

总结 2021/04/26:

Machine details: <https://docs.google.com/document/d/1-eTN1eb6HogMyD-pbs-6AWb4mrFzZkkGWb1i7vGmdlU/edit?usp=sharing>

KP920/2P/6CCL-机器一 (每个DDRC一个内存条, 2999MHz)	KP920/2P/8CCL-机器二 (内存条只有满插的1/4, 每两个DDRC有一个空闲)	KP920/2P/8CCL-机器四 (机器二 增加 内存条, 每个DDRC一个内存条, 2666MHz)
<h2>CCL 内存带宽</h2>		
<p>1. 在12个并行进程及以内,当每个CCL 两个进程(占用两个core)时, 测得的内存带宽达到最高. 把进程分布到更多CCL 不能显著增加内存带宽. 比如: 4进程, 分配到 2 CCL 内存带宽最高; 分配到 3CCL/4CCL, 内存带宽没有显著增加 (<4%).</p> <p>2. 在16个并行进程及以上时,每个CCL都会至少有两个core 被占用. 此时,在保证一个进程一个core的前提下,把所有进程分配到4CCL, 5CCL,或者 6CCL时, 所获得的总内存带宽没有本质区别.</p> <p>3. 对于单进程实际获得的内存带宽, 1 NUMA 只跑 1 进程时, 单进程带宽最高. 随着进程数增多, 平均到每个进程的单进程带宽呈现增量下降趋势. 举例: (4进程内存总带宽 / 4) < 单进程带宽. 再举例: (12进程总带宽 / 2) < 6进程总带宽.</p> <p>4. 均衡点大约出现在16进程/6CCL. 即, 并行进程总带宽 在 16个进程分布到6CCL (或5CCL) 时,达到最高. 此时可以认为单 NUMA 节点的内存带宽利用率达到了最大.</p> <p>5. 多进程CCLs间的不均衡分配, 不如 均衡分配. 举例: 4进程分配到 2CCL, (2:2)分配所得内存带宽 大于 (3,1)分配. 再举例: 6进程分配到 3CCL, (2,2,2)分配所得内存带宽 大于 (4,1,1) 或者 (3,2,1)分配.</p> <p>注(2021/04/12): 2. 16进程 4CCL/ 5CCL /6CCL, stream copy 带宽依次是: 64677, 65793, 65246. 最高/最低 = 1.017, 不足 %2 的差别, 所以我说</p>	<p>1. 在这台机器上看到的数据变化与 6CCL 机器非常不同。把多进程(从 4 ~ 32 进程)密集部署(每个CCL 跑 4个进程)或者 稀松部署(每个CCL 只跑 1个进程), 测得的内存带宽基本一致。并且 stream 四种测试都是这个结果。</p> <p>* 最大的差别也就是在 4 进程时, 不同部署方式的最大/最小带宽能相差 5~9%。</p> <p>* 而并行 6~28进程, 不同部署方式的最大/最小带宽 相差 小于 1%。可忽略不计。</p> <p>2. 总带宽在 20个并行进程时达到接近最大值 (32GB/s), 之后并行进程数增加到24 , 28 , 32 , 也并没有获得太多带宽增益。最大在32进程时, 总带宽 (33GB/s)</p> <p>* 20进程/8CCL 这个并行度, 与 6CCL机器的 16进程 是大致对应的。都接近NUMA单节点满载的三分之二。</p> <p>3. 对于单进程实际获得的内存带宽, 1 NUMA 只跑 1 进程时, 单进程带宽最高. 这一点与 6CCL机器 相同。可以总结为 : 单进程带宽收益随着进程数增加而递减。</p> <p>异常数据点:</p> <p>1. 并行8进程所获得内存总带宽 高于 并行12进程(某几个 case下还高于 20进程)。期待的趋势是进程数越高带宽越高。</p>	<p>1. 同机器二相比, 各个测试结果的内存带宽结果都有显著增加。峰值32进程增加接近1倍 : 33GB/s --> 61GB/s</p> <p>2. 并行进程数等于 4, 6, 8 时, 每CCL跑2个进程能得到最优带宽。部署密度高于此限时, 内存带宽变小。</p> <p>- 但是通过改变部署密度而获得的收益看, 只有 4进程 时较大, 有 24% 的差别。</p> <p>- 6 进程 , 8 进程 , 好的部署 vs. 差的部署 , 区别只有 6-8%</p> <p>3. 并行进程数等于/大于 12 之后, 进程部署形态(密集部署 或者 稀松部署)对最终的总内存带宽影响就更小了。</p> <p>- 12进程 最好 vs. 最差 , 区别只有 3% ;</p> <p>- 16, 20 进程 最好 vs. 最差 , 区别只有 2% ;</p> <p>- 24, 28 进程 , 在 1% 以下。</p> <p>4. 并行进程数为 20 时, 能够达到最高内存带宽的 95%, 可以认为部署并行进程数超过20以上, 对于获得更高内存带宽意义不大。</p> <p>- 注:最高内存带宽出现在 32进程分布在8CCLs时, 带宽为 64.5GB/s</p> <p>- 注:并行 20进程 内存带宽能达到 60GB/s</p> <p>与 6CCL机器一 横向比较:</p> <p>1. 在核还没有用满时, 机器一 比 机器四 内存带宽要高。高出的比例与 内存工作频率 相同。机器一 用的是 2933MHz 内存, 机器四是 2666MHz, 等于(1.1 : 1.0)</p>

“没有本质区别”。

4. 如上的数据，最高，但是也是与 16进程/4CCL 差别不大。如果资源紧俏，我觉得部署到 4CCL 就够了。不过，或许需要指出的是，针对这个最大，我主要是指 STREAM copy/scale/triad 三种测试。
- 如果单点看 STREAM add 测试，最大值出现在 20进程/6CCL 时。

5. 这个我补充了测试。4进程，6进程，非均匀分配下，我都测过。不如 (2 : 2 : 2) 均匀分配。

举例：

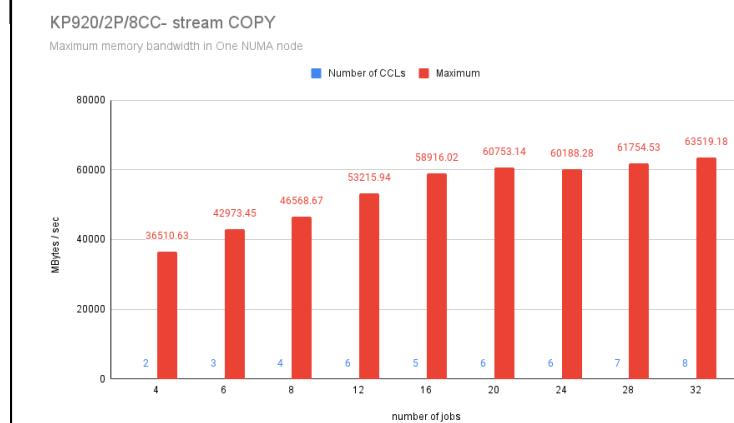
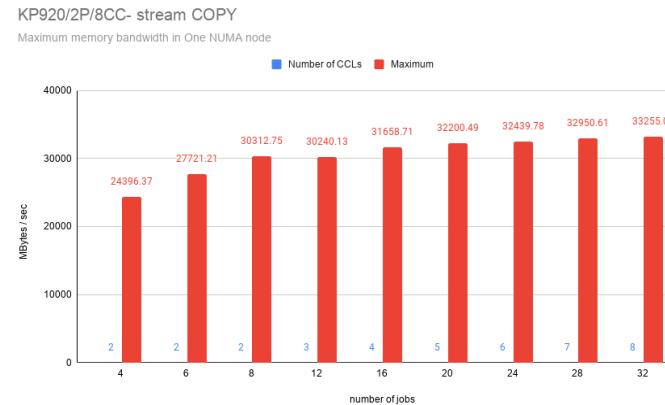
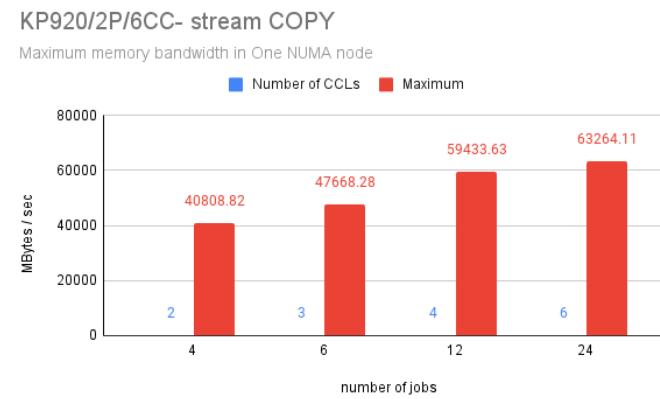
单 NUMA 并行进程数	机器一/2933Mh	机器四/2666Mh	比例
6	47 GB/s	42 GB/s	1.11
12	59 GB/s	53 GB/s	1.11
24	63 GB/s	60 GB/s	1.05
内存频率	2933 MHz	2666 MHz	1.10

2. 当核用满时，24进程/机器一（饱和）与 32进程/机器四（饱和），测得的内存带宽相等，都在 63GB/s，这时判断瓶颈点已经不在内存上了，而是 Die 内的 Ring Bus。

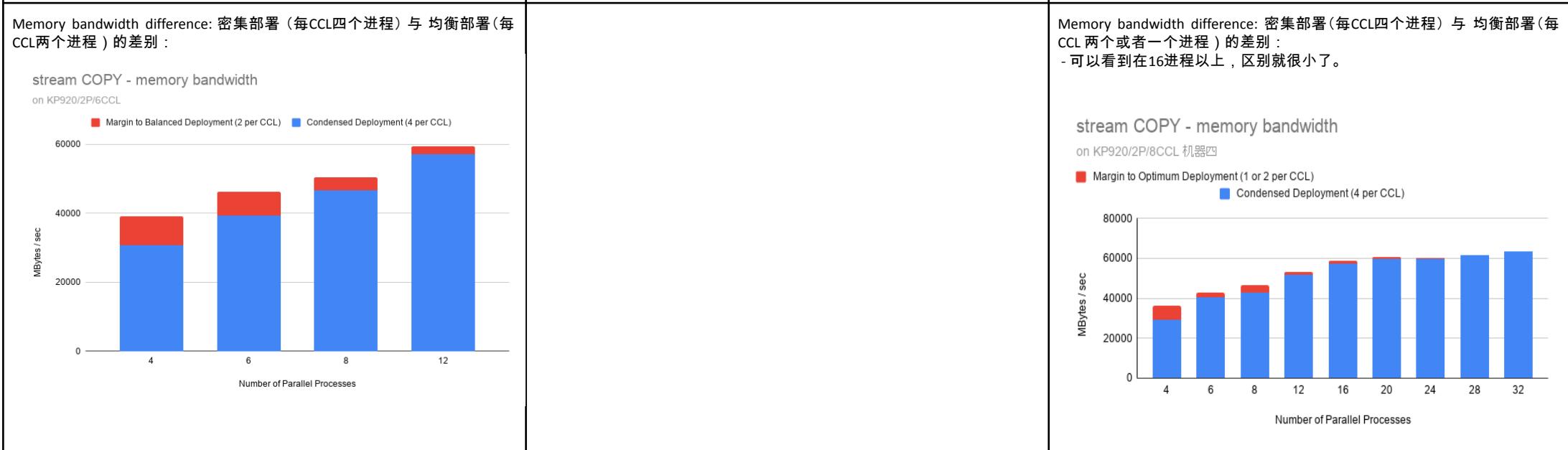
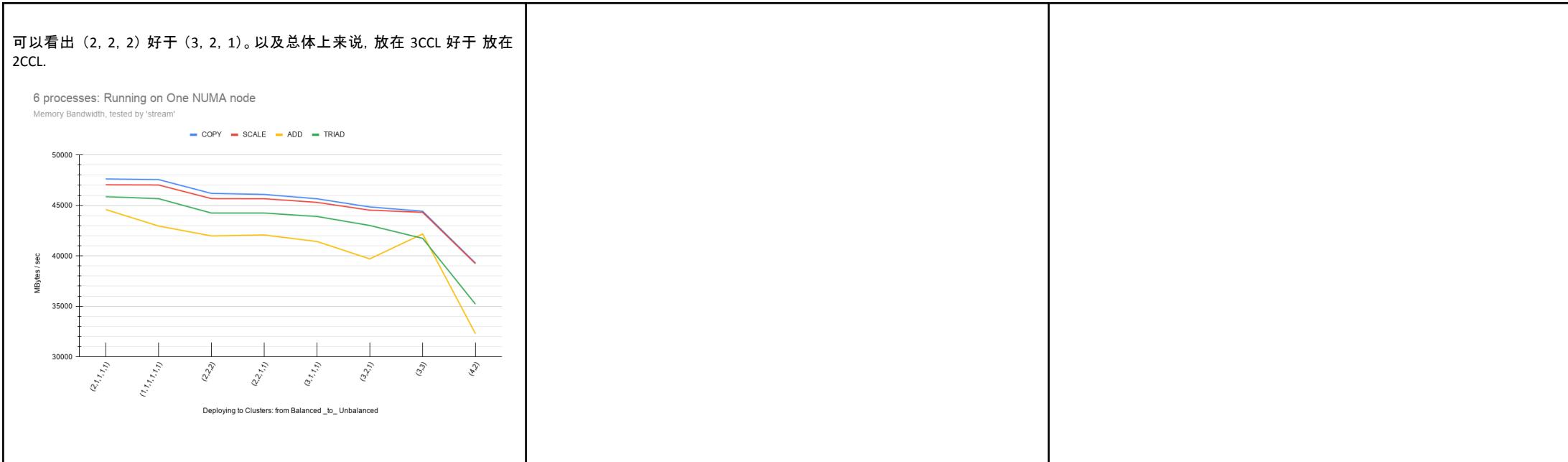
- 考虑到 6CCL机器一 上面部署的内存条都是工作在 2933MHz, 8CCL机器四 是2666MHz, 可以推断这两种情景的差别，瓶颈点在 Die 内的 ring bus，而不是内存条本身的速度了。

- 柱线1：横轴进程数，纵轴该进程数情况下，内核带宽峰值；
- 柱线2：横轴进程数，纵轴均匀分布到几个CCL后，基本再扩散开已经没作用了（或者作用已经非常不明显了），CCL数量拐点。

注：此处只统计 stream COPY

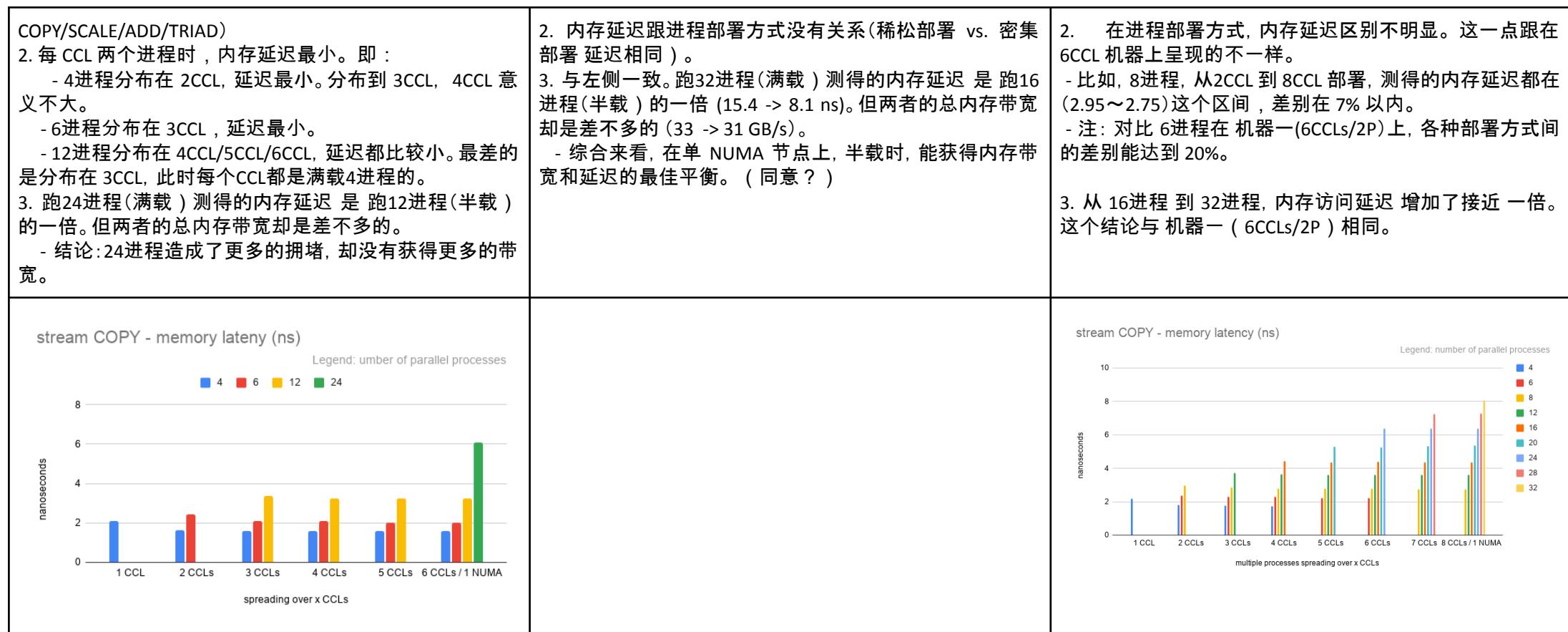


6 Jobs, running in One NUMA node, from Balanced deployment to Unbalanced deployment:



CCL 内存延迟

- | | | |
|---------------------------|-------------|-------------------------|
| 1. stream 四种测试所呈现的趋势是一致的。 | (1. 与左侧相同。 | 1. stream 四种测试所呈现的趋势一致。 |
|---------------------------|-------------|-------------------------|



跨NUMA 内存带宽

<p>1. 彼此不存在内存重叠的并行进程，在跨NUMA 运行时，所测得的 总内存带宽 与单NUMA 相比，呈等比例增长，符合预期。比如：8进程 均衡分布在两个 NUMA 的 总内存带宽 = 4进程 均衡分布在一个 NUMA 的 总内存带宽 * 2. 再比如：24进程 均衡分布在四个 NUMA 的 总内存带宽 = 6进程 均衡分布在一个 NUMA 的 总内存带宽 * 4.</p> <p>2. 均匀部署 优于 不均匀部署。举例：8进程部署到 三个 NUMA, (3,3,2) 优于 (4,2,2)</p> <p>3. 综合结果，在均匀部署的前提下，从 4进程 到 72进程，分布到 1NUMA / 2NUMA / 3NUMA / 4NUMA，都呈现出 总内存带宽 线性增长的趋势，4NUMA 最高。比如：24进程，从 1NUMA 到 4NUMA，对应的内存带宽依次为 (63, 117, 149, 181).</p>	<p>1. 与左侧一致。</p> <p>2. 参照“CCL 内存带宽”，部署方式在这台 8CCL 机器上不重要。</p> <p>3. 与左侧一致。</p> <p>异常点：</p> <p>1. 16进程分布于 2 NUMA 和 3 NUMA 时，测得的总带宽大于20进程。期待的情况是 20进程总带宽更大。</p>	<p>1. 与机器一(6CCLs) 1. 2. 3. 规律都一致。</p> <p>2. 异常：测试带宽在 16进程 和 20进程 分布于 3NUMA 节点时出现“倒挂”(如图)，20进程测得的内存带宽更低了。</p>
--	--	--



跨NUMA 内存延迟

- 当我们把多进程分散到多NUMA节点时，内存访问延迟按照预期，呈现了逐步下降趋势。
- 这个趋势在12进程和24进程时，最为明显。举例：有12个进程时，放在一个NUMA上，内存延迟是3ns；放到四个NUMA时，内存延迟就能下降到1.5ns。
- 4进程时，分布到1NUMA 和 4NUMA 区别并不大。虽然4NUMA更快，但是也仅有20%左右的缩短。

- 趋势性比内存带宽的好，没有数据异常点。
- 每个NUMA节点的进程数越多，那么其内存延迟越大。
- 同样数量的进程，分散部署在4 NUMA 的延迟，到3 NUMA，到2 NUMA，到1 NUMA 的延迟，呈现递增趋势。

- 结论内容与6CCL-机器一相同。
- 横向比较，6CCL 机器一更快一些。比如：
- 12进程分布到4NUMA，机器一延迟1.51ns，机器四延迟1.72ns

NUMA0 CPU 分别访问 NUMA0/1/2/3 的内存带宽 对比

- 在这个测试里面，测试结果显示出：跨NUMA访问内存时，所获得的内存带宽变化规律，与实际计算类型/访问模式强相关。目前无法简单的总结。
比如：stream COPY/ SCALE/ ADD，这三个测试所呈现的规律基本一致。而 stream TRIAD 测试所呈现的趋势有时相反。具体见下面描述。
- 计算和内存位于不同package的NUMA节点时(如NUMA0 和 NUMA2)

- 跟左侧相比，同package，有更多的membind=1的带宽高于membind=0：从4进程并行到16进程，所有四种stream测试(COPY/SCALE/ADD/TRIAD)无一例外。
-甚至单进程时，stream ADD 和 stream TRIAD，也有这种现象。可重复。
- 有很多情景，内存放在不同 package 时，得到的内存带宽高于内存放在本地。membind=2, 3 大于 membind=0 的情景。

- 当讨论并行4 jobs stream ADD/TRIAD, 6 jobs ~ 16 jobs的所有四种stream测试，都是最大内存带宽出现在内存绑定在同package的不同NUMA node时(比如计算在NUMA0 --> 内存在NUMA1)。
- 计算和内存位于不同package的NUMA节点时(如NUMA0 和 NUMA2)，内存带宽会普遍小于计算和内存位于同一package的NUMA节点的情况。这符合预期，显示出同package的NUMA节点之间有更高的内存带宽。

<p>- 所有的四种 stream 测试都呈现出内存带宽下降的情形。但是，</p> <p>- 但是，所测得的下降比例有很大不同。对于 stream COPY/SCALE/ TRIAD，带宽下降比例在 30%左右。而 stream ADD 带宽下降比例高达 60%。</p> <p>5. 多进程在计算节点在 NUMA0 上的分布形态(比如4进程部署于 2CCL, 3CCL, 或者4CCL)，对最终结果影响不大。</p> <p>6. 在多次测试中，发现同一测试用例，有数据不稳定的情况。选择暂时忽略这些异常数据点。比如 <code>cpu(0, 1, 4, 8) membind 1 ADD bandwidth.</code> 这从另一个侧面反映出这个测试的动态随机性。</p> <p>注(2021/04/19)： 针对第 3. 条，在多进程对比后，发现应该更正如下。 针对第 5. 条，更正为“跨numa访问内存的带宽，...和不跨NUMA访问本NUMA内存的CCL带宽特征相似”。 在改进的测试用例中，我兼顾从 1 job 到 16 jobs 不同的任务负载。期望整理出跟普适的规律。总结如下：</p> <ol style="list-style-type: none"> 对于并行 1 Job 和 4 jobs, stream COPY/ SCALE/ ADD 这三种任务的最大带宽，出现在内存绑定在相同 NUMA node 时。次之的，是内存绑定在同package的不同 NUMA node 时（比如 计算在 NUMA0 --> 内存在 NUMA1）。 <ul style="list-style-type: none"> 而对于运算更复杂些的 stream TRIAD，平行 1 job 和 4jobs 的最大内存带宽，一律是出现在 内存绑定在同 package 的不同 NUMA node 时（比如 计算在 NUMA0 --> 内存在 NUMA1）。 当讨论并行 6 jobs ~ 16 jobs 时，更普遍的情况是无论哪种 stream 任务，都是 最大内存带宽 出现在 内存绑定在同 package 的不同 NUMA node 时（比如 计算在 NUMA0 --> 内存在 NUMA1）。 <ul style="list-style-type: none"> 有少数情况例外。比如 16进程 stream ADD 和 TRIAD 时，最大带宽在本地内存。 计算和内存 位于不同package 的 NUMA 节点时（如 NUMA0 和 NUMA2），内存带宽会普遍小于 计算和内存 位于同一package 的 NUMA 节点的情况。这符合预期，显示出同package的NUMA节点之间有更高的内存带宽。 在 计算和内存 都位于 同一NUMA节点 时总结出来的内存带宽规律，在 计算和内存位于不同NUMA节点时，多数情况下也是适用的。比如： <ul style="list-style-type: none"> 12进程运行在 3CCL 时内存带宽最小，6CCL 时内存带宽最大； 当把内存绑定到不同NUMA节点时，这个规律依然成立。 	<p>- 包括: stream COPY/SCALE/ADD/SCALE: 6, 8, 12, 16 进程。</p> <p>3. 4 进程并行时, membind=2, 3 跟 membind=0 相比，内存带宽差不多（稍微高/稍微低都有，但是差别不大）。本地存并没有优势。</p> <p>4. 单进程时，</p> <ul style="list-style-type: none"> stream COPY/SCALE: membind 0 > 1 > 2 > 3. 符合预期。 stream ADD/TRIAD: membind 1 > 0 > 2 > 3. (1 > 0) 是不符合预期的。 <p>5. 虽然在 CCL 测试中（如上面“异常点”描述），8进程 stream SCALE 带宽 高于 12进程 的，但是 <ul style="list-style-type: none"> 当把内存节点放到 membind=1, 2, 或3 时，测得的内存带宽又变回来了：8进程 < 12进程，符合预期。 </p>	<p>3. 对于计算在 NUMA0, 内存在 NUMA2 通常都比 内存在 NUMA3 带宽更大。</p> <p>4. 总体测试结论跟 6CCL-机器一一致。</p>
---	--	---

- 再比如，4进程 stream COPY, 分布到 2CCL, 3CCL, 4CCL 所获得的总带宽相差无几；当把内存节点绑定到不同 NUMA 时，这个规律依然成立。

5. 测试中，发现有些单点的异常情况(可重复)。这些在整理的Excel表格中标注为红色。目前不做进一步分析。比如，

- 8进程分布在5CCL, 内存绑定到同package相邻NUMA node的情况，所有四种stream测试都表现出了严重低于4CCL 和 6CCL 的带宽结果。(expected, 应该是 5CCL 与4/6CCL 差不多高。)
- stream TRIAD 测试中：4进程分布在2CCL, memnode = 3 时，带宽严重 ^ 高于 ^ 4进程分布在 3CCL 和4CCL 时。
- 在 stream TRIAD 测试中，我发现了更多的 membind 异置时的异常数据点。其中两个与 membind=2 有关。(同样的测试，和 membind =3 结果差异巨大)。

NUMA0 CPU 分别访问 NUMA0/1/2/3 的 NUMA 节点间内存延迟

这里的描述是基于 4进程并行的 stream 测试。

1. 在四种 stream 测试中，结果不一致。有的跟 ACPI SLIT table 中保存的 NUMA distance 相同，有的差别很大。这里的结论应该是要根据业务类型结合起来判断，是内存 bounded, 还是计算 bounded。
2. 具体说，从 NUMA0 到 其余NUMA 节点(NUMA0, 1, 2, 3)，延迟分别是：
 - COPY/SCALE : 10, 11, 13, 14 (Note: 体现单纯的 内存 bounded 的极限)
 - ADD : 10, 12, 22, 23 (Note: 与 ACPI NUMA distance 一致)
 - TRIAD : 10, 8 , 12, 13 (Note: 这个测试结果，为什么邻居节点 反倒比 本地节点 (8 vs. 10) 还快？需要解释)
3. 如上，为什么 TRIAD 时 NUMA0 访问 NUMA1 比访问本地 NUMA 还快？
4. 前提：因为指定了 membind=0 (或1, 2, 3)，我认为没有发生数据迁移的情况。

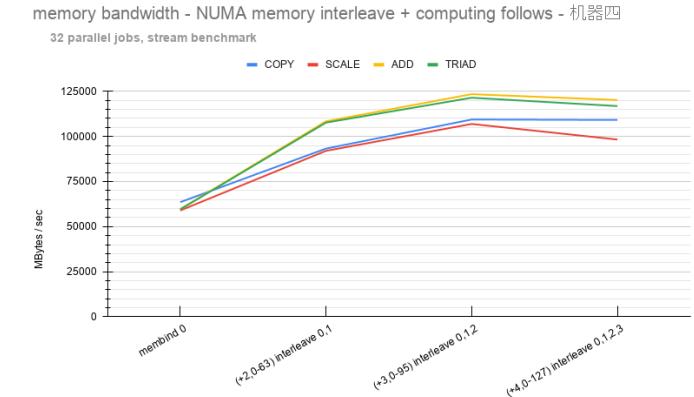
1. 出现大多数情况下，membind=1 的 latency 最小。
 - 各NUMA节点的 内存延迟 总体排名(最快 到 最慢)：
 membind=1, 2, 3, 0

这里有很大的问题，为什么本地节点 (membind=0) 的内存延迟最高？

测试包括从 4jobs 到 16jobs 的情况。总结：

1. 计算绑定在 NUMA0, 内存绑定节点变化。大多数测试中，membind=1 的 latency 最小，其次是 membind=0
 - 各NUMA节点的 内存延迟 总体排名: membind=3 > 2 > 0 > 1. (节点 3 最慢，节点1最快)
2. 上述结论在 4 jobs 到 16jobs 都成立。可以据此整理规律矩阵。

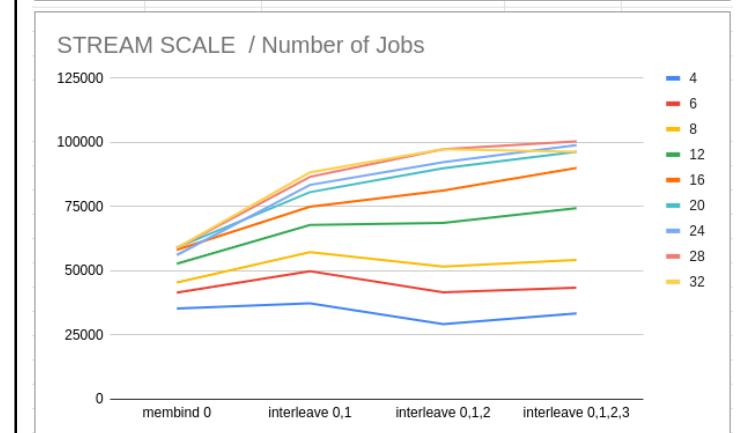
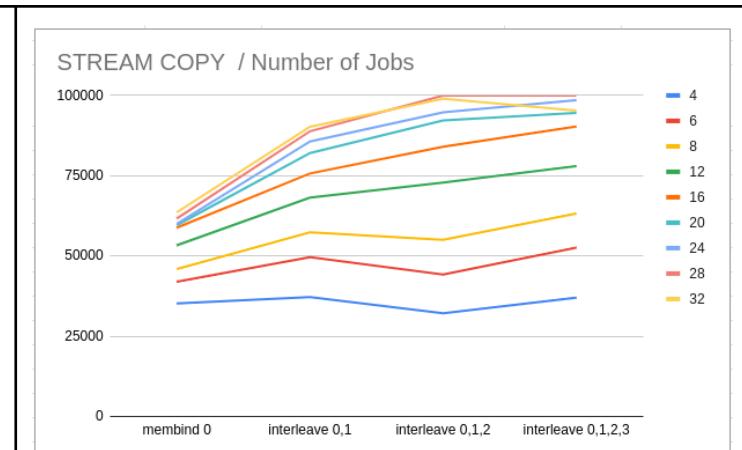
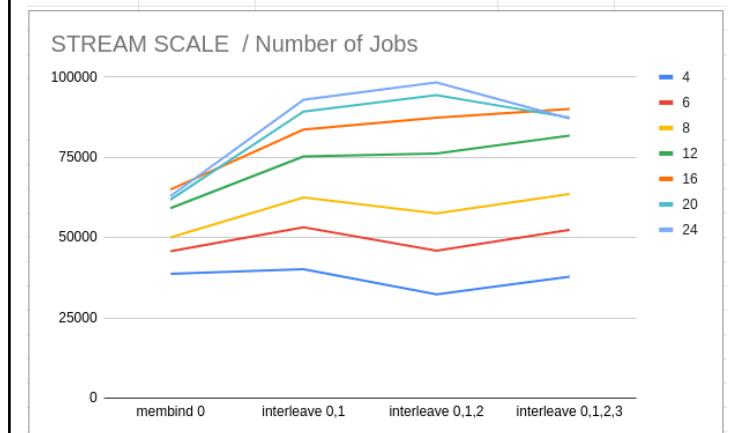
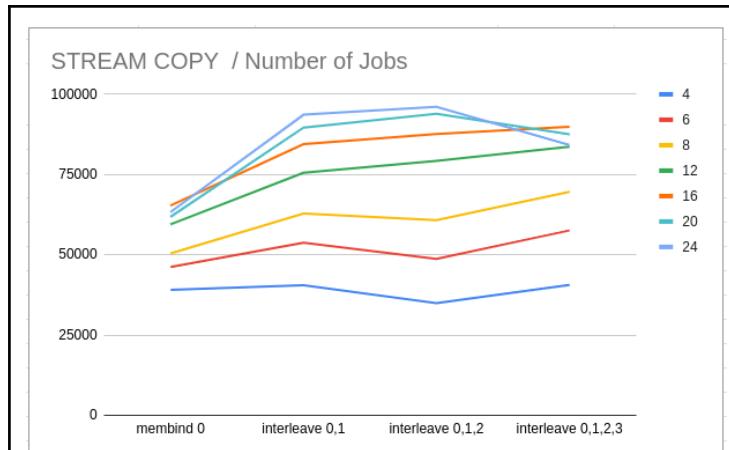
NUMA interleave 的内存带宽，计算节点跟随

<p>1. 背景:这个测试中的内存在哪几个NUMA节点间interleave, cpu计算也被部署到同样的几个NUMA节点中。24进程并行。</p> <p>2. 在所有cpu中依次 +2, +3, +4 间隔部署, 所产生的效果与按照每CCL中使用两个 core 的效果相同, 没有区别。都属于有效的均衡部署。这跟前述结论一致。</p> <p>3. 趋势:总内存带宽在 interleave 到 3个NUMA时, 达到最大。再往下, 依次是:3 NUMA > 4 NUMA > 2 NUMA > 1 NUMA (其中 ADD 在 2 NUMA > 4 NUMA)</p> <ul style="list-style-type: none"> - interleave, 从 1NUMA 到 2NUMA, 总内存带宽增涨 +50% <p>从 2NUMA 到 3NUMA, 总内存带宽增涨 +15%, 收效就如跨 2NUMA 时明显了。</p> <p>从 3NUMA 到 4NUMA, 总内存带宽又回落到跟 2NUMA 差不多的水平。</p> <p>4. 把同样的24进程分布部署到 多NUMA节点, 内存节点interleave 比 不做interleave, 总内存带宽要差。具体说 :</p> <ul style="list-style-type: none"> - 2NUMA interleave, 损失 -17% - 3NUMA interleave, 损失 -25% - 4NUMA interleave, 损失 -40% 	<p>1. 背景:这个测试中的内存在哪几个NUMA节点间interleave, cpu计算也被部署到同样的几个NUMA节点中。32进程并行。</p> <p>2. 趋势:与左侧不同。interleave 到 越多NUMA节点, 总内存带宽越大。4 NUMA > 3 NUMA > 2 NUMA > 1 NUMA. 单调上升。</p> <ul style="list-style-type: none"> - 分散到 4 NUMA时 , 内存带宽是 1 NUMA 的3.0倍。 <p>总结: 这台机器的 interleave 内存带宽收益, 远高于左边的6CCL 机器。(左侧的 6CCL 在最高时 interleave 的内存带宽是 1NUMA 的 2.0倍)</p>	<p>1. 背景:使用的是 32进程并行。这个测试中的内存在哪几个NUMA节点间 interleave, cpu计算也被部署到同样的几个NUMA节点中。</p> <p>2. 所获得的测试结论同 6CCL 机器一 相同。可参考下图。</p>  <p>The graph shows memory bandwidth in MB/sec on the y-axis (0 to 125000) versus interleave levels on the x-axis. Four series are plotted: COPY (blue), SCALE (red), ADD (yellow), and TRIAD (green). All series show an upward trend as the number of NUMA nodes increases from 0 to 3. The ADD and TRIAD series reach the highest bandwidth values, around 120,000 MB/sec at 4 NUMA nodes.</p> <table border="1"> <thead> <tr> <th>Interleave Level</th> <th>COPY (MB/sec)</th> <th>SCALE (MB/sec)</th> <th>ADD (MB/sec)</th> <th>TRIAD (MB/sec)</th> </tr> </thead> <tbody> <tr> <td>membind 0</td> <td>~60000</td> <td>~60000</td> <td>~60000</td> <td>~60000</td> </tr> <tr> <td>(+2,0,3) interleave 0,1</td> <td>~80000</td> <td>~75000</td> <td>~85000</td> <td>~80000</td> </tr> <tr> <td>(+3,0,9) interleave 0,1,2</td> <td>~100000</td> <td>~95000</td> <td>~110000</td> <td>~105000</td> </tr> <tr> <td>(+4,0,12) interleave 0,1,2,3</td> <td>~110000</td> <td>~105000</td> <td>~120000</td> <td>~115000</td> </tr> </tbody> </table>	Interleave Level	COPY (MB/sec)	SCALE (MB/sec)	ADD (MB/sec)	TRIAD (MB/sec)	membind 0	~60000	~60000	~60000	~60000	(+2,0,3) interleave 0,1	~80000	~75000	~85000	~80000	(+3,0,9) interleave 0,1,2	~100000	~95000	~110000	~105000	(+4,0,12) interleave 0,1,2,3	~110000	~105000	~120000	~115000
Interleave Level	COPY (MB/sec)	SCALE (MB/sec)	ADD (MB/sec)	TRIAD (MB/sec)																							
membind 0	~60000	~60000	~60000	~60000																							
(+2,0,3) interleave 0,1	~80000	~75000	~85000	~80000																							
(+3,0,9) interleave 0,1,2	~100000	~95000	~110000	~105000																							
(+4,0,12) interleave 0,1,2,3	~110000	~105000	~120000	~115000																							

NUMA interleave 的内存延迟 , 计算节点跟随

<p>1. 这个测试中, 采用了 24进程。这个进程个数对于 NUMA0 来说, 是饱和的。所以, 当 interleave 到 两NUMA 节点时, 按照期待, 会出现压力减轻, 延迟变小的情况。实际测试符合这个预期。并且, 在 stream 四种测试结果所呈现的趋势, 是彼此 一致 的。显示的都是如果使用 interleave 时, 访问延迟 与 业务类型大概率是无关的。</p> <p>2. 在 interleave 到 三个节点 NUMA(0, 1, 2)时, 内存延迟 最小。</p> <p>3. 在 interleave 到 两节点 和 四节点 , 内存延迟差不多大。</p> <p>接下来, 采用 12进程 , 看看 interleave 是否仍然有利 :</p> <p>4. 呈现出与 24进程不同的趋势。</p> <p>5. 对于 COPY/SCALE/TRIAD 三种测试, interleave 到 两个 / 三个 / 四个 NUMA 节点后, 内存延迟都比只在一个 NUMA 节点时缩短了。并且interleave到 2/3/4个NUMA节点的延迟</p>	<p>1. 这个测试中, 采用了 32进程。这个进程个数对于 NUMA0 来说, 是饱和的。</p> <p>2. 趋势:与左侧不同。interleave 到 越多 NUMA 节点, 内存延迟 就越低 , 单调下降。</p>	<p>1. 这个测试中, 采用了 32进程。这个进程个数对于 NUMA0 来说, 是饱和的。所以, 当 interleave 到 两NUMA 节点时, 按照期待, 会出现压力减轻, 延迟变小的情况。实际测试符合这个预期。并且, 在 stream 四种测试结果所呈现的趋势, 是彼此 一致 的。显示的都是如果使用 interleave 时, 访问延迟 与 业务类型大概率是无关的。</p> <p>2. 在 interleave 到 三个节点 NUMA(0, 1, 2)时, 内存延迟 最小。</p> <p>3. 在 interleave 到 两节点 和 四节点 , 内存延迟差不多大。</p> <p>综上所述, interleave 到两个 NUMA 节点, 所取得的 内存带宽 和 内存延迟 最优。</p>
---	---	---

<p>都差不多大，4节点延迟最小。</p> <p>6. 而对于 ADD 这个测试，看到的趋势有所不同。interleave 到两个 NUMA 时延迟最小。4NUMA 和 1NUMA 延迟一样，最大。3NUMA 居中。</p>		
<h2>NUMA interleave 的内存带宽，计算节点固定 NUMA0</h2>		
<p>1. 背景：测试从 4进程 到 24进程 的情景。所有进程固定在 NUMA0. 尽可能的半满部署 (2 jobs/CCL)。</p> <p>2. 所有情景中，interleave 到同 package 的两个 NUMA 节点 (同 package 的，0 和 1) 所测得的内存带宽都是大于只使用本地 NUMA 节点的。</p> <ul style="list-style-type: none"> - 并行进程数越多，分布到两 NUMA 的好处越明显。具体说：满载 24 进程时，interleave 到两 NUMA 能得到 +50% 的内存带宽；4 进程时，只能得到 +3% 的提高。 <p>3. 然而，在 interleave 到三个 NUMA 节点时，内存带宽普遍比 interleave 到两 NUMA 低。</p> <p>4. interleave 到四个 NUMA 节点时，依赖于 benchmark 类型，有的比两节点小（比如 stream ADD），有的比两 NUMA 高（比如 stream COPY）</p> <p>总结：interleave 到 两节点 最优。interleave 到 三节点 变差。interleave 到 四节点 有高有低。</p>	<p>1. 背景：测试从 4进程 到 32进程 的情景。所有进程固定在 NUMA0. 尽可能的半满部署 (2 jobs/CCL)。</p> <p>2. 所有情景中，interleave 到同 package 的两个 NUMA 节点 (0 和 1) 所测得的内存带宽都是大于只使用本地 NUMA 节点的。</p> <p>3. 并行进程数小于等于 8 时，interleave 到 三节点 (0, 1, 2) 和 四节点 (0, 1, 2, 3) 内存带宽跟 两节点 (0, 1) 提升不大。高复杂性的任务 stream TRIAD 时 三/四节点 还会下降。</p> <p>4. 并行进程数为 12 及以上时，interleave 到 的节点数量越多内存带宽越大：四节点 (0, 1, 2, 3) > 三节点 (0, 1, 2) > 两节点 (0, 1) > 单节点 (0)。</p>	<p>1. 背景：测试从 4进程 到 32进程 的情景。所有进程固定在 NUMA0. 尽可能的半满部署 (2 jobs/CCL)。</p> <p>2. 所有情景中，interleave 到同 package 的两个 NUMA 节点 (同 package 的，0 和 1) 所测得的内存带宽都是大于只使用本地 NUMA 节点的。</p> <ul style="list-style-type: none"> - 并行进程数越多，分布到两 NUMA 的好处越明显。具体说：满载 32 进程时，interleave 到两 NUMA 能得到 +50% ~ +80% 的内存带宽提高；4 进程时，只能得到 +5% ~ 18% 的提高。 <p>- 注：业务越复杂（比如 stream ADD，TRIAD），intrelave 得到的提升也越高。</p> <p>3. interleave 到三节点时，并行进程数 大于等于 16，还是能得到 内存带宽提升的。并行进程数 小于等于 12，测得的内存带宽就会比两 NUMA 小。</p> <p>4. interleave 到四节点，收效不大，复杂业务时不如两节点。</p> <p>总结：interleave 到 两节点 最优。并行进程数 大于等于 16 时，可以考虑 interleave 到 三节点 获得更多带宽。interleave 到 四节点 就不推荐了。</p> <ul style="list-style-type: none"> - 进程数多时，interleave 能够充分利用其他 NUMA 节点的内存控制器，从而提高内存存取效率。 - 但是如果 interleave 的节点太多，那么就会受制于 Die 间的 hydra bus 互联瓶颈。





NUMA interleave 的内存延迟，计算节点固定 NUMA0

1. 背景: 同上。
2. 结论与内存带宽类似, interleave 到 两 NUMA 节点时的内存延迟最小。interleave 到 三或者四 NUMA 节点时 内存延迟都比 两NUMA 大了。

1. 背景: 同上。
2. 几乎所有测试结果都符合预期的趋势 , 单调增或减 :
- 进程数越多 , 内存延迟越大。
- interleave 到的节点数越多 , 内存延迟越小。

异常点 :
3. 当 4 或 6 进程 interleave 到三节点 (0, 1, 2) 时, 测试中有几处异常点 , 偏离上述2中的趋势。但差别并不大。

1. 背景: 同上。
2. 结论 可以从内存带宽反推。interleave 到 两 NUMA 节点时的 内存延迟最小。并行进程数越多, 这种内存延迟缩小的越大。

pipe延迟 , CCL内/间, 跨NUMA

1. 在相同 CCL 内, pipe延迟 最小。从 NUMA0 到 NUMA3, pipe延迟最大
总结 : same CCL < cross CCL < NUMA 1 < NUMA 2 < NUMA3

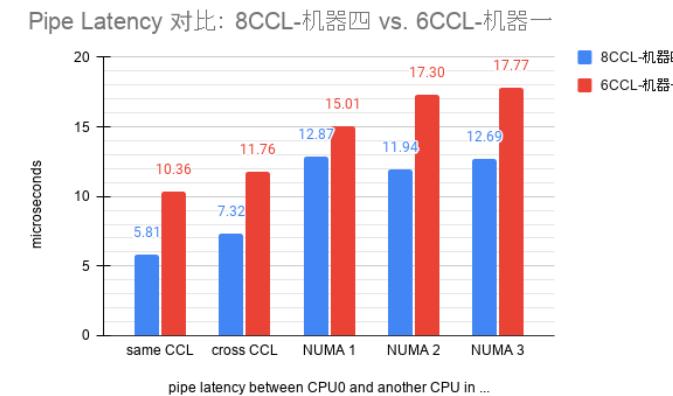
1. 在相同 CCL 内, pipe延迟 最小。NUMA1, pipe延迟最大, 不符合预期。

总结 : same CCL < cross CCL < NUMA 2 < NUMA 3 < NUMA1

1. 背景:测试的是两个进程通过 pipe 进行通信的延迟。
2. 结论是符合预期的。(延迟最小) same CCL < cross CCL < NUMA 1 < NUMA 2 < NUMA 3 (延迟最大)

3. 有意思的是, 在 6CCL-机器一 和 8CCL-机器四 的绝对数值比较。会发现在各个场景下, 8CCL-机器四 的 pipe 延迟都更小。

这个结果的不合理指出在于 机器一 使用的内存是更快的, 看来 pipe latency 的延迟不在于内存速度? 那是哪里呢?



lock延迟 , pthread spinlock/mutex , CCL , NUMA

1. 背景:采用4线程互锁测试, (4/27) 增加了 8线程 的测试用例。
2. 在同一NUMA节点内, 4线程分布在越多的 CCL, 则最终的 pthread spinlock 和 mutex 延迟越大。8线程也同样的规律。
3. 对于所有测试情形, 都有 pthread spinlock 延迟 小于

1. 背景:采用 4线程 和 8线程 互锁测试
2. 在同一NUMA节点内, 4 线程分布在越多的 CCL, 则最终的 pthread spinlock 和 mutex 延迟越大。8 线程也同样规律。
3. 8线程分布在两个 NUMA 节点时, NUMA 0, 1 之间的 lock 延迟最短。pthread pinlock 和 mutex 同样规律。

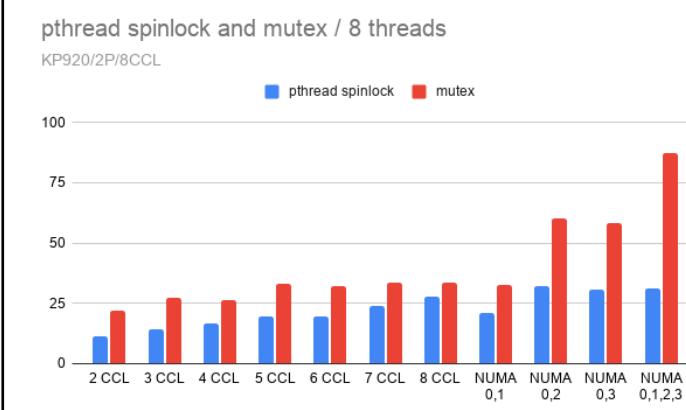
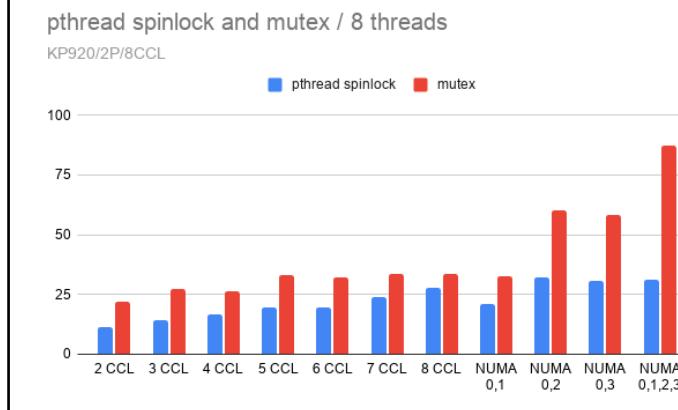
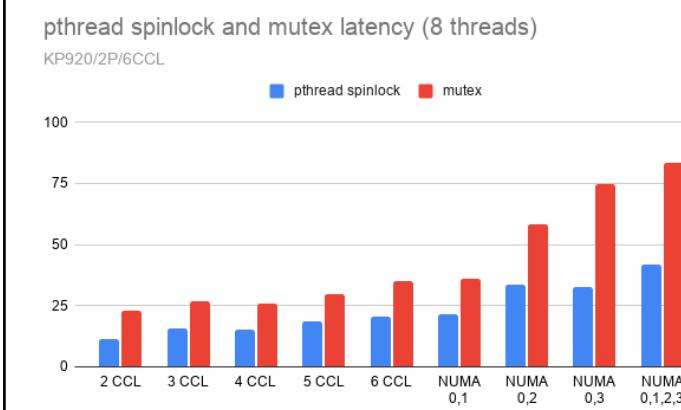
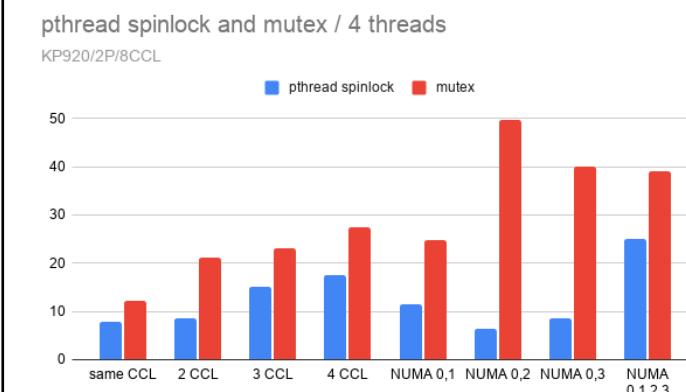
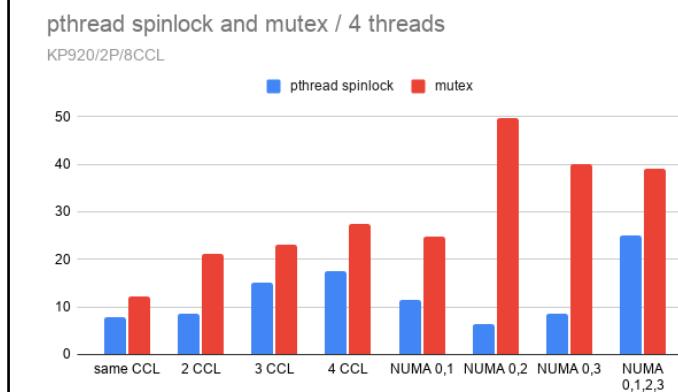
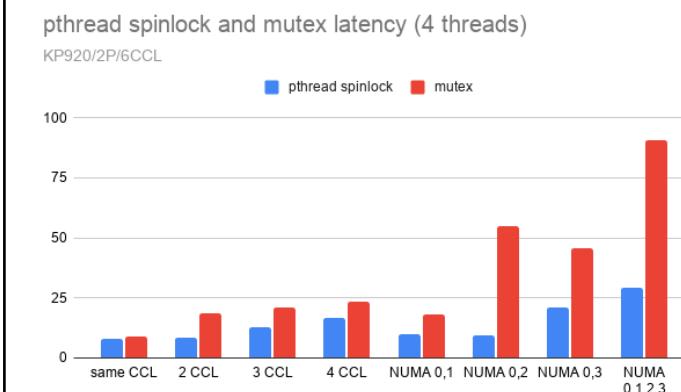
1. 背景:采用4线程互锁测试, (4/27) 增加了 8线程 的测试用例。
2. 结论 :与 6CCL-机器一 相同。
a) pthread spinlock 比 mutex 快。
b) 位于同一个 CCL 比 位于不同 CCL 快。
c) 位于同一个 package 比 位于不同 package 快。

mutex 延迟。

4. 当需要跨两个 NUMA 节点时, 无论 4进程 还是 8进程, 无论 pthread spinlock 还是 mutex, 都是分布在 NUMA 0, 1 时延迟最小。(比分布在 NUMA 0, 2 或者 NUMA 0, 3 要小)。

5. 所有情景中, 跨 四个 NUMA节点的线程间互锁, 都是延迟最大的。符合预期。

4. 4线程分布在两个 NUMA 节点时, pthread pinlock 延迟在 NUMA 0, 2在之间最短; mutex 延迟在 NUMA 0, 1之间最短。



代码交付

- 相关测试脚本, 以及测试数据 <https://gitee.com/docularxu/wayca-deployer/tree/working-kp920-6ccl-2p-benchmarking/> (包含6CCL 和 8CCL)
- 6CCL-机器一: kp920.2P.6CCL/log.*

```
- 8CCL-机器二: kp920.2P.8CCL.type2/log.*  
- 8CCL-机器四: kp920.2P.8CCL.type4/log.*
```

2. 测试工具-lmbench，原始代码来源于 <https://jaist.dl.sourceforge.net/project/lmbench/development/lmbench-3.0-a9/lmbench-3.0-a9.tgz>
有单点修改一个绑定CPU的bug，上传在 github:[URL](#) 和 OpenEuler-lmbench: [URL](#).

```
- Build: cd src; make clean; make results;
```

3.lock-test, 测试代码有修改，上传在 <https://github.com/docularxu/benchmarks/tree/working-timing>

```
- Build: cd lock; gcc -pthread lock_test.c -o lock_test;
```