI_rpc_one_way(dest, function_pointer, argument_array, etc) - MPI library of standard algorithms: - distributed queue, list, etc - distributed termination alg - load-balanced hash - etc
(better) support/tools for debugging MPI applications
Binary compatibility between all MPI implementations
Binary compatibility between different MPI implementation
Binding for Java
can't think of anything off hand
C examples instead of (or inaddition to) Fortran examples
Clarification of how environment variables (should) get provided to each MPI process by the launcher
clarify MPI_Abort()/MPI_Finalize()
Clear regulation WHEN and HOW OFTEN data is sent depending on (or rather regardless of) in which order sends and receives/probes are issued.
Coexistence with Threadsystems for hybrid programming - hints passed down to the process/thread schedulers that avoid competing for resources in a hybrid application and facilitate pinning
(collective) communication routines between neighbours in virtual topologies (i.e. as proposed in www.unixer.de/publications/img/hoefler-topocolls-mpi3.pdf)
collectives
Collectives for data exchange between neighbors in a topology (say, a 3D grid)
common ABI
Communicators that can overlap
Consistent support for both 32-bit and 64-bit integers throughout C and FORTRAN.

- correct and performant working parallel IO
CUDA interface. *as a minor detail, fortran integer size..*
Currently, we're having problems cleaning up after the mpi job is finished. An official cleanup script/exe would be nice.
Debugger
debugging
debugging possibilities
Derived datatypes are extremely difficult to code and debug. Perhaps add a collection of commonly occurring predefined types -- e.g., block-cyclic array distributions.
Description of C++ support
Differentiation between node-internal processes and those on another node.
dynamic communication
dynamic creation of processes ?
Dynamic creation of processes
Dynamic MPI tasks and clean job exit when one of the MPI ranks fails.
Dynamic process management, especially the shutting down of processes.
Ease of code generation and debugging is strongly needed.
easier management of rankfiles
Easy spawning of new MPI processes from within an MPI application (e.g. as in PVM)
Easy to handle parallel IO.
effective one-sided communication
>>Efficient<< one-sided communication
Efficient one sided communication to replace pt2pt communication on Infiniband networks.
enhanced graphs
Enhanced support for running hybrid models (MPI + threads/OpenMP)
error handling
Error messages or better handling when large numbers *of messags are sent to one process.*
f2003 binding, hybrid support
failover
fault tolerance
fault tolerance
Fault tolerance
Fault tolerance
Fault tolerance.
Fault Tolerance
Fault tolerance and the ability for an MPI application to adapt to faults and continue running without have to do checkpoint/restart
fault tolerance including error detecting, process restarting, environment rebuilding and configurable checkpointing
Fault Tolerance infrastructure
Fault Tolerance in large scale, say, over one thousand nodes.
*Fault tolerance! MPI *MUST* be able to survive losing a process even if it means the # of ranks has to decrease. We cannot rely on transparent checkpointing/migration from predicted failures - we have to be able to unplug power to a node (simulate HARD failure) and survive in some capacity.*
Fault tolerance on node crash. MPI program must be alive when node crash and process should migrate to another node. It seems for me very important because many thousands of processors in modern clusters are available.
fault tolerance / resilience
Fault tolerant feature, more control in the spawned jobs (skill, status), more node control (alive, crashed, busy)
fault tollerance

Fortran90 bindings
Fortran 90 interface for all MPI routines (Iknow it's a lot of interfaces...)
Fortran assumed-shape array support
Fortran Support
*full 64-bit support (i.e. consistent use for INTEGER*8 in Fortran)* *improved one-sided communication routines*
Full bproc integration and maintenance. *Tighter opencl integration.*
Full support of one-sided comms and MPI-IO by all MPI providers.
Function description should give a more detailed idea about internals
Get rid of MPI_Cancel!
Get rid of MPI::SEEK_* (you could strip out the rest of the C++ bindings while you're at it ;-)
get rid of the 'busy wait' in MPI. Poll wait is bad, interrupt driven wait is good.
give control or hinting to the underlying protocol (example 'I will reuse this buffer later so keep memory pinned')
Global arrays
Global counter for dynamical load balance
Good Fortran 90+ API
Good thread support
hardware-independent MPI-IO
Having more complex splitting policies of datasets
Heterogeneous support
higher performance of one-sided communication
Hints for mixed multithreading/multiprocessing paralleization (I dont't know how this should look like).
homogeneous bandwidth between all processes of all communicators even if some of them were instantiated later (adaptivity)
Hooks to support transparent fault tolerance (drain messages, coordinated checkpoint indication). *Fault return codes (FT-MPI) are of less interest to me.*
How about some sort of 'error tolerance', in the sense that there is some possibility to recover from a communication problem - a feature which seems vital once we hit really large numbers of processes.
I_Collectives
I'd like the one-sided communication to work more like SHMEM's one-sided communication
I'd like to see an 'MPI Light' definition -- a minimal set of functions and a reduced semantics (re: datatypes, tags, etc.) upon which most/all of the rest of MPI can be implemented. This would be useful for running MPI on accelerators or embedded systems.
I'd like to see bindings for the Java language included. Authors/Designers of parallel Java MPI libraries like MPJ Express (http://mpj-express.org) and mpiJava may also be involved in the process.
I'd love to see a peruse-like interface being integrated to allow low-level tools to reliably separate syncronisation time and data transfer time, especially in collective communication. *I believe that this is an enabling facility for efficient performance tuning tools on tightly coupled many-core many-CPU architectures.*
I like the standard, and learn from it when I read it. Keep up with the 'Advice to...' and 'Rationale' sections.
Implementor support. *Yeah, that's not in the standard. But we still don't have access to everything in MPI-2...* *Otherwise, merging GASnet as a replacement for the current remote memory bits.*
Improved compatibility with storage interfaces (file I/O)
Improved debugging and error/exception handling. I am tired of looking at meaningless

*messages telling me 'Oh sh*t, something went wrong!' Granted concurrency and other parallel logic bugs will always remain the developer's problem but when something on the MPI library (or ib library, etc.) side of the house goes pear shaped, it would be nice to be able to figure out what it is that is acting up and how to fix it.*
Improved fault tolerance interfaces
Improved Fortran bindings
Improved Fortran interfaces and integration of up-to-date Fortran standards.
improved interaction/functionality with OpenMP.
improved non-blocking communication
improved one-sided communication
improved support for multicore processors -- although maybe this is a hardware issue more than an MPI issue
Improvement in dynamic process management
Improvements for Fortran which makes it easier to debug.
improve: MPI-IO
improve parallel I/O
improve the performance of one-sided communications
Imrpoved the support for Hybrid implementation OpenMP/MPI or Thread/MPI
include fault tolerance
In clusters made of multi-core nodes, ability to communicate processes in the same node with shared memory and processes in different nodes via sockets.
In my experience MPI I/O has to be improved and is pretty much essential. Especially with applications that run on tens of thousands of processors, there has to be a very good I/O infrastructure. So I hope to see the biggest improvement for MPI 3 in I/O!
In my opinion the MPI standard misses an interface for platform specific information which helps to tune the application behaviour. E.g. to find out interconnect information or the cluster topology.
inquire/log functions for getting more insight what the MPI calls below the surface are doing. *Eg. is RDMA used, or FIFO type messaging, buffer sizes used, number of copies performed, ...*
Integrated checkpointing! (with little or no source code changes - if possible)
Integration of multi-threading.
Inter communicator.
interface check by prototypes, e.g. modules in F95
Inter-node and intra-node threads.
Interoperability with OS (like waiting for both MPI and kernel events in select/WaitForMultipleObject)
Introduce some more utils library to MPI standard.
It should be possible to dynamically link an MPI application such that different MPI implementations can be used with the same binary. This is important in particular for OS distributions that otherwise have to define a standard MPI implementation and link all applications against this one, or provide different packages that are linked against different MPI implementations. (e.g. gromacs-openmpi.deb, gromacs-mpich.deb, ...)
it would be nice to have asynchronous collective communications within the standard
I use mixed OpenMP and Mpi but sometimes does *not seem the optimal choice. In mpi_3 it would be nice* *to define a group of mpi processes belonging to the* *same node (using the fast memory access of the single node). I do not think there is this feature in mpi right now. Next generation of processors have* *several cores in the same node and it would be useful to make a different type of communication.* *E.g. suppose that you define an mpi process* *and an mpi_subprocess (a subprocess* *is done by all* *the cores of the node of the machine), it would* *be very easy to avoid OpenMP and make a more* *efficient code I believe.*
I use MPI mostly with Fortran. The FOrtran support of MPI is still basically F77 (mapping

of all MPI types to fortran integers, etc.) It would be VERY nice if there would be a modernized interface which actually makes of F2003 (or at least, say F95) features.
I would like to be able to overlap computation and communication for global operations, e.g. all-to-all or all-reduce. This is not very important for me, but it would be nice.
I would like to have a tool, which automatically tells me which MPI routine/method is best suitable (the fastest) for the architecture I'm using it on, i.e. the architecture I will run my job on.
I would like to see a well-defined standard for being resilient through hardware failures.
I would like to see the dynamic processes (e.g. spawn) removed from the standard.
Just some thoughts: 1. C++ templated reduction operators? 2. support for multi-threading? 3. Using mpi on co-processors?
Kill the existing RMA interface and replace it with something much smaller.
Larger message sizes
Latencies!
Let MPI calculate the average of a given variable (scalar or also array, if possible) over all ranks, without the need to use MPI_Reduce together with pre-/user-defined operators.
Malleability of used resources
Maybe it already exists: something like 'ANYEXCEPTROOT' variable for targeting to avoid for loops.
More compact way to define mpi data types
More convinient file i-o
more development of non-Cartesian virtual topology management; e.g., distinguishing between the 'in' and 'out' neighbors of a given node.
more dynamic management of resources
More efficient memory usage per node (as in openmp)
More flexibility in selecting structure of array to be transmitter in Gather/Scatter routines.
more intuitive (simplified) file io
More language bindings
(more ore less) automatic handeling of non uniforme job communication - as it is e.g. the case if one does mpi inter-node and openmp intra-node
More powerfull set of instructions for building MPI applications will save time of developers and may have positive influence on performance of a MPI applications.
More robustness or stability (whatever you want to call it), mpi problems are a frequent occurrence when porting a tool to a new platform or changing to a new release (of the application, not mpi).
Moving away from MPI as a programming model and toward MPI as an execution model. Programming model and execution model are two different things. Programmers should be encouraged to write at a high level. Adding features to MPI for more elaborate control of the hardware is the kind of thing you would want in an execution model. The evolution of MPI should be to make it more suitable as a high-level compiler target which means orthogonality among concepts, clear cost models. Interestingly this could allow MPI to have more features as long as the orthogonality is respected.
mpdboot should work more reliable when using a large number of nodes. I often had to execute mpdclean and try again.
MPI-2 one-sided operations
MPI_Comm_connect/accept/join/open_port/etc not depending on the MPI process manager used and not depending on the MPI implementation used. This at the moment does not allow the use of these functions on BG/Cray XT5/etc which is really annoying. An improvement of the spawn/comm function set would be really great.
MPI_Connect/MPI_Accept to not require dodgy features that are unsupported by vendors
mpi_finialize statement is very much dependent on the system: if used it crashes on one system, while if not used another system may crash. Please define a better standard!
MPI_GATHERW/SCATTERW
MPI_IBcast
MPI-IO
mpi-io and parallel file system integration. Support for many fortran compiler in one build,

Scali MPI had this feature using weak symbols.
MPI_PUT and MPI_GET more easy to use external32 format for MPIIO asynchronous collective operation possibility to send with a MPI_SEND directly a F95 type or a struct, without define a MPI type
MPI standard should have backward compatibility, since it will save softwares without upgrade or maintenance for long time.
Multicore support
multithreading
Nicer standards for C++ bindings, especially data types for C++ objects.
No name clashing with SEEK_SET, SEEK_END, and SEEK_CUR
non blocking all-to-all communication would be very useful
NON-BLOCKING COLLECTING OPERATIONS! I REALLY MISS IT! There are really a lot of application that will benefit from it including various Math operations like FFT or Matrix factorization. There are a lot of BROADCAST's ideologically. And now they are implemented only in a blocking way. It is a pain since I can't force customers to use my MPI implementation... I'm really really looking forward to it!
Non-blocking collective communication.
non blocking collective communications
Non-blocking collective communications.
non-blocking collective operations
Non-blocking collective operations.
non-blocking collective operations (e.g., MPI_IBcast)
non-blocking collectives
Non-blocking collectives
Non-blocking collectives?
Non-blocking collectives.
Nonblocking Collectives
Non-Blocking collectives
non-blocking con
Non-blocking reduce
Non-blocking & sparse collectives
Non-blocking versions of the specialized MPI routines like MPI_BARRIER and MPI_BCAST to make it easier to implement communication time-outs (for debugging parallel hangs).
Nothing in particular
NUMA awareness
Object passing, memory window access between mpi processes
Off the top of my head: RMA capability
One-sided comms done right - the mpi2 spec for them is very limitted.
one-sided communication
one-sided communication?
One sided communication. Make it as easy as in SHMEM or UPC.
One-sided communications
One-sided must no longer suck.
one-sided operations
one-sided should be simpler to use, and the performance implications should be more simpler (given a basic understanding of the level of system support for remote memory access)
one-side (RMA) communication with an easier interface
one sides communication should have the same performance as classical mpi-1 communications

the receiver. This feature could be enabled by an environment variable, and determined only once, at the time MPI_Init is called. It could be a accessed by the receiver in the status variable, say via a call like MPI_Timestamp(status). It is intended for use by profiling tools, to measure the time between when a message is sent and when it was received. The timestamps need not require a coordinated time across ranks; it could be left to the tool to make the necessary adjustments. Note that this feature would not require a change to any library function prototype.
Optional relaxation of ordering constraints for implementations that do not require strict MPI pair-wise ordering.
Parallel File I/O
parallel i/o built in.
Parallel I/O which non computer science people can easily implement into existing MPP Fortran codes (user should not have to understand the filesystem) (no extra overhead infrastructure beyond call open, call write or read, call close)
Pattern matchin on receive statements similar to erlang
performance
Performance
please add benchmark programs to evaluate the performance of every feature/concept (e.g., 1-sided communication)
Pleas improve fault tolerance and error handling. This is necessary to run MPI Applications in Cluster environments.
Possibly, an interface with OpenMP
Predefined expectations of integration so all MPI child processes terminate if the spawning process is killed or dies in an irregular fashion.
Profiling directly included in the standard, latency, bandwidth etc
profiling / latency measurements
Programmers error determination
python binding
Querying capabilities in the MPI implementation.
RDMA onesided calls
recommondation what functions to use in code that scales to 1000 s of cpus
Recover from failing nodes and/or unexpectedly dying processes..
Reduce the number of MPI functions. (Smaller API) But its to late. Having MPI-2 as a subset of MPI-3 avoid that.
Reduce the proliferation of different functions.
Relaxed one-sided semantics in order to use one-sided to improve performance
reliability of network communication
Remove mpif.h build error in the Intel versions of MPICH mpif90
remove the one-sided thing which does not fit into MPI altogether
representation (for efficient use) of 'local' memory and accelerators/GPUs
resilience and fault tolerance
resiliency
Restrictions on passive-target one sided communication primitives should be removed. It should potentially be possible to pin any memory area.
RMA but that's addressed
scaling for hybrid parallelized OpenMP/MPI applications, such that concurrent calls to MPI from multiple-threads effciently overlap. If this requires certain restrictions, e.g. not all threads are able to take to each other, so be it.
Scaling to very high number of cores (esp on Cray XT5/6 and BG/P)
Separate subroutines for sending integers, real, double precision, etc., instead of the MPI_INTEGER, MPI_REAL, etc. Errors in the latter are not caught by the compiler.
Shared memory access operations.
Should be save to use in multithreaded programes and should be able to get an overlap computation/communication for function calls like MPI_Isend (i.e. not waiting for an MPI_Wait call to start the communication, although the receiving process already issued a receive call meanwhile).

simpler
- Simpler calling sequence in basic MPI instructions *- Simplify synchronicity between send and receive*
Simpler one-sided message passing.
Simplified one sided communicatoion
Some capacity for mapping collections of pointers between processes (e.g. the GtsSurface data structure in the GNU Triangulated Surface library)
Some limits seem unnecessary in today's computers. For instance why have the message tag only guarantee to 32k, why not use the whole integer? The 32k is impractical for any large application today.
Some mechanism for fault tolerance
Some operations which help to make 'autonomic computing' on MPI applications
sorry I don't know.
space partitioning should be able to optimize for the properties of the interconnect network.
Standard binary interface - i.e. no mpich vs. intel MPI vs. PMPI .h. My application can be compiled with 'any' mpi.h, and would have to at most relink to the mpich vs. intel vs. pmpi libraries and it will work.
standardised options for compilation and program runs
Standardization / improvement of dynamic process management (i.e. MPI_Spawn...)
Standardized and simplified launch process, especially with respect to intra-process communicating applications (MIMD)
Standardized MPI header files.
Standarized PERUSE (available only on OpenMPI, IIRC)
stay slender
Structure
support for >2GB messages (in particular for MPI IO)
Support for accelerators (GPU, FPGA) with their own memory space. Being able to send data from a GPU to CPU/GPU on a different node.
Support for Active Messages.
support for asynchronous messaging. My primary interest in MPI is using it as a transport underlying the implementation of X10 (an asynchronous PGAS language).
support for intra-node parallelization on dedicated hardware components.
Support for migration. The ability to decide the migrate a job from one node to another. Granted only part of the solution involved MPI.
Support for multiple different compilers simultaneously. *I often have the problem that different applications run only with specific compilers. This currently requires to build MPI independently for each compiler suite and select the appropriate MPI environment when starting the calculation. It would be much nicer if I could just call the different compilers from within the *same* environment by, i. e.:* *mpicc_gcc* *mpicc_intel* *mpicc_pgi* . . .
Support for querying system topology. Since we have a multiple level parallelization and some levels require more communication than others, it would be nice to be able to optimize the number of processors assigned to each level based on the intercomunication topology. *So for example, in a cluster of SMP nodes, we could set the number of processors in the lower level partition equal to the number of processors per node, so all communication in that level would be carried inside a node.* *This can be in part done using topologies, but it is not possible to adapt the partition of data to the system. A function that return some kind of distance between processes would be enough, I think.*

supporting serialization of arbitrary objects) *Support for creating user-defined reduction operations with a user-provided context.* *Some way to overcome the 2GB entries limitation.* *Nonblocking, cancellable accept() and connect() functionalities.*
support of more programming language syntaxes. for example java...
task initiation specification, clarification of dynamic process handling (ok, that is 2, but they are related)
that would be good to have a stl like operations, like vector, list and set containers
The ability of a compiler to optimise the MPI-calls (compare with PGAS models). Now. in our HPC applications all communication needs to be hand optimised and application ported for maximal efficiency.
The ability to find out which processes are on the same node and/or host. Something along the lines of the experimental topology enquiry functions in MPICH2.
The C++-Interface. Really. C++!
The concept of RMA in MPI2 have to be improved and simplified. Too many exceptions and restrictions. Consider the example of shmem, simple, performant, clear.
The messages send compressed
The possibility of having non-blocking collective operations. Ie. operations that can be called, and then later sync'ed
The robustness towrds hardware errors of the MPI standard is a deep requirement. *At a lower priority, the congestion management and time response is important.*
The topology interface
thread parallization
Three things come to mind immediately: *- Strong guarantees of deterministic behavior (in reductions for example), as opposed to the strong worded advice to implementers seen, for example, in MPI-1.1's standard.* *- Standardized behavior for the interaction of multiple threads within an MPI process with the MPI library.* *- Portable support for thread-core and memory-thread affinity.*
Tightly coupled functionalities with some kind of shared memory programming such as OpenMP
To allow improving the efficiency of communication on shared memory architectures by not forcing different MPI 'processes' to make an intermediate copy of each message in shared memory pool, the standard should relax the requirement of independence of each MPI 'process' so that is is possible for a standard confirming implementation, to allow, if the user application accept, that the MPI 'processes' be in fact implemented as quite independent thread (in addition to a private stack each thread would have its own heap allocator, but global variables would be shared). This would allow MPI 'processes' to share the same memory space on one node and copy message directly from send to receive buffer for intra-node communication (e.g. without an intermediate copy in shared memory).
To have a command allowing comparing the load of the processors (during the MPI run) without loosing of the performance. This can help to optimize the processor load dynamically.
tools for helping me to do dynamic map from process to cores
Topology discover
Topology is clumsy and confusing and usually badly implemented.
transparent access (read-only would already be nice) on buffer sizes.
true asynchronous I/O. mpi_file_iwrite does block in at least one implementation right now.
 We run a lot of Monte-Carlo applications and it would be nice to be able to add and remove nodes, especially failed ones, without crashing MPI. We can work around a node failure using data from the other nodes without having to take everything back to a

previous checkpoint.

What about processes migration from one node to another. Some kind of virtualization with moving process with all it's data for rebalancing workload on the fly.. Sorry for too crazy idea :)
Whatever is needed to enable efficient hybrid programming (MPI + OpenMP/Pthreads/PGAS/CUDA/OpenCL)
Would be nice if the next generation of MPI has nice interface.

Question 10

How much are each of the following sets of MPI functionality used in your MPI applications?

	Not used at all	Trivially used in some places	Used moderately in conjunction with other MPI functionality	Used heavily in conjunction with other MPI functionality	Comprises the backbone of my application
Point-to-point communications	27	57	159	339	214
Collective communications	19	50	190	388	151
Derived / complex datatypes	228	169	219	99	41
Communicators other than MPI_COMM_WORLD	210	160	221	127	55
Graph or Cartesian process topologies	363	139	146	62	42
Error handlers other than the default MPI_ERRORS_ARE_FATAL	466	168	80	27	11
Dynamic MPI processes (spawn, connect/accept, join)	530	107	73	30	16
One-sided communication	376	154	158	39	19
Generalized requests	474	106	83	23	7
Parallel I/O	314	107	180	129	36
"PMPI" profiling interface	440	82	118	53	31
MPI_THREAD_MULTIPLE (multiple threads simultaneously using MPI)	474	77	92	65	36
Multiple threads, but only one in MPI at a time	384	100	140	81	37

Show/Hide Open Answers

1. In theory, cartesian communicators would help but somebody already did it the hard way. I also have an unusual problem of mapping a 4D (and higher) communication problem to a 3D torus network. 2. The interface to one-sided communication interface is difficult to use. 3. My application has no threading.
1-sided, gen.req. - what for?
1. Unnecessary for our applications 2. Not gotten around using it
3) Too complicated, and most clusters either all 32 or all 64 and all big- or all little-endian, so sizeof() will do. 4,5,6) Too advanced for our purpose. 7) Not used YET, but thought of. 8-9) No idea what that is. 10) Only master does IO. 11) Used VampirTrace once, never profiled since. 12) Only node master does MPI.
Academic use. Small tests, research, etc.
actually i do just use the mpi-1 methods
All 'Not used at all' features where not necessary for my needs so far
All questions marked in this way are interesting for me, but seems to me too complicated for real implementation, with not enough expected benefits. Maybe, I should learn more about these issues?
Application is tightly coupled and generally cannot proceed without blocking on data from other processes, so the standard blocking point-to-point and collective communications suffice.
Applications concern only pure high performance computing
Applications tended to in HPC user support don't use
apps not written by myself
A simple set of features to transfer data is sufficient in my application.
A thread-safe implementation is critical to me
(at the time) lack of MPI_THREAD_MULTIPLE functioning MPI implementations
Basicaly because I don't know it.
because a very basic set of MPI functionalities is enough for the applications I currently write
Because I don't use them much or at all.
Because I have to support platforms that don't support them efficiently (or at all). It'd be nice if grequests were file descriptors, so this would play well with other software... Except the thread foo. Constraining MPI to one thread is natural to me.
Because I'm not very familiar with MPI.
Because it is sufficient to achieve the program functionality that I desire; because I havn't used some of the functionalities.
because some codes needs do be entirely rewritten and too many people are involved in. No time and/or money to do the upgrade.
because that's how they are used. Please explain how/why I should explain.
Beyond my scope of knowledge or the application state of development
By using the data types provided by MPI or the topology by default, it is enough for my applications
Cartesian: more straightforward programmed oneself Error handling: no tradition of using these, might be a good idea. Generalized requests: leads to more convoluted code Dynamic MPI processes: leads to more convoluted code, not necessarily appropriate for app. PMPI: profiling done with TAU, other tools
Collective Communcation: Because the slaves may use different strategy/application. Derived/complex datatype: We have our own way to describe data, the MPI is too restrictive and complex di use. Graph or Cartesian process topologies: We don't have application that require such process layout. One-sided communication. We don't have application using it. Multiple threads, but only one MPI: Our application run always in a Multithreaded way
Collectives : MPI_Init and MPI_Finalize Communicators: With message tags, source, destination and message content groups seem

unecessary. Topology: Not so useful in irregular nonuniform geometric distribution of data. Error handlers: Just unaware of this. Will look into as this would be extremely useful. Dynamic MPI processes: again, unaware of this should be interesting. One-sided communication: Very important IF it works. Parellel IO: still doesn't work well. MPI_THREAD_MULTIPLE: Don't Do THAT (ever).
communicators, topologies, error handlers, PMPI: not neccessary in my application dynamic processes, one-sided communication, generalized requests, parallel I/O, multiple threads simultaneously using MPI: will be used in the future
complex code and theory behind
Concerning topologies, we think it is a very good feature. We however don't use it since it is not supported by the MPI implementation we have. With our new cluster (Nehalem Myrinet), we are thinking of permuting rank id to minimize communication by hand (e.g. by using knowledge specific to our cluster). Concerning error handler. In our applications, communication is so fundamental, a communication error is like a memory error and therefore there would be no point in trying to cope with it. Concerning the one-sided communication, we think it is a good feature, however we don't use it. This is maybe more for traditional reasons but we think that in what we are doing on our hardware (we are not using Infiniband) it wouldn't bring any improvements. Concerning the generalized requests, we are not sure what it is. We use only non blocking communication and only blocking I/O. We therefore use only one type of request. The profiling interface is a good thing. Maybe we should but we do not.
Currently I'm using mpirun to distribute Posix-thread parallelized SMP jobs over different nodes in an 'embarrassingly parallel' way.
datatypes, process topologies: not necessary rest: too less knowledge or haven't thought about using it yet
Derived / complex datatypes - a lot of code and not efficient in my experience. Groups - never needed. Custom error handlers - I just check error codes, but this could be actually useful. Dynamic processes - if I want mallability I go to higher leverl parallel libraries like ibis or proactive. Generalized requsts and one-sided communication - not sure what these are. Parallel IO - never needed so far (using NFS and splitting files). mpi-thread-multiple - been told this isn't efficient, I run 4 MPI processes on a quad node.
Derived / complex datatypes: Most are floating point tensors, only few modules use compex numbers. Communicators other than MPI_COMM_WORLD: Only in few cases MPI_COMM_X/Y/ZBEAM is used. Dynamic MPI processes: The number of processes stays constant during a simulation run (no adaptive mech refinement). MPI_THREAD_MULTIPLE/Multiple threads: There is only one thread running per MPI rank (which corresponds to a dedicated CPU core).
Derived / complex datatypes: explicit buffering practically always faster. Communicators other than MPI_COMM_WORLD: very useful in rare cases. Graph or Cartesian process topologies: never used. MPI_ERRORS_ARE_FATAL is a most practical default handler. Dynamic MPI processes: most of the time, resources need to be claimed ahead of time anyway (e.g. batch queuing system). One-sided communication: would only be used if a lot simpler (Cray shmem-like). Generalized requests: don't even know what these are.

Parallel I/O: strong preference for Fortran/C I/O support. MPI_THREAD_MULTIPLE: interesting feature, but anticipated benefits did not materialize (true computation/communication overlap not faster)
Derived complex datatypes - I prefer contiguous sequences of bytes for efficiency reasons. Communicators other than MPI_COMM_WORLD - seems to have been a major source of problems for Global Arrays, with message tags, source, destination and message content it's easy enough to live comfortably without communicator groups. Graph/Cartesian process topologies - rarely - most of my design is for irregular structure or things that even if on a regular grid would suffer too much load imbalance if distributed that way. Dynamic MPI processes, haven't used them yet but that sounds interesting - I could be persuaded. Generalize Requests - not sure what's being referred to here. Parallel I/O - rarely works well if at all too much lock contention. One is frequently better off having each proc control its own writing/ or managing the collection to select writer procs oneself. MPI_THREAD_MULTIPLE -why would you want to do that? Multiple threads but only one in mpi at a time - already do that with programmer discipline (unix fork and shmem)
Derived / complex datatypes: We tried to used them for non-contiguous memory access, but implementations normally have a high overhead in memory and time for datatypes. Error handlers other than the default MPI_ERRORS_ARE_FATAL: for the moment we don't need them Dynamic MPI processes (spawn, connect/accept, join): They are not supported in the enviroment we run (suppercomputers). For the other, we might be interested in using them. But we haven't had the resources needed to implement them in our code.
Derived data types are a real bother to use. I prefer my own way of implementation. Graph and Cartesian topologies add an extra layer of complexity while I have never seen I help in performance. No need (yet) to use dynamic MPI processes, generalized requests or MPI_THREAD_MULTIPLE. MPI I/O only seldomly needed. I use better tools than PMPI
Derived data types are not required for my numerical simulations. Complex variables, for example, are handled by two double variables.
Derived datatypes are only used for parallel IO, MPI-Errors are not handled, therefore the handlers are not used at all. Dynamic MPI processes would be nice, but unfortunately the application is not ready to use it yet, nor is the common scheduling system. One-sided communication might be helpful in some places, but it is not implemented in the application. The profiling interface is currently of no use for the application, though it might be useful in conjunction with some more informations (a online evaluation library). The application is MPI only, so there are no threads involved.
* - Derived datatypes. My developers and I have seen enough bugs and inefficiencies in MPI implementations in the wild (including in fundamental point-to-point operations like blocking MPI_Send / MPI_Recv ... data corruption, incorrect handling of zero byte sends, hanging when making too many communicators, improper handling of message tags, broken heuristics for selecting all-to-all communications algorithms, broken command line parsing, unreliable/untested MPI_Init for large jobs ...) that we barely trust basic MPI-1 functionality when encountering new platforms, much less more advanced MPI features. Thus, for portability and reliability, we are forced to use the most stripped down basic*

features of MPI and then do so only with great trepidation. In short, virtually the only MPI data type used is MPI_CHAR. *- Graph or Catesian processes: See above. We MPI_Comm_dup to MPI_COMM_WORLD to create a sandbox comm that our application will use and then use MPI_Comm_split to create additional communicators for our application within this sandbox.* *- Error handlers other than the default: See above.* *- Dynamic MPI processes: Our applications do not need it.* *- One-side communications: Rarely supported efficiently in the wild (which is not surprising given state of hardware and OS support for the underlying operation on commodity clusters).* *- Generalized requested: See above.* *- Parallel I/O: See above. (In practice, we have always have to write our own parallel I/O due to deficiencies and bugs in all parallel file systems and 3rd party parallel I/O libraries we have encountered.)* *- PMPI profiling interface: See above.* *- MPI_THREAD_MULTIPLE: See above. Would love to use it. But, as far as I can tell, the MPI standard explicitly does not require MPI implementations to support this; our ability to exploit this in the wild is minimal.*
Derived data : we do our own packing *Communicators : used experimentally* *Topologies : not bothered; assuming non-blocking switches* *Dynamic mpi processes: not implemented in our code. Would be non-trivial.* *One-sided comms: no use for that but then not familiar with it.* *PMPI: might use that but not at the moment* *threads: ,,*
Developping an out-of-core library, we prefer write operations local to disks than Parallel I/Os. *For the other features, there are plans to use them, not time.*
Did not get around to do it ? ;)
Did not need them
Didn't know of their existence
Didn't need it, cause 90% of parallelization in m program is done by the FFT-routine (FFTW)
Do not see any need for or (mostly) do not know the feature.
Don't know the functionality
don't know what Generalized requests means
don't know where to make use of it
Don't need any of those complicated features and/or don't know how I can benefit from them
Don't need complex types, do N-to-N I/O.
Don't use process topologies as data is often unstructured and process count is arbitrary. *Would have used more MPI_THREAD_MULTIPLE coding but implementations didn't support this when we started the project and hence were forced to used semaphores to serialize MPI calls amongst threads. Still not complete support across all our target platforms*
Do't know
Due to the structure of the application, only one communicator is needed, the topology is simple cartesian, number of processes and their jobs are known at start, currently no profiling needed
 Dynamic MPI processes -- not helpful for my applications, interact poorly with standard queue systems. Could disappear completely and I wouldn't care.

Error handlers -- I'm a bad and lazy programmer of dissolute practices. But even if I weren't, any MPI error in my applications is generally non-recoverable. *Generalized Requests -- I like the idea, have yet to find a compelling reason to use it in my applications.*
Dynamic MPI processes - not used because this feature is hardly ever provided. *PMPI - not aware of this feature* *MPI_THREAD_MULTIPLE - feature often not provided.*
Dynamic MPI processes: We don't leave spare processors around for spawned processes, and we have no need to join other applications as with connect/accept join etc. *Gerenalized requests: current blocking collectives fulfil our blocking collective needs.* *PMPI profiling: we don't use this explicitly, as there are already many tools that use this interface and that already provide the functionality for which we would wish to use the interface.* *MPI_THREAD_MULTIPLE: as the provision of this functionality can cause extra overhead in the MPI library implementation, we don't use this for our hybrid MPI/OpenMP codes.*
Dynamic MPI threads not yet supported on our MPI implementation, if it was better supported will use it more. *In stead of PMPI we use our own profiling tools. Nevertheless PMPI is useful if there is no alternative.* *Expect that MPI_THREADS_MULTIPLE will perform worse than multiple threads in one MPI task, but have never test it.*
Dynamic processes - Don't fit well with schedulers *One-sided - MPI2 standard is close to useless* *Thread multiple - Don't trust implementations*
Dynamic processes have been unnecessary, so far, and frequently have problems with operating systems and scheduling practices. I use parallel HDF5 in preference too MPI I/O directly. I may use more one-sided communication, but I expect to move more towards UPC or OpenSHMEM.
** Dynamic processes - It's generally difficult for a library to pack the data as efficiently as an informed user.* ** Error Handlers - Until MPI implementations have better resiliency features, this isn't particularly useful. I suspect it will be much more useful once implementations are more fault-tollerant.* ** Dynamic processes - I've never found a use for this.* ** One-sided communication - In general, MPI implementations have not produced a one-sided implementation that is performant enough to make this viable.*
Dynamic processes: Not needed on dedicated cluster. One-sideds: Terrible semantics make these useless. Threading: Single-threaded codes are simpler to write; convenient to use MPI within the box.
Dynamic processes: not scalable. MPI_THREAD_MULTIPLE: too complex of a programming model
Dynamic processes: not supported *Process topologies: rarely useful* *Error handlers: applications are not fully robust* *One sided: Inefficient implementations* *Generalized requests: not sure what this is* *THREAD_MULTIPLE: lower performance*
dynamic processes would be nice to use, but have been discarded due to inhomogeneous bandwidth between different communicators / processes spawned later.
either don't know what it means or I don't use them, eg Parallel I/O - I have my own :-)
Either I don't know what it means, or it isn't necessary for my application (high performance computing on multi-core systems).
Either I have no need to use these features or I don't understand them well enough to use them.
Either my application does not demand these features or I do not know how to use them or my hardware does not support them (dynamic MPI processes).
Either no need or unawareness of that specific feature
Either too complex to use, or too bad performance-wise

either too complicated to be used or don't reflect applications needs
Either too hard to use or not applicable to our program
Encapsuled MPI communication with always the same scheme.
Error handlers and process topologies: not much value without better vendor support (such as better process mapping)
error handling: would be nice to implement it but takes a bit of time. parallel I/O: I need my processes to write single unique files and for my stuff I don't see the point in parallel I/O
Error: No way to go on. Most Applications can't just reconnect to the other process in case of an error. dynamic: conflicts with batchsystem resource allocation. Generalized: No need PMPI: Should not be part of MPI. Multiple threads: All threads are mostly symmetric to each other.
For Dynamic MPI processes, because it is not implemented by IBM. For Graph or Cartesian process topologies, Error handlers other than the default and 'PMPI' profiling interface, because I do not need it
For most, not needed by the application. For MPI_THREAD_MULTIPLE, something to be investigated further.
For one-sided, we use SHMEM or Co-Array Fortran instead of MPI. Derived datatypes are not well-supported across all platforms of interest. For performance portability, we have avoided its use. We have not had a need for dynamic MPI processes, generalized requests, or MPI_THREAD_MULTIPLE (yet).
For self developed applications, I try to use only the simple communication functions to simplify the MPI part.
From my user point of view: simply because it is not implemented.
functionality is not needed for the target application (DFT code)
- Functionality not needed - No time yet to use it (e.g. one-sided-comm, parallel IO, profiling)
Functionality not required.
functionality unknown to me
Generalised requests- Don't know enough about them to use them with confidence. MPI_THREAD_MULTIPLE- Previous experience shows that most of the codes perform no better using a single MPI process per core.
Generally because not required. We did try single-sided communication at one time, but found it was not portable, (some MPIs didn't have it at the time) so moved back to point-to-point. So we tend to avoid 'cutting-edge' features.
Generally only use functionality when it's appropriate and use less common functionality when it's necessary
good file system support lacking for mpi i/o, only recent additions of one-sided communications, and too restrictive interpretation of dynamic process
Graph or Cartesian process topologies: For current scaling not important; Error handlers: Errors are mostly fatal - and data to recover is written per-iteration numbered file; DynamicMPI: Not used so far. (Or I missed it, I have only contributed to a small part of the program)
Graph or Cartesian process topologies not applicable to the program; MPI_THREAD_MULTIPLE is uncertain to work properly on all different available machines (right now)
Graph or Cartesian topologies offer no advantages to this application - MPI_COMM_SPLIT plus application-managed point-to-point are sufficient. Support for dynamic MPI processes to date has not been widespread enough to make applications depend on it; this has prevented some otherwise promising avenues of experimentation. MPI one-sided communication is not used by this application (although it can be compiled under SHMEM), because it is not thought likely to offer any performance benefits, and the interface is more awkward than SHMEM's. Multiple threads have not been used in this

application to date, partly because the anticipated effort of trying out hybrid OpenMP/MPI is estimated to be quite large, for little, or no, reward.
*hard question. how to explain why I *don't* use something.*
Has not been needed / useful yet.
Haven't had time yet to learn about these features or to check whether they would benefit my applications.
Haven't needed these features much yet.
I am just a beginner in this topic.
I am not an expert in MPI and do not want to be: I aim to use the most standard basic functions which will, I assume, be those which are most reliable and most highly optimized for the cluster geometry.
I am not familial with Cartesian toplogy, I only have experiences with small size nodes with direct connections.
I am not familiar with those.
I am unfamiliar with this.
I am usually just applying existing codes (i. e. hydrodynamics, radiation transport), and it is just my impression that these features have not beeen used, although I am not an expert on this.
I believe that keeping things simple gives you the best performance. The scientific problem I solve only requires a reduced set of MPI functionality. However, I have also run code that uses other communicators than MPI_COMM_WORLD and derived/complex data types. These codes only scale to about 100 CPUs. My code scales to more than two magnitudes higher numbers of processors.
I cannot answer these questions, the model is set up by a collegue and I just run it.
I debug with write statements *I don't understand dynamic processes* *--------- need one-sided communication or the other things* *I would like to do parallel I/O but don't do so yet*
I define complex data types within my application and use only mpi send, receive, broadcast and their derivatives. The applications has a non-variable processor space once started.
I did not needed it
I did not need those features.
I didn't need any further error handles so far. *one-sided communications are only needed during initialisation process.* *dynamic MPI processes are not needed. the programme is designed to run with a fixed number of processes.*
I do not have the need of any of these functionality in my application. For the last point, I had to develop a MPI_THREAD_SINGLE version of the application to use any MPI implementation.
I do not know most of the cammands.
I do not know this functionality. I will check if it solves my problem.
I do not need it
I do not need some of the above functions. Regarding derived/complex datatypes: I find it easier to communicate them 'by hand' than using the MPI function.
I don't expect a strongly improved scaling of the problem using those stuff. Some of them are also unknown for me.
I don't know
I don't know and don't need all MPI functionality.
I don't know most of the features I markes as 'Not used at all' and neither do my colleagues, the code we are using is quite old and changes are done mainly when they become immediately necessary.
I don't know of any place where they are used.
I don't need it; my favorite app (time explicit finite differencing on a grid) is boring, and I don't use fancy stuff to implement it.
I don't need them
I don't need them. *Or, rather, I don't understand them well enough/at all to know that they could be used to*

do what I am trying to do in a much neater way.
I don't need this fuctionality at this time, but I am very glad that it is implemented and I may well use it in the future
I don't see how they help in my situation
I don't use one-sided communication because I don't find MPI-2's support rich enough to warrant it; so I build my own one-sided communication routines using two-sided communication. *For the other things, I have simply never needed/investigated them sufficiently to use them.*
I found one sided communications were so restrictive as it was more straightforward to redesign the algorithms. *Spawning also didn't really suite dedicated HPC platforms where resources are request from outset. Hard to see how this could have a future.*
I found the learning curve too steep, and my problems were solvable without resorting to those functionalities.
I got no experience in MPI and just reused another developers code, so I only use the MPI functions I'm familiar with.
I had no need for this functionality.
I had not enough time to learn how to use some of the more sophisticated MPI concepts.
I have already had a working solution using MPI and OpenMP, and I am reluctant in changing dramatically the running system.
I have began to use MPI only a short time.
I have coarse grain parallelized program so its not necessary to use topologies or complex data types and communication pattern. By the way as a 'beginner' in MPI programming its relatively hard to learn such functionality.
I have not yet investigated process topologies, I think it isn't so useful on the clusters I use. Error handling is not well specified in the MPI standard, it isn't clear that error handlers are useful (but if they were specified, it could be extremely useful for robust applications). My application doesn't use dynamic processes or one-sided communication, although they look very useful for some tasks. Also when I developed this application, these facilities were not reliably available. The advantages of MPI parallel I/O, versus handling I/O at each process, is not clear to me. I use MPI_THREAD_MULTIPLE, except in (fortunately nowdays) rare cases when this is not available. Multiple threads but only one in MPI at a time is a terrible programming model!
I haven't explored yet that functionality
I'm a student :-) My projects aren't really large-scale.
I'm just speculating about what could be needed for my application if it were based on MPI
I'm not very familiar with these features.
In all cases for Not used at all is because there is no apparent need to use them. In some cases, like complex datatypes or dynamic processes, that this reduces the portability of my code (complex datatypes are often slower on some machines than others while dynamic processes would not be supported on most machines I run on.)
I need not them for my implementations and applications.
i) no considerable speedup for my applications *ii) if necessary, I would rather use this features indirectly via the use of libraries, e.g. using the Global Arrays Toolkit instead of dealing with the MPI one-sided communication function calls myself.*
In part lacking knowledge (error handlers, gen requests, threads), or features not necessary (derived datatypes, communicators, one-sided comm, dynamic processes).
Intermittant errors in most applications are too hard to handle programmatically, so it's usually not worthwhile to use Error handlers. Most errors indicate either programming errors or other hard errors, that cannot be fixed automatically. *I woud love to use multiple threads, but there is not a single MPI-2 implementation out there, that scales well in this mode hence I am stuck with a single thread communicating at a time.*
Internally the library uses point-to-point communication only. Other features are rarely required. However, users are able to configure the library using some of these features.
I/O handled by hand *communications are wrapped in higher level C++ communication classes that handle*

buffers, complex types etc...
I only created some introductionary exercises for the students, those had to be kept simple.
I primarily write IO libraries and support IO related activities. These features are not ones that I have had a need to use based on my communication needs.
Is not needed or appicable in my applications.
I still stayed in MPI1 yet.
I think the functionalities are important, and I expect to use them at some point, but I've been able to do everything I need to do with the other MPI functionalities.
It is not that I will never used that, is just that I haven't get to he point where I need those features.
it's not necessary
It was not required.
I use cartesian topologies but my MPI does not make use of it. I don't need other error handlers. Dynamic MPI processes, one-sided communication and MPI_THREAD_MULTIPLE is only lousy supported (if at all) by many MPI implementations but could be useful for me. Generalized requests would be helpful but are slightly broken in the standard.
I use MPI as basis for a runtime system of another middleware, and these functionalities just aren't needed there.
I've not the need of specific error handlers and dynamic MPI processes.
I will perhaps use parallel I/O in future versions of my application Other sets: no need (e.g. instead of using 'PMPI' I use tools like scalasca)
I work on a middleware that couple multiple MPI applications, offering in a higher-level of abstraction an Hierarchical SPMD-like programming model. The features markes as 'Not used at all' are features that can be used by users, but within each independent MPI applications, and 'Trivially used' are those that we provide bindings but don't have few or any extra supporting code.
I would like to use more threading with interleaved MPI but ...
I would like to use other error handlers, but I don't have confidence in the error handling of current implementations.
lack of time for improving my programs and use all te potential
legacy code did not make use of it
Legacy code, much of the implementation done using the MPI-1 standard and very early (and not complete) MPI-2 features. No attempt has been made at using the full potential of MPI-2
legacy, complexity or not necessary
mainly because it is not needed for my application, which is lattice QCD multiple threads are not yet used, but are to be used in close future.
master-slave construct, master does all I/O, slaves run independent of each other, communication only between master and slaves
Monte Carlo calculation: Normally copy input data to every node, every node does the same job with different random numbers, and at the end the data is summed on one node
More than one communicator adds complexity. MPI2 functions like spawn has not yet been included. Generalized requests same as above. Profiling rely on Scali MPI built in tools. Using mpi from more than the master thread is complex and have so far been avoided as it is perceived as unsafe (I know it is safe in most mpis).
Most applications are large scientific codes of legacy type.
Most features are not needed because just a fixed cubic data structure is distributed to the MPI nodes.
Mostly because the huge code needs adapting and there is no time. :o) Parallel I/O for example is a great idea, just not implemented yet. Collective communications are actually avoided on purpose for the obvious idle-reason.
Mostly only basic MPI subset of functions used
most non used at all features are unnecessary in my context, exept : - MPI_THREAD_MULTIPLE: would be convenient but is not well supported by many implementations.
Most not needed or not supported in implementations. MPI_THREAD_MULTIPLE has been badly implemented.

the MPI infrastructure. MPI I/O is just starting to be used in some codes (e.g Fluent and one user code).
Most of the MPI functionality was not needed in order to get a good performance. It would just make things more complicated without an obvious reason.
Most of the 'trivial' items above either don't fit in our job launch model or don't provide important functionality for our uses.
MPI-2
MPI_COMM_WORLD: I don't need another one *Dynamic MPI processes: I'd like to use it but I don'T know how to use it.* *Parallel I/O: One of the next things I will implement*
MPI_COMM_WORLD sufficient *Dyn processes non necessary* *No profiling performed*
MPI support is not fully implemented yet. Will investigate some of the above features to improve performance and robustness in the future.
MPI_THREAD_MULTIPLE is not used in our system because its implementation is not solid enough, not because it is not required.
MPI_THREAD_MULTIPLE : unfortunately, not available in my MPI implementation (Please, make it mandatory to support it!). Same for much of the others (PMPI, parallel I/O, ecc). Make features mandatory, so library vendors all support them!
MPI_THREAD_MULTIPLE: waiting for a stable open source MPI implementation. *Complex data types: in my middleware, I choose to not expose this feature* *Other error handlers: I intend to use this to implement a a fault-tolerant version of my middleware.* *PMPI: intend to use in future* *Rest: conflict with my middleware programming model.*
Much legacy code which has been ported without much expertise
My applications are SIMD or moderatley MIMD type so I don't need process spawning (however, this will probably change). *In my applications either every process writes its small output to own file or only the master process does so. Therefore, I didn't need MPI I/O so far.* *I use TAU for profiling. Didn't try PMPI.* *Never needed use non-default error handlers.*
My applications don't have a need for most of the features that are marked unused. There are a couple of noteworthy exceptions: Dynamic MPI processes are of interest to me, but the last time I looked at them they were not usable on very many MPI implementations running on machines we use. I _should_ be making use of communicators other than MPI_COMM_WORLD, but had encountered performance bugs in the distant past and haven't taken the time to retry this again with current implementations. I haven't yet tried MPI_THREAD_MULTIPLE, but this may be end up being of interest to me in some cases going forward.
my code does not need them for the time being, but I may consider such as 'Graph or Cartesian process topologies', 'Dynamic MPI processes', 'Parallel I/O', 'Multiple threads' later.
my code is not currently threaded, but I do plan to use MPI_THREAD_MULTIPLE if possible when it is
My programs do not require them.
My research is not focused on multithreading. *I am not used to PMPI.* *I have not used dynamic process management yet.* *All these features, I hope to use in the future.*
Never had a case that I could use it. Also, as a consultant helping to develop user's MPI code, I need to keep things as simple as possible.
Never had the need

No complex datatypes because too complicated in Fortran to be useful for my application. Cartesian topologies: not well suited for problem. Error handlers: I should use them more, but other developments have precedence due to deadline constraints. Dynamic MPI processes: My programming model is MPI on top of OpenMP threads, this is not well suited for the way I do this. One sided comm.: I couldn't find a benefit so far in my app. Generalized requests: Haven't looked into it. PMPI: I use third party profiling software. Mult threads, one MPI at a time: Not useful for my app
no experience / skill
No MPI IO because of parallel file system and external (non MPI) libraries. This allows the use of non parallelized applications working on the resulting data platform independently without the need of MPI Libraries. --> Post processing
no need
No need and no gain.
No needed in my applications
No need for certain features. Computation time vs. development time.
no need for the application
No need for them
No need from the application.
No need/no sufficient knowledge of MPI
no need, no time to implement yet
no need or benefit is still unknown
No need to accomplish the aim
No real need demonstrated for those features.
No reason to use is most of our applications, though I don't have access to most of our users applications, so they may be used more.
Not all implementations provide all features, not all are efficiently implemented, and some don't fit to the application.
Not available on systems I use, or not needed by the application. But note that I am a tool developer, so these are applications I use for testing, not ones that I develop myself.
not enough performance gain on my system or feature not needed or no performance info available
Not essential for the applications.
not familiar or my app is not thread-safe
Not familiar with most of them;parallel I/O not needed as my application does not perform heavy I/O; threading not implemented in my application; use TAU for profiling
Not familiar with the Graph or Cartesian process terminology Dynamic MPI processes were no working well when tried but the desire to have them is there.
Not familiar with them. Only a few functionality is actually needed.
Not familiar with these
not necessary
Not necessary
Not necessary, additional complications, implementation problems
not necessary for my application
not necessary, some not known
not necessary to use this option
not needed
Not needed
Not needed at this point: -- Dynamic MPI processes -- PMPI -- MPI_THREAD_MULTIPLE

Must ensure majority of MPI implementation support it: *- One-sided communication*
Not needed, basic features/calls meet my needs.
Not needed by the application. *There is no need to try to use all* *eleventeen thousand MPI-2 functions.*
Not needed for my applications
not needed in my application
not needed in my application.. but usable sets..
Not needed in my applications
not needed in the code
not needed / not yet tested
Not needed or too damn complicated or not performance portable.
not needed up to now
Not needed up to now (will probably change soon, at least for parallel I/O). Special features often have bugs in implementation (even MPI_Probe is not robust in most implementations), thus one never can trust really special features.
not required by the respective implementations
Not required or not suitable for my application.
Not supported by the currently developed library. This might change in the future.
- Not supported by the System *- derived types not performing better than copying itself* *-performance of MPI_THREAD_MULTIPLE not there yet*
'Not used at all', because problem does not need this kind of communication
Not used at all: These features are currently not needed in our application
not used to the functionality of these procedures
Not used, when teaching only a subset of the standard is chosen. This thus reflects my (students') prevalent use of MPI.
not useful for what I am doing or too complex to impliment
not worth thinking about hybrid mode
Not (yet) necessary for our application (highly parallel CFD using DG discretisation)
No used at all because this funtionality are disabled in many computer that i use.
One-sided and multi-threading capability would be used more frequently if their performance were better. (As for multi-threading, I just says MPI/OpenMP hybrid implemantation is not so efficient that I dare to do making my code heavily complicated.) *As for other functionalities, simply I don't need them so far.*
One sided barely used because they suck - would like to use more
One sided comms are not fully support by all MPI implementations and thus fully portable code can't be generated using them. *Not familiar with dynamic MPI processors so haven't used them.*
one-sided comm: the concepts/api are not the ones I'd prefer *dynamic processes: not sure about the usefulness in an environment where jobs are submitted to a batch system where the job gets a fixed number of cpus*
One-sided communication is badly design and cannot be implemented well. *Multithreading in MPI is not implemented well and most of the time turned off in production builds.* *I don't use profiling tools so I don't use PMPI that much.*
One sided communications are a pain to use in MPI - so I don't.
One-sided communications are used heavily, but MPI-2 one-sided sematics are not sufficient so other libraries are used for the one-sided communications (ARMCI, LAPI, etc).
one-sided: difficult to use vs,. SHMEM/UPC *derived: too much hassle and benefits unclear* *Dynamic spawn: not supported on many machines I have run -- or messes up parallel scheduler*

Generalized: hmmm.. don't know what they are.
One-sided I/O potentially useful but not supported on current HPC comms hardware. (Eg cray) Same with dynamic processes. Potentially very useful but not usable in any current HPC batch environment.
One sided routines perform poorly. Creating dynamic MPI processes does not fit into the way people use our cluster. Hybrid programming doesn't help performance.
Only check that implementation standard-confirm. We have no many requests from users to fix bugs
Our applications heavily relies on MPI_ISEND and MPI_IRECV calls, all is asynchronous and we manage a cyclic buffer for send buffers. We are used to work with subcommunicators but not groups. We may be interested in the use of MPI_THREAD_MULTIPLE in the future. As the number of MPI processes is fixed by the user of our library, we do not spawn new processes. One-sided communications could be interesting for a process to know the state of other processes (current memory usage, amount of work ready to be done) but at the moment we still use MPI_ISEND / MPI_IRECV for that purpose too, with a dedicated communicator. I am not familiar with generalized requests.
Our codes from a computer science point of view are simple: rectangular domains, fftw, and I/O operations from time to time. Nevertheless the codes are eager resource consuming. We just need to compute massively in each core and do transposition where communications take place (over 30-50% of the time code) (and we use indeed a MPI_send_receive... working better than the MPI_alltoall, at least in Mare Nostrum). So we just use few MPI calls.
Our data topology is simple enough to keep messages trivial. At the same time we need to transpose all our data between nodes, this means that we need to communicate between nodes (not cores) to keep message sizes above the latency threshold.
Parallel I/O - not portable 'PMPI' profiling interface - done with timings MPI_THREAD_MULTIPLE - either MPI or OpenMP
parallel IO performance is horribly bad. don't really know what to do with the other sorts of things, or simply don't need them.
Performance is the reason we use MPI. Eg derived types can add clarity, ease programming requirements, but if doesn't provide a performance benefit, in a portable manner, we don't use it.
point-to-point: for developping/debugging only parallel I/O: hardly needed rest (= 'not used at all'): too less knowledge
Poor support in vendor MPI
porting costs
probably, because U use an old application based on MPI 1
Process topologies have not really caught on at all; to be useful, they'd need to be dynaimcally adaptable and extend to hybrid applications. Since there is no consensus across implementations on which errors are recoverable, most apps assume that an MPI error kills the whole application. Generalized requests are a solution in search of a problem.
Regarding Cartesian and graph topologies, my end-user applications are related to solving PDE's on unstructured grids. The Cartesian topologies have no application; the graph topologies seems to have little use beyond storing neighbor info and it is unclear that MPI implementations take any advantage of them. Regarding RMA features, they have too complex semantics, and benefits are unclear on distributed memory architectures (which are the main target of my applications). Regarding generalized requests, the lack of MPI-provided mechanisms to make progress are a major drawback. I do have thread support in my everyday working platforms, but that is not enough to motivate the usage of Grequest's in general scenarios.

Regarding parallel I/O, it has not been a strong requirement for my applications, though I'm not actually using them because of my laziness to update the scarce places where C stdlib I/O is performed.
Regular data structures can be used directly in MPI routines. *I/O through other API layers.*
Same answer in all cases: some of the functionality of MPI is not (yed needed) in my code. Communicators other than MPI_COMM_WORLD, Cartesian topologies and parallel I/O will be used in the code within the next 1-2 years, however.
sending datatypes is a little bit hard to do it *error handlers i used them when i had made a fault tolerance application* *Generalized requests - first time i heard about them* *'PMPI' profiling interface - i used other profilers*
simply not required
So far, they were not needed, partly because there were simple equivalent MPI-1 constructions. *However, one or the other of these features may be used in the future, in particular multiple threads, when entering the era of hybrid programming.*
so far we had no need for such functionality
Some are not needed in my application, others are not known enough to see them as helpful
Some features not known, some not needed
Some of the features appear interesting for large applications designed by large professional teams, though certainly things like MPI-I/O and hybrid OpenMP/CUDA/OpenCL/etc. stuff is becoming more interesting for smaller apps as well. One-sided comms might be useful when integrated into the language (e.g. PGAS), but in MPI itself it seems useless except maybe for implementing PGAS runtimes.
some of the functionality is not needed for the code, others is 'too new (i.e. not in MPI 1.1)', others we would like to adopt, but have had no time to implement yet (e.g. parallel IO)
Something I do not know, something is not useful for me.
spawn is difficult on micro-kernel machines that are batch based *derived datatypes are too non-performing* *most mpi implementations don't handle other errors well* *one-sided operations are too cumbersome in their current format* *other I/O librarires are better* *most mpi implementations are not portable with MPI_THREAD_MULTIPLE*
still using MPI 1
Support for MPI I/O seems spotty, we don't use it. I wasn't aware MPI was thread safe, so we don't use multiples. I never saw the win with one-sided communications since you can't be sure something has happened til you check (which is two sided). The graph or cartesian process topologies I usually stick in my code not inside MPI. *The rest I can only say that I don't use them in my MPI code, note that none of our clusters use them.*
teaching
that's just my finding in the benchmark codes I have seen so far.
The bulk of the communication used in my codes is collective exchanging floating point arrays. I have used one-sided communication, but the performance was worse than a self-implemented version of it based on standard point-to-point and collective communication. I think PMPI is used by scalasca, which I use to profile my codes.
The Code is a CFD code, MPI is only for exchange of boundary values (ghost cells) used. It's a very simple implementation. Actually we are combining OpenMP and MPI.
The code is a particle code that parallelised embarisngly well. As much as possible of the communication is done in postprocessing. On most of the runs we could work without MPI at all.
The code I work with is quite old and was developed when most of these functionalities were not really reliable - and now it's too time-consuming to change this.
The code uses subdomain decomposition parallelism with a cartesian topology, which requires only a subset of MPI functionality (covered by MPI-1 already)
 The most important of the 'not used at all' are: *1) Dynamic MPI processes*

2) One-sided communication #1 - Is just not implemented broadly enough. #2 - One-sided messaging in MPI2 was very poorly designed and doomed from the start. This entire area needs to be reworked.
The need did not arise
The primitives were not necessary in the application
The problem is really trivially parallel.
The problems that I am concerned with are not well parallelizable and scale well only on a limited number of processes. The key features of MPI mostly are sufficient.
There is simply no need for it.
There's no error handling in our code. None at all.
There was no need to use these features.
These are not need by my application, which can be implemented using a very small subset of the MPI standard. Dynamic processes do not have good support in my batch system.
These calls are not needed.
These features are not known to the developers well enough and not implemented.
the usage was not necessary
This roughly corresponds to the usages in the various MPI test suites that are available, which is the primary 'application' that I run.
Those were simply not needed in my application.
Threading is evil.
- Thread performance for current MPI implementations is really bad as far as I can tell, especially when trying to overlap communication with communication over a different network segment. - Parallel I/O is trivial to implement without MPI in our application. - Dynamic MPI processes provide only limited flexibility and are not supported by many production infrastructures (they tend to use fixed-size reservation mechanisms).
Threads: I do not see the advantage of hybrid MPI/OpenMP programming. It buys a one-time performance advantage at the cost of mixing application code with machine architecture details. I'd rather write architecture-agnostic applications and require MPI (automatically or with hints from the user at start-up) to optimize the communication depending on the process mapping.
to miminaze total computing time.
Too complex, to much deadlocks possibilities
too complicated: one-sided comm., parallel I/O currently not required: topologies, error handlers, dynamic processes, generalized requests
Topologies are archaic and a poor match to modern hardware. Dynamic processes are not much used in my kind of HPC. One-sided communication is not as useful as it appears. I have never needed generalized requests, nor parallel I/O, but can see uses. And most current threading specifications (e.g. POSIX) are a reliability and performance disaster area.
Topologies: I have no experience. Error handlers: If there are errors, then the application will need to abort anyway. dynamic processes: Batch queue handlers do not allow dynamic allocation. One-sided communication: Was not available everywhere when our application was developed. I plan to use this (or some other kind of RDMI) in the future. Parallel I/O: Planning to use this, likely via an existing library such as pHDF5 or ADIOA PMPI: We use external tools that use this ABI, but we don't have PMPI calls in our application. multiple threads: I think this is not available in MPI implementations on the HPC systems we are using.
Tried some, but didn't find a performance benefit, and sometimes found a degradation (eg derived types). With performance, this functionality would be valuable.
Typically don't like this functionality so don't use it when I don't have to.
typically, most of our testing is done with traditional non mpi2 constructs.
unknown features for me

implementations.
use of MPI-1 standard only, only arrays (but no data structures) are send by MPI
Use other MPI profilers
using MPI v1.2 & fortran77
- usually build my own simple packet structured on top of MPI so I only send/recv MPI_BYTE and do casts *- the infiniband communication layer is reliable (no err detection required)* *- MPI_COMM_WORLD suffices: no need for virtual topologies, usually impossible to make use of them* *- my stuff usually uses 1 MPI process per machine and use multi-threading then internally*
very coarse grained parallelization: nodes compute independently for some hours, then exchange about 5 GB data via MPI_REDUCE/MPI_BCAST, then compute again for some hours, etc.
Was bedeuten denn die ganzen Abkürzungen?
We are still at the development stage of the physical model we would like to use. So, no sophisticated MPI function is considered.
We are using an older code where this functionality is not present. We wrote our own posix shared memory code to do one-sided communication. As we get newer machines with many cores, we might yank out our code and use mpi.
We are using MPI as a portable abstraction layer above infiniband etc so higher-level functionality is not needed. Error handlers and dynamic processes we would like to use in future.
We don't do much I/O. Also, on the HPC platforms we use (BlueGene most notably) threads are an issue. Further, in parallel linear algebra there is not much need for very complex datatypes.
We don't use MPI for error handling at all. *I don't even know the dynamic process interface.* *We don't use threading -- multiple processes on one CPU are treated like processes on different CPUs (just that behind the scene they communicate faster)*
We have had not had the need, because the execution model is very simple :)
We have hybrid OpenMP/MPI codes. For MPI-IO *we partly had to rewrite the functionality, because the implementations available for MPI-IO are damn bad.*
We have not come to the point of incorporating those features (such as parallel I/O) yet.
Well, they are not needed. Still happy with the basics.
We mainly develop an instrumentation library that intercepts the MPI calls. Although we have developed simple MPI applications. These applications do not need specific error handlers, One-sided communications, generalized requests because we found clearer solutions using the rest of the MPI calls. *Regarding the MPI_THREAD_MULTIPLE, our applications do not call MPI where threads are spawn, so there's no need to.* *Finally, the Dynamic MPI processes is not supported by our environment (batch system using Slurm/Moab)*
We never saw the need for a communicator other than MPI_COMM_WORLD.
We pick out those functions we think are the most useful for our application without spending too much time on implementation details. We usually start from a serial program which already works and modify it in order to run it in parallel.
We run monte-carlo codes, so we just scatter the problem, and after a large amount of time, gather the results.
We used to use process spawning in PVM but the implementation in MPI is too cumbersome. Many MPI implementations still do not support mutli-theaded applications so it not feasable to become dependent on them.
We use only MPI 1 features
We use the subset that provides the most useful functionality and we refrain from the more exotic ones (in our opinion) like one-sided and thread-multiple.
Whether they are not implemented yet (Parallel I/O) or there is no need so far or nor idea how to improve the code with this mechanisms

Would use dynamic spawning if could be used in conjunction with fault tolerance (maybe I'd spawn off a replacement process). Never had the need for one-sided. Datatypes are OK but a bit clunky for our scientists sometimes... we do use them though. Do any implementations allow for multiple threading? We might use it if it performed well... we really want a separate MPI process that makes process. A lot of our scientists (LANL) get into iSend/iRecv and then find out no progress is made outside of a Wait and they feel cheated.
yet not used, because only standard is 1.0

Question 11

Which of the following do any of your MPI applications use?(Select all that apply)

Threads	336
OpenMP	451
Shmem	117
Global Arrays	107
Co-processors / accelerators	132
PGAS languages	45
I don't know	82
Other	18

Show/Hide Open Answers

ARMCI
BSP
Cell-'threads'
Cilk
computation and communication overlay
CUDA
DDI
I think no one of the above
Math libs (MKL) that themselves use threading
mmap for trivial in-node shared memory
mpich2
none
none of the above
osiris
TBB
Was ist MPI? Max Planck Institut?

Question 12

When answering the following question, please remember that that C++ MPI applications can use the C++ and/or C MPI bindings. Do you have any MPI applications that are both written in C++ and use the MPI C++ bindings?

No	551
Yes	165
I don't know	107

Question 13

The following question refers to the ability to use extremely large count values with MPI operations such as sending/receiving, file actions, and one-sided operations. It makes the assumption that the largest value that a signed C "int" and a default Fortran INTEGER can represent is 2 billion. My MPI application would benefit from being able to reference more than 2 billion items of data in a single MPI function invocation.

Strongly Disagree	53
Disagree	210
Undecided	375
Agree	102
Strongly Agree	62

Show/Hide Open Answers

2^31 has caused so many problems elsewhere, it will be a problem here at some point.
> 31 bit counts would be nice for the future, I don't really need it today. Especially for file operations I'm convinced I will need it someday.
64-bit IO should be supported by default. With an OpenMP/MPI combo, having >= 2 GB per MPI process is more and more common.
All count/index/offset vars should be 64 bit quantities for consistency. Even now, we have problems with > 1 billion cells/particles/items/... . We might have machines that a single process will be operating on > 1 billion things. Seems to make sense to make the change now.
A lot of functionality nowadays is working with large arrays that can be indexed only using 64-bit integer. It is a concern regarding Matrix operations - we do use Integer64 in Intel MKL.
Although I've not required this in my own home-made applications, I'm the author of MPI bindings for Python and third-party users had certainly mailed me about this (more specifically, how to workaround the limitation)
Any limit is sooner or later a problem.
A pathological which can be solved by the apps by blocking up the transfer,
Arrays in my application are easily larger than 2 billion items and now must be split up for transfer (or a suitable datatype created).
As node memories get larger, we're tending towards sending more data per call. We're not yet at 2 billion but want to keep that option open.
assuming 32 bits is restrictive for codes that are scaling to 200K processors
At the moment we use a grid with roughly 3 million data points. Sometimes the arrays have another dimension (factor 10-20 more data). At the moment we are fine, but if computing power is further increasing, the grid size will also be increased. So there could be the need in the future to send such big arrays around.
big arrays of simple data types
big jobs. big memory. big numbres.
By sending such a large arrays you basically mean streaming? If so, it would be useful, I'm not aware of streaming in MPI.
Checkpointing and IO should definitely use offsets beyond 2 billion.
Computing power increases, so do meshes and memory used. At one point, you may need to send more than 2GB in

one shot.
Counting elements on systems with 100000 cores may be limited due to the 32bit data type 'int'. Therefore this feature has to be manually programmed.
Currently I do not need this capability.
Customer requests
data sets get more and more frequently larger than 2^31
Don't know of a need, but also won't be happy if I suddenly do and it becomes a limitation :)
especially in file actions and say global operations 2GB i.e. 'just' 256 million double words are at close distance in the future. So important to have.
file sizes > 2 GB require pointers this size no ?
For future use this would be good.
For reading large chunks of data.
For startup routines it is sometimes necessary to send large amounts of data and it is easier to not have to do a buffered sending.
Given the growth of high performance interconnects, processor power, cost of RAM, cost of storage, and problem complexity, applications will only need to send more data between ranks in the future.
Huge arrays is more and more commonplace and suddenly you find yourself in a position where the vector to be transferred is 1 TB or more.
huge input data
I am working with large data sets.
I could see some value in this. Current implementations aside, the ability to track one-sides ops in these ranges could be quite useful in future implementations.
I don't have to break an array in small sub arrays
If I understand this correctly, it means I could do SendRecv with blocks larger than 2GB, which in the future (comp resources allowing) I will need to do.
... if that helps simulating with larger meshes ... going to smaller number of processing units?
I have no need for this, but in the lifetime of the MPI-3 standard this may become not unusual. Already individual nodes often have >> 2GB RAM.
I marked 'Undecided', I mean, to be future proof, maybe I'll have huge arrays in the far future...
I'm doing numerical simulations. Though not very likely, it can happen that I have to communicate more than 2GB at a time.

It would be a hassle if I had to break it up in chunks of 2GB. If I had to, I would just create wrapper calls around library functions, which do the breaking up for me. Since I'm probably not the only one who needs those wrappers, it would be best for the library to provide them, or just make the basic MPI functions work right in that case.
increasing use of 8-byte integers throughout application space, making it awkward to use smaller integers for MPI communications
*in FORTRAN, INTEGER*8 should be reducable via MPI*
In my MPI application I do not send so large data sets.
In my software, the data structures which are handled have now well above 2G data items. If I am to keep data distribution arrays consistent with MPI function calls, the latter must accept int's which are larger than 32 bits. A 64-bit MPI interface would be just fine to prevent me from maintaining two sets of arrays.
In some places (e.g. message tags) a larger value is desirable. On the other hand: We link to a lot of legacy libraries. Changing the size of an int will break a lot of code!
*In the future I can imagine to allocate a huge vector with shared memory on future sp6-sp7 that I want to communicate with other nodes. Now sp6 has 32 cores a single mpi process can have 128GByte allocation. A real*8 vector can easily exceed th2 2 billion number in future huge calculations. (sp7 will have probably 128 cores/node ?).*
I simulate a small basin having 180 million nodes. To describe completely physical processes i will have to increase it by 1000 times. Also the multigrid solvers increase the number of iterations to reach the solution thus expanding data needs.
I suppose we won't need over-2G data count in the kernel of apllications, but in initialization, for example, we could do communications with a huge number of items. Of course we can code such a infrequent communications with up-to-2G counts but it should be simply boring.
It'd be nice to be able to run tests with larger msg sizes.
I think that mpi is too minute, and we have to use wrappers to communicate bigger dqata sets
it will be helpful for running a large

simulation that will create huge number of individuals (such as modeling bacterial growth)
i use big data on big machines
I use large matrices
*I've just spent some time bug fixing a code that uses integers larger than 2 billion and having to re-cast some variables to integer*8 and leave others as integer*4 is a pain. Many compilers can promote *all* integers to integer*8 but this means that many MPI calls fail because the types no longer match.*
I want to use more than 4GB memory for one-sided operations.
I would rather need large message sizes. Usually, the message size is limited to a much smaller size than the possible address range in memory.
Large database distribution to local disks.
Large data sets to be shared/exchanged
Large matrices.
Large runs of our particle in cell plasma code can reach particle numbers above 2 Billion.
Many simulations nowadays make use of datasets that exceed 2 billion elements. Basically any code with n log n complexity can pull this off, so this extension is highly necessary.
Matrices get bigger and bigger :)
Memory of a single node exceeds 2 billion. MPI needs to be able to, for example, account for 2 billion accesses to memory. Same for sends/receives etc. By limiting counters to 2 billion you require coalescing the MPI calls and the coalescing layer now has to live outside of MPI and MPI becomes a heavy library for large requests that don't exceed 2 billion in count.
Model and problem sizes are growing and users can unexpectedly specify large message communications in their codes without realizing it, so this ability would act as a fail-safe. This would also provide a cleaner interface to MPI codes that use large integers in calls to things like the AMD Common Math Library.
MPI functions using int argument is not suitable for large items in 64bit platform with 4byte int.
My application (quantum chemistry) typically deals with large vectors/matrices which need to accessed efficiently.
My code when running on a large number of x64 machines, runs very quickly which is nice! However, extra resolution would be available should the largest signed

integer be converted to a 64 bit variable
My distributed data structures can have length greater than 2 billion.
My major is Romote Sensing Image Process. I use the large count value usually.
My programs are able to manipulate many and many data
My programs do large file I/O.
Nodes with large amount of memory are now easily available.
no. of elements to be processed may reach 2 billion in near/mid-term future
not actually for my application, so i haven't thought about it
Not currently but it's not crazy to think it could happen.
Not now, but maybe in the future, depending of the further development of hardware.
Not sure if we shouldn't use other programming models instead for this kind of situation
Occasional use for distributing large amounts of initial data. It's not an important requirement.
On multicore nodes, often one can use only one process per node but yet use all of the nodes memory (like 16 or 32 GB). I would think it very likely that when a node is used like this that the count sizes could approach and surpass 2 billion.
Otherwise it is cumbersome to deal with huge data sets.
Our grids are 105 Millon nodes (600M cells). We project that in about 18 months our grids may approach 2 billion cells.
Our library supports this. It also makes implementation of algorithms independent of the problem size
Particle codes, and the number of particles sometimes is larger than 2 billion items
PDE solvers applied to high dimensional ensemble runs with frequent master controlled IO
Probably a useful feature, as long as there are no major disadvantages to going to 64 bits.
Problems are getting large fast these days and 2 billion is not very big. We should have overloaded API's that allow long or int for backward compatibility in both c and Fortran 90.
Remove need to down cast from size_t to int for MPI calls.
required for MPI IO and perhaps some data types
Required if MPI is to be a candidate

programming/execution model for exascale systems.
Sending/receiving 2 GB or larger messages is reasonably common for loading datasets from bulk storage.
Simulation with 4096^3 mesh cells
Size of my current codes is limited by available hardware only. As the latter improves - larger arrays/communications will be used.
Software engineering. *It's too difficult for many programmers to mix integer types. They create subtle bugs that don't occur for months or years... Being able just to throw 64-bit integers into sizes would help avoid some classes of problems.*
Sometimes my applications pass large amount of data through collective communication.
So so!
Starting up MPI aps I want to distribute large database files (>2G in size). Then I want to be able to broadcast (via point to point communications) those files to nodes with MPI. It would be nice not to have to break them into pieces and have multiple sends.
Systems will not get smaller...
The performance, rather than usability, is the key here. For functionality, users can always use distributed data structure to handle very large data set. However how to achieve a good performance across most major systems is quite challenge.
This seems too obvious to me to explain.
We are developing parallel external algorithms handling *multi Terabyte inputs including the current record in the Sorting Benchmark for 100 Terabyte. Furthermore,* *even local RAM sizes are in the multigigabyte range* *by now.*
We are using much memory: just bought 2 machines with 144GB (2 quadcore XEONS). But communication is anyway split up into smaller chunks.
We have a C to Fortran interface for 64-bit architectures that uses long for the C part and 64-bit integers in Fortran is the counterpart
We have occasionally needed to use MPI to do large rearrangements of data and needed to work around MPI's restrictions here. At the same time, these operations are not dominant in our code and are most often related to working around parallel file system deficiencies.
We have to transfer more than 2GB data

which seems to be the upper limit in most implementations. This also hurts when we try to write restart-files, etc in parallel where each process wants to write more than 2GB.
well, local memories are growing, aren't they? With the trend to hybrid parallelisation there could be one MPI process for a complete 2-socket node with a lot of memory, and then ...
well, not yet, but memory get's larger and larger, so if MPI-3 should be future proofe ...
We might reach that size in a couple of years, right now the maximum is ~200 million items.
We never needed this so far
We often have very large files that require reading on one node and distributing to processes for processing. (Wrote fns to send the data in 2GB chunks). The 2billions item limit has also been found to be false in some implentations as at some point the data gets referred to in bytes and hence for larger datatypes the amount of data we can moved at once is less.
We use a self written parallel IO, which needs longs to denote file offset for writing, thus we positively need long integers.
We used to have asynchronous communications with huge buffers, where the default Fortran INTEGER may have become insufficient sometime in the future. We then decided that this was not reasonable and those huge buffers are split into smaller ones (e.g., with several calls to ISEND), at the cost of higher synchronizations in the code.
We use radiative transfer programs where more than 1e9 photons are emitted for realistic simulations.
Why not? Assuming the data type is 'char', I can allocate this much memory on my laptop, let alone a cluster. I can deal with files of this size. I don't see why supporting this should impact performance in the <2B case.
Why not? The monte carlo people here in my site would love it. ;)
With the emerging of new many-cores architectures, the 2 billion limit (32 bit) will become a real problem (for example when doing checkpoint-restart IO)
working in climate research, for the currently targeted resolutions we will need massive parallel I/O of huge files

Question 14

One-sided remote memory access (RMA) is an advanced MPI concept. The following question assumes familiarity with the complex issues involved and deliberately makes you choose between two options that may or may not be mutually exclusive. The goal is to find out which is more important to you, regardless of whether they are mutually exclusive or not. If you are unsure how to answer and/or are unfamiliar with MPI RMA concepts, feel free to leave this question unanswered. MPI one-sided communication performance (e.g., message rate and latency) is more important to me than supporting a rich remote memory access (RMA) feature set (e.g., communicators, datatypes).

Strongly Disagree	13
Disagree	59
Undecided	245
Agree	160
Strongly Agree	71

Show/Hide Open Answers

After all mpi is about supercomputing and performance is paramount, in addition programs are complex enough as that are. Keep it simple, lean and efficient.
Agree, but ... communicators are important, too. don't know about datatypes. also non-blocking RMA's thanks! and collectives via RMA, if possible!
all are equally important
All my friends who ever tried RMA abandoned it after discovering that message rate & latency sucked.
A performnace vs usability question is problem specific, but, if had to generalise.
a rich feature set can be bolted on top. performance can't.
At his point, one-sided communication is too slow to be useful in my application. I have replaced an attempt to use one-sided communication by a send/receive pattern that performed much better.
a true overlay of comm. with calculation would be very advantageous
Because a user may provide additional layers of code around RMA requests to communicate/synchonize if they are not provided by the core MPI implementation
Because I'm interested in High Performance Computing.
Because one-sided do not require hand-shake.
big memory problem
Both are important. Low-latency doesn't mean much other than for little micro-benchmarks.
Cannot judge what will become more important in the future
Communication delay is crutial for speed-up of our applications.
Communicators and Datatypes are the most important part of MPI in all my applications (among other things, the ease/flexibility of slicing 1-D lines or 2-D planes out of a 4-D domain decomposition strategy)
Complex feature set will always decrease performance, while one-sided communications can help improve both performance and simplicity of an application.
Cray shmem :-)
Current code is based on message passing, though we did at a certain point unsuccessfully try to exploit RMA. Although RMA may be useful in certain cases, message-passing appears to be our main mode of operation also in the future.

is important for applications in C++.
Depends on the application I'm running
Don't really use too advanced features.
efficient existing one-sided communication routines would certainly simplify the development of parallel algorithms in computational quantum chemistry
Every case I've seen of someone attempting 1-sided communication (whether via MPI or some other interface) has been for improved latency and bandwidth.
Experience with one-sided MPI has indicates that the implementations are not efficient, the effort in going from two-sided to one-sided was not worthwhile, for the meager performance gains (and often performance losses). Either one-sided implementation are made more efficient by relaxing the correctness constraints OR rich RMA support is provided. The bottom line is that application developers don't care about the paradigm as long as they can get performance gains.
few communication
First of all, RMA should not be there at all in a _Message_Passing_Interface. Having a slow RMA because of data type conversion is even worse (and I didn't think there could be anything worse than the on-sided thing which I really don't like)
for us, performance and scalability are the most important factors for using MPI.
Generally, one-sided is used to shortcut the traditional MPI_Send MPI_Recv route for message passing. Because of that, a simpler, but higher performing interface is generally preferred.
Given good performance, I can overcome limited features. (But not the other way around.)
I agree somewhat. I think supporting communicators for example is important too, with complex datatypes less important.
I am very willing to trade better latency for structured data, as my data type set is small, static, and I don't have interoperability requirements.
I can give an opinition about that because I have nor used it never on my applications
I can not compare the performance of these options.
I do not undestand it
I do not use one-sided communications
I don't use one-sided communication.
If I bother coding up one-sided, it is for

speed. Usually use shmem instead of mpi though - I've never go them to mix very reliably except on quadrics
If its well implemented allows for overlap of communication and computation better than nonblocking comm.
if one goes through the effort of adopting RMA, likely it is because performance is critical. We have experimented with RMA, but had to back out the changes because performance was poor.
If people want to write performance-sapping stuff on top of the existing basic RMA functionality, that's well and good, but if RMA performance starts sucking I'll have to stp using it.
If performance doesn't matter, one can use existing MPI features - MPI two-sided, threads, and probe - to get one-sided behavior.
If performance isn't great, I wouldn't use one-sided communication.
If simple operations don't show high performance, I will not invest more time in coding complicated (=error-prone) operations, that might not perform either.
If the fundamentals are correct (communication performance), the application can handle the rest.
if we do HPC, performance counts - otherwise we can switch to java and ignore optimized blas
if you don't care about performance the current one-sided implmentation is fine. The balance between performance and functionality is important. if it is simple message injection its useless without a supporting memory features that allow for say an accumulate operation
I have no idea of the overall gain when using RMA in my applications
I have no strong opinion actally but I am already accustomed to MPI implementations that only support low level things efficiently.
I have not tried the RMA features yet, so I do not know whether our projects can benefit from it performance-wise.
I like generality, but not if it compromises performance.
I may have use for one-sided comm in my app, but i'm not rewriting it for RMA.
IMHO, simpler is better. Optional hints to improve the performance would be ok.
*IMHO, this seems to be the wrong question - as long as the *semantics* of RMA remains as contorted as it is in MPI-2, that keeps most developers from touching it.*
I'm not really using RMA so far, but if I would, I think that performance is an

issue.
I mostly write and teach to write malleable and portable applications; as such, being able to define a proper parallel structure for the app with limites SW development effort is more important than the raw performance of one-side communications. Except for top-scale supercomputing, the development/maintaining effort of applications is more significant than the performance gain.
I need to map different memory areas as the same data type.
In my impression, one-sided needs more careful coding than two-sided. Therefore, if the performance is not unattractive, I don't have any reason to dare to use them.
In our case, I think one-sided communications could be used to just 'read' some simple data in the memory of other processes.
In the end, complex datatypes are also made of bits and bytes. In principle, you only need the MPI_BYTE data type for all your communications. However, communication performance is crucial if you want to speed up your application.
in the end I'd be (slightly) more interested in speed.
I see RMA as quick access to well-structured remotely-resident data and believe that performance matters most.
Is not more important, but equally important as RMA such as, for instance, communincators
is RMA really useful?
I suppose you could have a first implementation that performs well and supports the basic functionality and worry about extending it to more specialized fields afterwards. Users can test the current functionality and send feedback while more features are added.
It depends on application-specific and running conditions/configuration. For some cases RMA is not best choise
It depends probably on usage. Both options should be available.
I tend to favour performance of basic operations. If performance and features are mutually exclusive, then it is a difficult balance and needs to be answered by someone who actually uses these features (I currently do not).
I think, algorithms can written in a more flexible way, when there is a rich RMA feature set. Of course performance is also very important, but flexibility is more important for me

the same performance (as high as possible) if there are richer features included as well or not. Complex features might run at a lower performance, but they should be just as optimized. Ideally, implementing them in the application using basic features only should not provide more performance. You have a bit more flexibility when implementing them inside the library.
I think message rate and latency are overrated. It is often the synchronization that kills (RMA) performance. However, rich features may require more sync or more data copies at lower layers - therefore I chose undecided.
I think MPI one sided should be easier to use. There should be a 'minimal layer' that's very fast and a 'convienience layer' that sacrifices some performance for ease of use.
It is essential for every kind of communication that it is of high performance.
It's deceptive. In order to know when the transfer has completed, you need to add most of the calls needed for two-sided, and you don't get the advantages of checkability. The language specification problems are a nightmare area in any of the (dozens) of languages I know.
It would be nice to have signaled put and get operations, but I don't need fancy datatypes.
I use mainly applications that require high percentages of long range communications. Message rate and latency is a major bottleneck.
I've seen no evidence of a desire for one-sided communications, but rich RMA could lead to code that is easier to understand and support - so an efficient implementation would be desirable.
I view one side communication as a convenience rather than a performance optimization at this point.
I view one sided stuff as a tuning to reduce communication latency. The only data type I really care about is a contiguous number of bytes.
I want RMA operations to be faster then point-to-point. If one-sided is not faster than normal send/recv there is not reason (for my point of view) why they should be there.
I work as a performance engineer on Cray systems. To our group, performance is far more important than functionality (though I recognize that the reverse is

true for many people). Missing functionality in MPI can be replaced by other mechanisms, perhaps at the loss of portability.
I would like to have both
I would not say performance is more important than features. I will start using one-sided communcations when it has both the reasonable performance and functionality.
I would prefer one-sided communications to provide the lowest latency possible. Other functionality can be derived.
Latency for next neightbour point-to-point communication has the biggest impact on performance, while the most complex datatype needed is a subarray.
Latency hiding is very important in general...I am prepared to go to low level for performance, even at the risk of losing portability and good coding practices.
latency is a big issue for us and is why we sometimes bypass MPI and use lower level comms routines sometimes
lattice QCD calculations strongly depend on MPI communication performance
Main purpose of one-sided communication is better performance and simpler programming. Application can use two sided communication for derived data types or additional communicators.
Most of my applications are both latency and message rate critical. RMA is very useful when using co-processors/accelerators. RMA has the potential to provide the performance I need. However, more features are welcome, such as remote read-modify-write. This could be useful for quick lightweight synchronization
MPI one-sided communication is used for performance.
Must be fast to be used
My application, lattice QCD, would benefit more from improved bandwidth and lower message latency.
never thought about it
not used
Obviously I want both but exposing the fundamental network operations with good performance is more important than building high-level features.
Obviously, performance will be tweaked (in HW and/or SW) over time IF people really heavily use provided MPI one-sided communication functions. However, doing so will take time and not offering them in the first place, will stop the process right at the beginning. Allow experiments!

crucial is some of our applications, however, often we require it work over custom communicators. Being forced to run RMA only over say MPI_COMM_WORLD could be problematic.
One-sided operations are typically used for short (8 byte) messages with less strict ordering requirements. Limited features are just fine.
Our application benefits mainly from highly performing parallel linear algebra. Thus, message rate and latency are crucial (also in RMA).
Our applications need to transfer all the data in each process to the other processes in each iteration, with each process receiving an equal portion of the data transferred. This all-to-all communication typically takes 25% of the machine time, so it is paramount for us to speed up (massive) communications as much as possible.
Our applications tend to be communications latency dominated above all else. MPI design (i.e. any process can send to any other process a message of any size at any time) tends to result in library implementations that are not as low latency as possible for our application. *Thus, to improve latency, the aforementioned communication layer we use is already designed to wrap thinly around a to-the-metal RDMA communications interface if available. And, we already have successfully used implementations of it based on hardware specific RDMA libraries (e.g. Verbs on Infiniband).* *To enable a lower overhead RDMA protocol than most stock MPI implementations, our communication layer is based on packet exchanges and requires the application to bound the maximum packet size that can be exchanged for each link. This is straightforward to do in our applications.* *Thus, if MPI supported very primitive one-side RMA type primitives efficiently, we certainly would try to make use of them and could likely do so very quickly.*
Performance always is primary. Features which have poor performance are not used.
Performance critical program can be written using Point-to-Point message passing, RMA's role is to support another parallel programming paradigm.

Performance is certainly extremely important, but without a one-sided semantics that avoids excessive synchronization, there's little point in using one-sideds versus two-sideds.
Performance is key.
performance is key for one-sided operations
Performance is most important for my work
performance is the main motivation for using this capability
Performance is very important!
performance is very important and a rich feature set is not necessary for our application
performance is very important but features like noncontinuous rma (as mentioned before) would also be an improvement
performance is very important, but without derived datatypes and the Cartesian communicator it is useless for my Lattice Boltzmann application
Performance matters
Performance matters most to me.
RDMA APIs are better.
RDMA is mostly important to organize asynchronous work. So in general supporting a rich memory access feature set would be nice.
Rich remote access are quite always not optimal anyway. Let's stick to the basic functionality offered by quite a lot of IC nowadays : RDMA (and really benefit from it !)
RMA? I'm usually using SHM MPI Device for communications inside one node. And in a nutshell I do need better communications between remote nodes - inner communications can be done without MPI using threading - that's not an advantage of MPI. Though I hope that RDSSM will be improved.
RMA is not needed in my applications
RMA needs to be combined with datatypes. This is more a hardware capability question than a software support layer question. Ideal would be RMA hardware, that supports strided access patterns with MPI Datatypes. Only in point-to-point szenarios RMA with block-windows can beat Datatypes. In collective operations, where aggregate bandwidth is key, RMA usually doesn't pay off, but overlapping gathering & scattering of data with the actual transfer pays off huge. In essence I would not want to sacrifice one for the other.

question. At present MPI_Put/MPI_Get does not have any advantage to MPI_Send/MPI_Recv (and all its variants). I've tested this in quite a few codes.
Simple, 90% of the users tend to use 10% of the features. Therefore, it is best to emphasize on performance of commonly used features first.
Simple operations that are fast can be used to model most everything else effectively. Having the core functionality work really well would enable any sort of custom functionality I would want to build on top.
Since a lot of my work is with discrete event models asynchronous messaging is important.
Structure of my remote data is very simple; only speed is of practical interest.
Structures like communicators, datatypes make the application much more readable. I won't give this up for higher performance.
supporting a rich remote memory access feature sounds too complicated. Keep It (MPI) Simple. That way, there may fewer issues with the implementations. I prefer solid, stable software to highly featured, quirky software.
The current performance limitations on one-sided communication is restricting adoption of the paradigm. Enabling better performance as the initial step will go a long way towards encouraging usage.
the features mentioned belong to the (mature) basics of MPI, they should not be abandoned a priori in favor of hypothetical performance benefits
the hole point for using DRMA is that it gets higher performance, so a direct mapping of functionality to the hardware seems the best option even if it is a low level solution.
The main point of this is to reduce transfer overhead an latency. If it is very fast you can pass each piece of a complex data type separately and still get better performance.
The MPI one-sided routines are difficult to use. If one had the Cray shmem syntax for one-sided routines with high performance, this would be useful.
the need for RMA has not come up so far, so if/when it does, it is unclear which factor will be more important
There seems to be much marketing behind one sided operations, and at first glance they look attractive. In the real world they seem to be of little use.

The whole point of one-side communication is performance, but there interface should be accessible to the average application developer.
The whole reason to do one-sided comm is performace!!
They both are important.
This is a hard question to answer, but I lean towards more flexibility in programming rather than raw performance. Maybe there could be a switch in the code for truly optimal performance at the cost of some features.
this is difficult. For an expert in MPI this might not be as crucial, but for people starting to develop new codes/adapt older codes performance AND usability is crucial (rich feature set)
Throughput matters for one-sided, but there is no point in using one-sided for latency-critical messages (same for message rate). I assume that having a rich RMA feature set may disturb latency and message rate but not throughput, so that's OK.
To me, communicators with one-sided comm are mostly useful for translating to processes for point-to-point one-sided communication. But then the MPI RMA performance is so poor that I avoid it and use some other mechanism (GASnet, GA).
To me, one-sided should be used in order to support true asynchronous remote updates, i.e., you´re either using an old or a new value of a datum, you do not really care. The other use is to improve performance. My take is that neither of the above can be done using todays one-sided semantics.
Unfamiliar about the difference.
Up to now, implementors of MPI libraries appear to have put no effort into improving one-sided performance, with the argument 'nobody is using it'. Nobody will use a poorly performing one-sided implementation, no matter how rich the functionality it supports. Minor changes to the existing chapter of the standard should be enacted such that implementations can omit any unnecessary overhead in checking overlapping accesses etc. and still remain standard compliant. Performance is the most important aspect in this regard.
Use point to point if you want rich features.
usual functionality vs efficiency
Various levels of RMA is built in any hardware these days. MPI should not mask it with rich API with excessive

runtime overhead.
We already use MPI wrapped in a small number of higher level communication classes dedicated to our application, we only expect performance and portability. We don't want to rely on complex features that might be unoptimized or bugged in some vendor libraries.
We are facing the problems of latency in our communications when trying to scale an application beyond couple of thousands cores, so for us it is really important reduce the effect the latency, thats why we are porting our codes to hybrid OMP-MPI in order to reduce the number of message and the same time that we increase the size of itself as well. *We are still afraid of how our application will run in high core count using MPI at the node layer and OMP withing the node.*
We can always write our own wrappers for usability.
We consider it our job to puzzle out and design parallel algorithms, including comm, and we can design them to use 1-sided comm, presuming the recipient can respond using 1-sided comm when necessary. We are not afraid of programming; it is better to have one highly optimized building block from which we can build a custom comm engine, than to have a rich but slow general purpose machine. Give us something simple so it can be crazy fast, and let us worry about complexifying it.
We don't find ourselves limited by MPI's current RMA feature set, OTOH as we scale our software up to the hundreds and thousands of cores the communication performance is limiting.
While communicators and datatypes can be very convenient, I can 'fake' them manually. However, I can not make a poorly performing put or get fast (without moving to a different mechanism such as CAF).
whould be nice if this allows complex applications crossing borders between operating systems
Without good performance RMA is pretty much useless for me as would have to implement every thing with two-sided communication in an extra thread without the nice semantics of RMA. *I would rather implement handling of complex datatypes myself, should that need arise.*
Would use it only in performance critical code kernels where all synchronization/dependency issues are

handled at a rather low level
Yes, would use MPI one-sided ops if they were faster. We have our own low-latency comms library using librdmacm

Question 15

The MPI standard provides certain semantic guarantees that may not be required by a particular application. It also provides functions that many applications never use. The MPI Forum is considering an "assertions" interface that would let an application identify specific functionality it does not depend on, such that an MPI library could improve performance or reduce memory usage by disabling that specific functionality. The described "assertions" interface would be valuable to my MPI applications.

Strongly Disagree	7
Disagree	23
Undecided	244
Agree	375
Strongly Agree	110

Question 16

The following is a broad list of topics that the MPI Forum is considering for MPI-3. Note that it is probably safe to assume that using any of the new functionality will involve at least some degree of change to your existing MPI application (e.g., it is unlikely that MPI-3 applications will automatically become fault tolerant; it is much more likely that you will need to add additional fault tolerant logic using new MPI-3 API functions). If you are unfamiliar with a given topic, feel free to leave its rating blank. Rank the following in order of importance to your MPI applications (1=most important, 6=least important):

	0	1 (most important)	2	3	4	5	6 (least important)
Non-blocking collective communications	181	243	135	120	86	45	28
Revamped one-sided communications (compared to MPI-2.2)	267	50	76	115	90	145	95
MPI application control of fault tolerance	223	74	129	125	144	95	48
New Fortran bindings (type safety, etc.)	210	68	72	78	64	99	247
"Hybrid" programming (MPI in conjunction with threads, OpenMP, ..)	160	217	175	105	89	59	33
Standardized third-party MPI tool support	223	32	84	103	132	140	124

Question 17

Rate the following in order of importance to your MPI applications (1=most important, 5=least important):

	0	1 (most important)	2	3	4	5 (least important)
Run-time performance (e.g., latency, bandwidth, resource consumption, etc.)	105	397	206	89	27	14
Feature-rich API	162	14	38	70	283	271
Run-time reliability	125	149	201	271	62	30
Scalability to large numbers of MPI processes	114	158	254	225	70	17
Integration with other middleware, communication protocols, etc.	170	17	31	55	234	331

Question 18

Use the space below to provide any other information, suggestions, or comments to the MPI Forum.

Show/Hide Open Answers

-
;-)
1) quibble: Shouldn't that 'assertions' interface be more like a '#pragma' interface? I.e. let us specify features we want turned on or turned off? *2) IMPORTANT. We have tested MPI in conjunction with threaded linear algebra libraries (ATLAS in particular) and it kills the performance of BOTH. You need a switch that lets you play well with others; we may be calling a threaded linear algebra library (like ATLAS, Goto, or PLASMA), a threaded graphics library, and other threaded libs all in the same application.*
A consistent implementation of collectives for a given network (latency, bandwidth) would make my support work a lot easier.
Again. *Rework the One-Sided Communications and give us non-blocking collectives!* *And focus on algorithms for performance.*
already great work :)
As a matter of fact, I use MPI just as a portable communication layer for middleware implementations, nothing else. *From that point of view even the current MPI standard is already overspecified and induces a lot* *of unnecessary overhead.* *Thus I am not interested at all in complex features, even issues like support for collective communication or heterogeneity are unimportant, since these problems are solved elsewhere. Therefor I'd strongly opt for a concise subset of the MPI standard that is just able to deliver high-level, minimal overhead access to state-of-the-art communication networks, including networks with user-level communication facilities.*
As I have told before, mpi should have two types *of communications say intra node and inter node.* *(may be three the noremal one also).* *In this way we can tell exactly what has to be communicated and how. And we can easily optimize.* *This would be beatiful and should fit the new supercomputer architectures.* *OpenMP is easy and sometimes efficient but makes* *a lot of stupid and unnesessary cmmunication intra node. I bet in most cases I could do better but I do not have the language yet (I think).*
As we share subroutines among several developers and research groups, it is very important that the currently used codes (partly many years old) do not have to be changed if somebody adds features that require the new standard.
Avoid one-sided functionality in MPI altogether!
Better documentation of the C++ binding would be highly appreciated and would help with the integration of advanced MPI 2 (or 3) features.
 Thanks for this effort in MPI 3
cannot comment since i am not completely familiar with the power of MPI.
clean up interface!
consider MPI to be the corner stone of parallel computing, keep it functional, performant, understandable.
Don't hasten out the next standard version
Dynamic process management (especially MPI_JOIN) would be infinitely more useful if the MPI Forum actually made statements about how start-up and peer discovery should work.
Enforce that FORTRAN mpi module must be provided and for backwards compatibility also mpif.h. *Make parallel I/O portable.*
Fault tolerance please! :)
For multiple mpi processes (pure mpi model) on a single node better shared memory comm. is needed, L2 only comm. Numa control etc. For hybrid models thread safeness is needed.

defined vocabulary is required. So posing more pressure on documentation giving more information.
Good Luck!
I am afraid that the mentioned 'assertions' interface may allow to specify inconsistent subsets of the MPI API. I would strongly suggest that the MPI forum defines not more than a handful of subsets. Let the MPI implementors make the suggestions for the calls and features that cause the most trouble in the implementation.
I cannot rank/answer the upper too questions due to lack of knowledge
I'd rather see MPI-3 optimized and faster than 'rich' in more API. Mechanisms for fault-tolerance and malleability would be extremaly useful. Performance portability is something you should think about. What if besides all kinds of XSend and YRecv operations, you'd have one 'default' send and recv, and which is best could be determined automatically based on hardware/architecture etc.
I hope newly introduced functionalities will be carefully designed paying much attention to their performance. If perfomance is poor, I'll never use. More importantly, some people will use new features without being aware of their performance to make their applications slower.
Important features would be: *- Fault Tolerance!!* *- Convenience functions for debugging purposes!*
I'm very optimistic about the future of MPI. It seems to be getting a lot of energy from the OpenMPI project -- wonderful stuff.
In addition to better one-sided communication I'd also like to see active message support and possibly support for hierarchical communicators to support hierarchical architectures.
I NEED NON-BLOCKING COLLECTIVE OPERATIONS. It's either I'll die without it or wright my own realization.
In general I'm very happy with MPI, it allows a large set of applications to 'just work' with a large set of hardware making them into productive research resources. Any support for migration, or virtualization would allow more flexibility for long running jobs which would be quite valuable.
In my application I contend against non-repeatable results depending on the number of processors used (I'm quite sure that this results from an ill-conditioning of my system of equations but I can't change this) A feature which improves reliability (e.g. sum up values always in the same order) would perhaps help
In the last question, rank 1 - 3 are pretty much as important as each other.
I strongly appriciate you all time.
I suspect fault tolerance will be outside the scope of MPI.
- I think we should not need OpenCL or the MulticoreAPI (MCAPI) to support multi-core hybrid heterogeneous computing. MPI should be enough. At least, from the semantics and syntax perspective. Keep parallel programming as simple as possible by reducing the number APIs, we already have enough of them out there. Perhaps, introducing 'MPI profiles', targeting different application segments (large processor count profile, embedded profile, accelerator profile, grid-computing profile, etc.). This could enable many optimizations. *- I think one-sided communications can be very y useful for multi-core and accelerator-based systems. The 'Remote' part in RMA might not be that remote (same chip, shared memory) and MPI can provide a standard way to access this memory. Simple accelerator RMA example: CPUs open up the windows, accelerators read the data and put the results back, then synchronize. No need for point-to-point, and it can be done in the local host or a remote one.* *- Although there is no silver bullet for parallel computing, MPI should be the closest to that.*
It would be nice to access information on semantics provided by an implementation that are beyond what is required by MPI (for example, message ordering guarantees that are stronger than the MPI non-overtaking rules).
It would be nice to make the MPI standard more strict. Currently MPI implementations have too much freedom (e.g. OpenMPI is quite different from say MPICH2 and clones)
I use MPI only with F90, it works but it don't make fun. Compiler optimization sometimes cuts out dozens lines of code. Wrong usage of subroutine calls (forgotten one argument, ...) doesn't throw compile errors but throw segmentation faults, why(?), its hard to debug

because you expect compile time errors if you call an API in the wrong way. For sure some errors are not the fault of MPI(-Standard) but in the end the simple programmer don't care about whose responsible but may stop using MPI.
I was working for Cray, we had shmem, developed by Bob Numrich. It was just simple and fast. Please keep performance in mind, do not overload the standard with issues which might be interesting for computer science, those people who are interested in the usage of computers, but not so much the computational scientist, the person who just wants to use it and needs to get a difficult task done.
Java applications are starting to be run on HPC resources. It would be valuable to have some initial standardization or bindings for Java.
Keep It Simple. A high performance scalable reliable core is far more important than the bells and whistles... and often, an application can create a better/customized version of the bell&whistle features.
Keep up the good work!
Less would be more.
Memory footprint is an issue - larger core numbers sometimes provoke ridiculos pre-allocated buffers. Probably not really a standard issue, but control of the max amount of memory used might help.
MPI is really at a crossroads right now. For fundamental reasons, hybrid programming is becoming ever more important, and on the other end, Petascale machines drive up the MPI scalbility requirements. I'm not sure that both constituencies can (or should) be served by one standard - maybe a bifurcation will, in the ned, provide better solutions for everybody.
MPI is too bloated we should try to look into other message-passing based paradigms like erlang or scala to make the API simpler to use. A function call with more than 10 arguments scares people. :)
MPI will only survive if it is simplified
My big bugbear with MPI is an implementation issue and not obviously addressible in a standard, but here goes anyway. *Debugging should be a priority for implementations. Diagnosing hangs and MPI errors is extremely difficult and unscalable to large number of processes. A 'debugging mode' where collectives check their arguments and provide usable traces and error reporting woud be a big boon.*
N/A
no comment
Nope. You guys are doing a great job. Thanks.
no suggestions
One this I did not see was overlapping communication and compute. This was one of the main features of PVM that most MPI implementations ignore.
Please provide benchmark programs to evaluate the vendor MPI-implementations of all major concepts.
Please publish the results!
Please revisit some early proposed APIs and to make them solid. If some features are so powerful in spec and most people got trouble to make it right and fast, what's the point? *Put in another way, if some existing APIs don't have a good implementation, maybe it's time to see what's going on, and why, rather than to include another function set.*
Please see earlier suggestion.
- recv with timeout option -- good for fault tolerancy. *- F90 interface.* *- a function to detect whether MPI is threadsafe AFTER MPI_init has already been called (I know there is MPI_init_thread, but if someone already has called MPI_init earlier -- and I do not have access to that library)* *- Keep C++ binding for god sake !*
*Regarding the last question: I need an MPI that allows me to solve problems using minimum resources, which also includes development time (cf. usability). I don't necessarily need a 'feature-rich' API but rather one with the *right* features to enable also the implementation of complex but more efficient algorithms. Performance should always be targeted towards the real applications not only to specific parts (like lat or bw) that are nice for benchmarks but maybe counter-productive for the applications.*
Scalability is still THE issue for the upcoming years (10^7 MPI processes) together with

fault tolerance.
see earlier suggestions box in here
Some optimization options could be of great help.
Stability and performance much more important than feature-rich API.
Standard utils library, instead of barely needed API. *Processing fault such as one node dead, even if in a big granularity.*
Thanks a lot for this good work.
Thanks for involving us in the process !
Thanks for the opportunity to contribute our thoughts about the MPI-3 standard.
Thanks for your efforts.
Thanks to all devoting their time to this effort!
The C++ bindings are virtually useless. All C++ users I know start from the C bindings. C++ can offer some great advantages (eg, the boost MPI library) but the design of the C++ bindings is a disaster. I don't think anyone would complain if the C++ bindings were omitted from MPI-3 (most of the bindings can be implemented on top of C anyway, and porting a C++ code to use the C++-on-C shim is probably easy in most cases). I doubt there is enough C++ expertise in MPI-3 to consider a new set of bindings in this round of standardization. It would be much better to let library developers gain experience with the new MPI-3 features for some years to learn how best to use these with C++.
The current RMA interface is a non-starter. Get rid of it and start over.
The insistence of MPI to support non-cache-coherent architectures is one of the worst things to ever happen to the rest of the HPC world.
The last point (integration with other middleware) *can be deferred to additional libraries/wrappers,* *so I thought this was not so important.*
The mpd job launch mechanism used in MPICH2 has been problematic at my site (does not work well with job schedulers such as LSF when the scheduler gives overlapping hosts to the same user running multiple MPI jobs; job launch failures when submitting to more than 50 hosts). The job launch mechanism in OpenMPI is much better. Perhaps robust job launching will be addressed in MPI 3?
The standard is too permissive and includes too many features. As a consequence implementations are bugged or unoptimized...
This is more a hardware request to which MPI could greatly take profit: parallel computers should have 2 networks: *1 efficient for point to point communications* *1 efficient for broadcast, global communications*
To me, the most interesting parts of the MPI-3 work is the new Fortran bindings, and better support for hybrid programming. *I'd also like the new standard to be implementable with a reasonable amount of effort, such that we might actually see conforming implementations within a reasonable time. Also, providing an incremental upgrade path for existing MPI applications is, I believe, crucial to the success of the effort.*
Try to minimalize integration with other software from the MPI side. It is a structural time sink, and introduces the risk of MPI (partially) breaking once one of the other components receives a major update.
We don't use a lot of MPI's power, but there are a lot of users like us who use simple almost-batch workloads.
We have a lot of library developed in MFC, So I hope I can make GUI easier
We understand the need for introducing some fault tolerance at large scale (>10,000 MPI tasks) but we are undecided on the right approach. Anything that MPI can do by way of a _standard_ would be enormously helpful. While we are looking at PGAS for some parts of our applications, this is to get around physics issues and hardware limitations, rather than any dissatisfaction with MPI - I expect we will be using MPI indefinitely (>>10 years).
We were not able to successfully apply asynchronous RMA a while ago. Performance was very bad and we needed something like a 'Critical Region' to implement our algorithm efficiently.
Will be following MPI-3 efforts closely. Thanks for your hard work, and Happy New Year.
You do a great job! Thanks for all the heavy lifting.