Charm++ - Bug #2060
Communication performance degrades after succesive load balancing when migrating many objects
03/20/2019 10:13 AM - Juan Galvez

Status: New
Priority: High
Assignee: Migration
Category: Migration
Target version:

Start date: 03/20/2019
Due date:
% Done: 0%
Estimated time: 0.00 hour
Spent time: 0.00 hour

Description
This was observed on Blue Waters, running on 128 physical nodes, with chares communicating in a stencil-like pattern with large messages sizes (80,000 bytes). Doing AtSync load balancing, with +LBSyncResume and a barrier after resumeFromSync, and measuring the time it takes for the program to complete a series of iterations between load balancing steps, the time increases after each lb step (see attached plot). As far as I can tell, the performance keeps degrading and I haven't seen it reach a ceiling yet (I have run up to 300 lb steps). Also see the attached test program to replicate the issue (I ran with `+balancer GreedyLB +LBPeriod 0.001 +LBObjOnly +LBSyncResume +LBCommOff`). Issue is observed in both non-SMP and SMP mode.

The issue appears to be in location management, since that is the only outstanding process after a load balancing step (in this test computation resumes only after all migrations have completed). Also, in a separate test I implemented an application-level mechanism to force the update of locations before resuming computation, and that restores the original performance. And finally, if I "reset" the location tables on every PE like so:

```c
void CkLocMgr::resetLocationTableEntries()
{
    std::vector<CmiUInt8> ids;
    for (auto itr = id2pe.begin(); itr != id2pe.end(); itr++)
        ids.push_back(itr->first);
    for (auto &id : ids) {
        if (homePe(id) == CkMyPe()) continue;
        updateLocation(id, homePe(id));
    }
}
```

the original performance is also restored, and slowly degrades again afterwards unless I keep "resetting" the tables in this fashion. The reasoning behind the above code snippet is that the best performance was obtained at the beginning, when location managers assumed that chares were in their home PE. But still not sure where the exact bug lies.

History
#1 - 03/20/2019 10:28 AM - Juan Galvez
- File loc-bug-charm++-smp.png added

I should add that in the test program, all chares have the same load, so the performance after load balancing should always be the same (should be a flat line instead of continuously increasing).

#2 - 03/20/2019 01:05 PM - Laxmikant "Sanjay" Kale
What happens with rotateLB? That could clarify some issues with a more controlled scenario.

#3 - 03/20/2019 01:18 PM - Eric Mikida
How many chares are you using in this case? I know this debate has come up before, as to whether we should proactively update location tables after known migrations (ie LB), or let the updates come on a need-to-know basis as messages bounce around. I think the debate pre-dates my time at PPL, but from my understanding, the hesitancy against doing full updates after LB is that it fills all PE location tables with information that it may likely never touch.

04/10/2019
A couple of thoughts for this particular case:

1) I wonder if the issue would be as bad if communication locality was preserved, since this is a stencil computation.

2) How frequently are you load balancing? I could imagine that with frequent load balancing, the location tables are not updated quickly enough and are essentially always lagging behind the migrations. How many messages are sent from each chare to each other between LB calls? In the extreme case, if it were just 1, then that message would trigger a location update, but then that update would never be used because the chare would move before the next update.

3) I think there might be (or at some point there was) a compile time option to force full location updates after LB. `CMK_GLOBAL_LOCATION_UPDATE`. Is that what you used to force updates, or did you do something else.

If this is not actually a bug in the code, and just a configuration issue, maybe we should re-evaluate the default behavior. Or even print out some kind of end-of-run warning if we can detect a lot of lagging location updates, saying to compile with global updates on?

---

Laxmikant "Sanjay" Kale wrote:

What happens with rotateLB? That should clarify some issues with a more controlled scenario.

I second this. It would at least see what happens with the original communication locality preserved.

---

Eric Mikida wrote:

How many chares are you using in this case? I know this debate has come up before, as to whether we should proactively update location tables after known migrations (ie LB), or let the updates come on a need-to-know basis as messages bounce around. I think the debate pre-dates my time at PPL, but from my understanding, the hesitancy against doing full updates after LB is that it fills all PE location tables with information that it may likely never touch.

A couple of thoughts for this particular case:

1) I wonder if the issue would be as bad if communication locality was preserved, since this is a stencil computation.

Communication locality should not be an issue, in the sense that if I run the same simulation using a random map for initial placement of chares (with very bad hopbytes), and use DummyLB, the performance is good.

2) How frequently are you load balancing? I could imagine that with frequent load balancing, the location tables are not updated quickly enough and are essentially always lagging behind the migrations. How many messages are sent from each chare to each other between LB calls? In the extreme case, if it were just 1, then that message would trigger a location update, but then that update would never be used because the chare would move before the next update.

I am doing load balancing every 5 iterations (approximately every 320 ms), so it could seem that the location update mechanism cannot cope with this frequency. However, I'm not sure about this, because the mechanism I implemented to update locations was to go through every chare in sequence, where a chare does a ping-pong to its neighbors, and then the next chare does the same, and so on until the last one. After this I resume the simulation. This process takes a long time because there is no parallelism (10 seconds at first I think). After each LB step it would take longer and
longer (up to 20 secs if I remember correctly).

3) I think there might be (or at some point there was) a compile time option to force full location updates after LB. `CMK_GLOBAL_LOCATION_UPDATE`. Is that what you used to force updates, or did you do something else.

I used a different mechanism (see above). I’ll try to test with CMK_GLOBAL_LOCATION_UPDATE also.

If this is not actually a bug in the code, and just a configuration issue, maybe we should re-evaluate the default behavior. Or even print out some kind of end-of-run warning if we can detect a lot of lagging location updates, saying to compile with global updates on?

I just ran the test with RotateLB, and the issue is not present (at least up to 300 lb steps). I guess that with RotateLB message bouncing after migrations will happen inside the same physical node or adjacent node which is why the issue is not observed.

---

#6 - 03/20/2019 02:24 PM - Juan Galvez

The issue happens also with CMK_GLOBAL_LOCATION_UPDATE.

Also, I told this to Eric Mikida offline, but for the record just want to note here that `updateLocation(id, homePe(id))` in the above code snippet updates the location of chares to make it point to the homePE (I am **not** updating it with the correct location). I only do this on non-home PEs because I don’t want the home PE to lose the correct location.

---

#7 - 03/20/2019 04:41 PM - Eric Mikida

Another experiment to try that would be particularly useful would be to only call LB once. We should see an initial spike in iteration time as the location updates trickle in. But then it should stabilize back to initial timings as all the caches are updated. If it doesn’t then it’s definitely some kind of bug in the update code.

---

#8 - 03/20/2019 11:05 PM - Eric Mikida

Just saw what you said about GLOBAL_LOCATION_UPDATE. That is surprising. After looking at the code, I had a few more thoughts I will look into more.

1) With large messages (which this test uses), if you set a message buffering threshold, CkLocMgr attempts to completely avoid multihop messages by sending locationRequests before actually sending messages it is unsure about. Although...on inspection of the code, after the update comes in it will never hit this path again because it will always think it knows the location of the message.

2) I don’t think there is anything in the location update logic that prevents multiple updates to be sent to the same PE for the same chare. ie, if I send 10 messages in quick succession to a PE where I think a chare has moved, and those messages take multiple hops to find the chare, the chare will send 10 updateLocation messages to the sender. That is one potential way that we could be stacking more and more overhead.

3) Related to the above, I don’t believe there is any check for update ordering in the code. So if two updateLocation messages from right before and right after a migration arrive on the same PE, then the order they arrive will determine which PE is considered the real location.

I will try to run the test program above at some point to trace the path updates are taking and see how many times messages are hopping etc.
Here is what I think is happening:

When an object migrates, the old PE updates the location to point to the PE where the object has migrated to. The problem is that if there are no senders that remain in the old PE to fix that location as the chare continues migrating in successive lb steps, the PE ends up with an incorrect location. As an object moves it can thus end up setting a chain of "pointers" which replicates the migration path that the object followed, and this gets longer and longer with each lb step.

Now the problem occurs if a sender suddenly migrates to one of the PEs in this chain (which can happen with Greedy and a random lb). The messages will follow that chain until the destination tells the sender to update the location. It may seem that the chance for an object to land on a PE where a neighbor was is very small (I ran the test with 2048 PEs in non-smp mode). But given that there are 16384 objects, the probability that no chare lands on such a PE is almost 0.

If I modify `CkLocMgr::emigrate` so that when an object migrates the location is updated with the home PE instead of the PE where the object is migrating to, this gets rid of the steady increase in computation time in my test case. It does come with a slight performance cost though, which I assume is because this always forces PEs to send to the home PE first after lb (even if they actually know where the receiver is).

I'm not sure what an ideal solution could be. It might be better to purge the entry if no senders remain after lb. Or maybe complement this with information from the load balancing strategy (for example: how many objects migrated) to decide whether entries should be purged or not.

Ah that's interesting. It is definitely possible that `CK_GLOBAL_LOCATION_UPDATE` is broken since it is AFAIK untested. And it explains RotateLB performance. In some sense the communication locality is "important" in the sense that if the communicating objects migrate together, you won't end up with a sender suddenly arriving at a long chain. I still find it weird that the time just continuously increases more and more, as I would think we'd instead be seeing spikes, but its hard to say with so many chares and migrations happening. Maybe its more likely than I think.

I also think I should be able to create a smaller toy code that's just a ping pong with migrateMes to confirm this phenomena on a smaller process count. If what you say is true, it appears there is no bug in the sense that the code is behaving exactly as intended, but our intentions are bad in some cases such as this one. Seems like it will mostly effect applications with frequent migrations and low-degree comm graphs. Not sure what the best solution is but I will think about it while working on the small example.

Was able to confirm with a small 2 chare example. Migrating chares leave behind a chain of location "pointers" in some sense, and that chain is only broken when a chare on a PE in the chain attempts to send to the chare. Even then, it only breaks one link in the chain, and breaking that link still requires a message with one hop per chain link.

Your modifications to emigrate essentially make it so that a missed message will (almost) always have three hops. One to stale location, one to home, one to new location. That is obviously better than extremely long chains, but it is also worse than the best case of 2 hops, where one hop is to stale location, and one hop to new location.
The reason why the performance gets progressively worse is not because it becomes more likely of this happening, but because the chains get longer and longer. In the very first lb steps there will already be chares migrating to a PE where a receiver was but is no longer there (at least in my test case given the high number of chares). The difference between the first lb steps and later lb steps is that the chains are longer.

I'm not sure I understand why you say 3 hops after my modifications. Wouldn't it be 2 hops in most cases? Since a message will go to the home PE first, and from there to the right location (I assume that by the time the computation resumes, and since I have syncResume on, the home PEs would already know where the chares have migrated). And if a home PE has wrong information then wouldn't it be 4 hops instead (homePe->wrong_loc->homePe->right_loc)?

EDIT: I guess you mean for anytime migration cases where messages can arrive to old location soon after the receiver has migrated, although that situation will not happen in my test case.