CIEL：用于分布式数据流计算的通用执行引擎

简介：

本文介绍了 CIEL，一种用于分布式数据流程序的通用执行引擎。与以前的执行引擎一样，CIEL 掩盖了分布式编程的复杂性。与那些系统不同，CIEL 作业可以做出与**数据相关**的控制流决策，这使它能够计算迭代和递归算法。
我们还开发了 Skywriting，一种直接在 CIEL 上运行的图灵完整脚本语言。执行引擎为 Skywriting 脚本和用其他编程语言编写的高性能代码提供透明的容错和分发。我们在云计算平台上部署了 CIEL，并证明它可以实现迭代和非迭代算法的可扩展性能。

1 介绍

许多集群越来越需要处理大型数据集。分布式执行引擎（如 MapReduce 和 Dryad）已成为此类集群的流行系统。这些系统公开了一个简单的编程模型，并可以自动处理分布式计算的难点：容错，调度，同步和通信。MapReduce 和 Dryad 可用于实现大多数算法，但对于迭代算法则显得低效。迭代算法构成了许多机器学习和优化问题的基础，但需要更具表现力的编程模型和更强大的执行引擎。为了解决这些限制，并将分布式执行引擎的优势扩展到更广泛的应用程序，我们开发了 Skywriting 和 CIEL。

Skywriting 是一种脚本语言，允许使用命令式和函数式语言语法直接表达迭代和递归任务并行算法。Skywriting 脚本在 CIEL 上运行，CIEL 是一个执行引擎，为分布式数据流提供通用执行模型。与先前的系统一样，CIEL 协调根据数据流 DAG 划分的一组任务（数据并行）的分布式执行，并受益于透明扩展和容错。不同的是，CIEL 通过在**执行任务时动态构建 DAG 来扩展以前的模型**。正如我们将要展示的那样，允许任务创建更多任务，使 CIEL 能够支持依赖于数据的迭代或递归算法。我们在第 3 节介绍了 CIEL 的高级架构，并解释了 Skywriting 如何在第 4 节中映射到 CIEL 的基元。

我们的实现包含了第 5 节中描述的几个附加功能。与现有系统一样，CIEL 为 worker 节点提供透明的容错功能。此外，CIEL 可以容忍集群中主机和客户机程序间的故障。为了提高资源利用率并减少执行延迟，CIEL 可以记录任务结果。最后，CIEL 支持在同时执行的任务之间传输数据。

我们在 Skywriting 中实现了各种应用程序，包括 MapReduce 风格(grep，wordcount)，迭代(k-means，PageRank)和动态编程(Smith-Waterman，最优化代价)算法。在第 6 节中，我们在 CIEL 集群上运行时评估其中一些应用程序的性能。

2 动机

一些研究人员已经确定了 MapReduce 和 Dryad 编程模型的局限性。这些系统最初是为面向批处理的工作开发的，即用于信息检索的大规模文本挖掘。它们**旨在最大限度地提高吞吐量**，而不是最大限度地减少单个作业延迟。这在迭代计算中尤其明显，其中多个作业被链接在一起并且作业等待时间成倍增加。

尽管如此，MapReduce – 特别是它的开源实现 Hadoop 仍然是一个流行的平台，用于大输入的并行迭代计算。例如，Apache Mahout 机器学习库使用 Hadoop 作为其执行引擎。几种 Mahout 算法 – 例如 k 均值聚类和奇异值分解 – 是迭代的，包括在未收敛的循环内的数据并行内核。Mahout 使用一个驱动程序，向 Hadoop 提交多个作业，并在客户端执行收敛测试。但是，由于驱动程序在 Hadoop 集群外部以逻辑方式（通常是物理上）执行，因此每次迭代都会导致作业提交开销，并且驱动程序无法从透明容错中受益。这些问题并非 Hadoop 独有，而是 MapReduce 和 Dryad 原始版本都有的问题。

Feature	MapReduce [2, 18]	Dryad [26]	Pregel [28]	Iterative MR [12, 21]	Piccolo [34]	CIEL
Dynamic control flow	✗	✗	✓	✓	✓	✓
Task dependencies	Fixed (2-stage)	Fixed (DAG)	Fixed (BSP)	Fixed (2-stage)	Fixed (1-stage)	Dynamic
Fault tolerance	Transparent	Transparent	Transparent	✗	Checkpoint	Transparent
Data locality	✓	✓	✓	✓	✓	✓
Transparent scaling	✓	✓	✓	✓	✗	✓

Figure 1:现有分布式执行引擎提供的功能分析。

分布式执行引擎的计算能力由它可以表达的数据流决定。在 MapReduce 中，数据流限于由 map 和 reduce 任务数量参数化的二分图; Dryad 允许数据流遵循更通用的有向无环图(DAG)，但必须在开始作业之前完全指定。通常，为了支持单个作业中的迭代或递归算法，我们需要**依赖于数据的控制流，即基于先前计算的结果，动态创建更多工作的能力**。同时，我们希望保留任务级并行性的现有好处：透明容错，基于局部性的调度和透明扩展。在 Figure 1 中，我们根据这些目标分析了一系列现有系统。

MapReduce 和 Dryad 已经支持透明容错，基于局部性的调度和透明扩展。此外，Dryad 支持任意任务依赖，这使它能够执行比 MapReduce 更大的计算类。但是，它们都不支持依赖于数据的控制流，因此每个计算中的工作必须是静态预先确定的。

各种系统提供数据相关的控制流程，但牺牲了其他功能。 Google 的 Pregel 是支持控制流的分布式执行引擎的最大规模示例。 Pregel 是一个用于执行图算法（如 PageRank）的批量同步并行（BSP）系统，Pregel 计算分为“supersteps”，在此期间，对图中的每个顶点执行“vertex method”。至关重要的是，每个顶点都可以投票以终止计算，并且当所有顶点投票终止时计算终止。然而，与简单的 MapReduce 作业一样，Pregel 计算仅在单个数据集上运行，并且编程模型不支持多个计算的组合。

最近的两个系统为 MapReduce 添加了迭代功能。CGL-MapReduce 是 MapReduce 的一个新实现，它在几个 MapReduce 作业中缓存 RAM 中的静态（循环不变）数据。HaLoop 扩展了 Hadoop，能够评估减少输出的收敛功能。两个系统都不提供跨多个迭代的容错，也不支持 Dryad 样式

的任务依赖图。

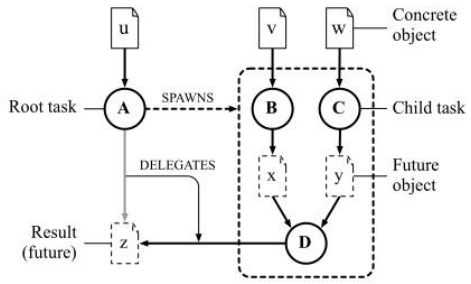
最后,Piccolo 是一种新的数据并行编程模型,它使用分区的内存中键值表来代替 MapReduce 的 reduce 阶段。 Piccolo 程序分为“内核”函数，它们并行应用于表分区，通常将键值对写入一个或多个其他表。“控制”功能协调内核功能，并且它可以执行任意数据相关的控制流程。Piccolo 支持用户辅助检查点（基于 Chandy-Lamport 算法），并且仅限于固定的群集成员资格。如果单个节点出现故障,则必须从具有相同数量节点的检查点重新启动整个计算。

我们相信 CIEL 是第一个支持 Figure 1 中所有五个目标的系统。CIEL 设计用于跨大数据集的粗粒度并行,MapReduce 和 Dryad 也是如此。在整个数据集可以放入 RAM 的情况下,Piccolo 可能更高效，因为它可以避免写入磁盘。最重要的是，实现最高性能需要大量的开发人员努力，使用显式消息传递等低级技术。

3 CIEL

CIEL 是一个分布式执行引擎，可以执行具有任意数据依赖控制流的程序。在本节中，我们首先描述 CIEL 支持的核心抽象：动态任务图。然后，我们描述 CIEL 如何执行表示为动态任务图的作业。最后，我们描述了用于分布式数据流计算的 CIEL 集群的具体体系结构。

3.1 动态任务图



(a) Dynamic task graph

Task ID	Dependencies	Expected outputs
A	{ u }	z
B	{ v }	x
C	{ w }	y
D	{ x, y }	z

Object ID	Produced by	Locations
u	–	{ host19, host85 }
v	–	{ host21, host23 }
w	–	{ host22, host57 }
x	**B**	∅
y	**C**	∅
z	~~**A**~~ **D**	∅

(b) Task and object tables

Figure 2:CIEL 作业由动态任务图表示，其中包含任务和对象。在此示例中，根任务 A 生成任务 B，C 和 D，并将其结果的生成委托给 D。C IEL 使用任务和对象表来表示图。

在本小节中，我们定义了三个 CIEL 原语 - 对象，引用和任务 - 并解释它们在动态任务图中的相关性（Figure 2）。

CIEL 是一个以数据为中心的执行引擎：CIEL 作业的目标是生成一个或多个输出对象。对象是一个非结构化的，有限长度的字节序列。每个对象都有一个唯一的名称：如果两个对象同名，则它们必须具有相同的内容。为了简化一致性和副本，对象在写入后是不可变的，但有时可以附加到一个对象中。

能够在不拥有其全部内容的情况下描述该对象是有帮助的; CIEL 使用引用来达到此目的。引用包括名称和一组位置（例如主机名 - 端口对），用于存储具有该名称的对象。位置集可

以是空的：在这种情况下，引用是对尚未生成的对象的未来引用。否则，它是一个具体的参考，可以被消费。

CIEL 作业通过执行任务来进行。任务是一种非阻塞原子计算，完全在一台机器上执行。任务具有一个或多个依赖项，这些依赖项由引用表示，并且当所有依赖项变得具体时，任务变为可运行。依赖项包括一个特殊对象，它指定任务的行为（例如可执行二进制文件或 Java 类），并可能将某些结构强加给其他依赖项。为了简化容错，CIEL 要求所有任务计算其依赖关系的确定性函数。任务还具有一个或多个**预期输出**，这些输出是该任务将创建或委派另一个任务创建的对象的**名称**。

任务可以有两个外部可观察的行为。首先，任务可以通过为这些对象创建具体的引用来发布一个或多个对象。特别是，任务可以为其预期输出发布对象，如果其他任务依赖于那些输出，则可能导致其他任务变为可运行。但是，为了**支持依赖于数据的控制流，任务还可以生成执行额外计算的新任务**。CIEL 对任务行为强制执行以下条件：

1.对于每个预期输出，任务必须发布具体引用，或者将子任务的名称生成为预期输出。这确保了，只要子项最终终止，任何依赖于父项输出的任务最终都将变为可运行。

2.子任务必须仅依赖于具体的引用（即已经存在的对象）或已指定运行任务的未来输出的引用（即已经预期要发布的对象）。这可以防止死锁，因为**循环不能在依赖图中形成**。

动态任务图存储任务和对象之间的关系。从对象到任务的边缘意味着任务依赖于该对象。从任务到对象的边缘意味着期望任务输出该对象。当作业运行时，新任务将添加到动态任务图中，并且当预期新生成的任务产生对象时，将重写边。

动态任务图提供类似于尾递归的低级数据相关控制流：任务产生其输出（类似于返回值）或产生新任务以产生该输出（类似于尾调用）。它还提供数据并行的功能，因为可以并行分派独立任务。但是，我们不希望程序员手动构建动态任务图，而是提供 Skywriting 脚本语言以编程方式生成这些图。

3.2 评估对象

给定动态任务图，CIEL 的作用是评估与作业输出相对应的一个或多个对象。实际上，可以将 CIEL 作业指定为仅具有具体依赖关系的单个根任务，以及指定计算的最终结果的预期输出。这导致了两种自然策略，即拓扑排序的变体：

Eager evaluation:由于任务依赖关系形成 DAG，因此至少一个任务必须仅具有具体的依赖关系。首先执行只有具体依赖的任务;随后在所有依赖项变得具体时执行任务。

Lazy evaluation:寻求评估根任务的预期输出。要评估对象，请确定生成预期对象的任务 T.如果 T 只有具体的依赖关系，请立即执行;否则，阻塞 T 并使用相同的过程递归地评估其所有未实现的依赖关系。当被阻止的任务的输入变得具体时，执行它。将所需对象的生成委托给安排的任务时，请重新评估该对象。

当我们第一次开发 CIEL 时，我们尝试了这两种策略，后来专门使用 Lazy evaluation，因为它更自然地支持我们在 §5 中描述的容错和记忆功能。

3.3 系统架构

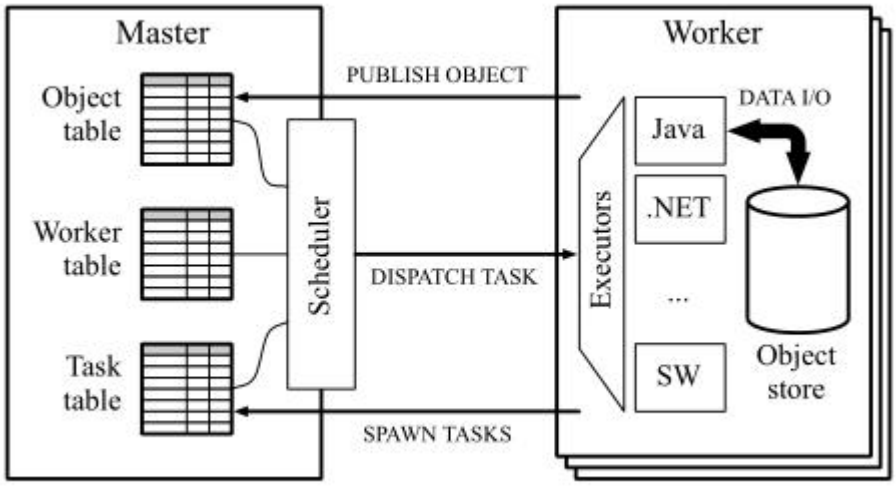


Figure 3:CIEL 集群有一个 master 和许多 worker。 master 将任务分派给 worker 执行。任务完成后，worker 发布一组对象并可能产生更多任务。

Figure 3 显示了 C IEL 集群的体系结构。单个 master 协调作业的端到端执行，并且多个 worker 执行单个任务。

master 在对象表和任务表中维护动态任务图的当前状态（Figure 2（b））。对象表中的每一行都包含该对象的最新引用，包括其位置（如果有），以及指向期望生成它的任务的指针（如果有的话：如果一个对象通过外部工具加载到集群中，它将没有任务指针）。任务表中的每一行对应一个生成的任务，并包含指向任务所依赖的引用的指针。

master 调度程序负责在 CIEL 计算中取得进展：它采用懒惰策略评估输出对象并将可运行任务与空闲 worker 配对。由于任务输入和输出可能非常大（每个任务的千兆字节数量级），所有批量数据都是存储在 worker 上，并且 master 处理之间的引用。master 使用基于多队列的调度程序（派生自 Hadoop）将**任务分派给离数据最近**的 worker。如果 worker 需要获取远程对象，它将直接从另一个 worker 读取该对象。

worker 执行任务并存储对象。在启动时，worker 向 master 注册，并定期发送心跳以证明其持续可用性。将任务分派给 worker 时，将调用相应的执行程序。 executor 是一个通用组件，它准备输入数据以供使用，并在其上调用一些计算，通常是执行外部进程。我们已经为 Java，.NET，基于 shell 和本地代码实现了简单的执行程序，并为 Skywriting 实现了更复杂的执行程序。

假设一个 worker 成功执行一个任务，它将希望发布的一组**引用**回复给 master，并为它希望生成的任何新任务回复一个**任务描述符列表**。然后，master 将更新对象表和任务表，并重

新评估现在可运行的任务集。

除了 master 和 worker 之外，还会有一个或多个客户端（未显示）。客户端的角色很小：它将作业提交给 master，并轮询 master 以发现作业状态或阻塞直到作业完成。

作业提交消息包含**根任务**，该任务必须只具有**具体的依赖关系**。master 将根任务添加到任务表，并通过懒惰策略评估其输出来启动作业。

请注意，为简单起见，CIEL 当前使用单个 master。尽管如此，我们的实现可以从主故障中恢复，并且在我们的评估期间不会导致性能瓶颈。但是，如果它在将来成为一个问题，那么就可以对主状态（任务表和对象表）进行分片。在多个主机之间，逻辑上保留单个 master 的功能。

4 Skywriting

```
function process_chunk(chunk, prev_result) {
  // Execute native code for chunk processing.
  // Returns a reference to a partial result.
  return spawn_exec(...);
}

function is_converged(curr_result, prev_result) {
  // Execute native code for convergence test.
  // Returns a reference to a boolean.
  return spawn_exec(...)[0];
}

input_data = [ref("ciel://host137/chunk0"),
              ref("ciel://host223/chunk1"),
              ...];
curr = ...; // Initial guess at the result.

do {
  prev = curr;
  curr = [];
  for (chunk in input_data) {
    curr += process_chunk(chunk, prev);
  }
} while (!*is_converged(curr, prev));

return curr;
```

Figure 4:在 Skywriting 中实现的迭代计算。 输入数据是 n 个输入块的列表，并且 curr 被初始化为 n 个部分结果的列表。

Skywriting 是一种用于表达在 CIEL 之上运行的任务级并行性的语言。Skywriting 是图灵完整脚本语言，可以使用 while 循环和递归函数等构造表达任意数据相关的控制流。Figure 4 显示了一个计算迭代算法的 Skywriting 脚本示例;我们在 k-means 实验中使用了类似的结构。

我们在之前的一篇论文中介绍过 Skywriting，但在这里简要介绍一下主要特征：

•**ref(url)**返回对存储在给定 URL 的数据的引用。该函数支持常见的 URL 方案和自定义 ciel 方案，后者访问 CIEL 对象表中的条目。如果 URL 是外部的，则 CIEL 将数据作为对象下载到集群中，并为该对象指定名称。

•**spawn(f，[arg，...])**产生并行任务来评估 f(arg，...)。Skywriting 函数不会有副效果，所有参数都按值传递。返回值是对 f(arg，...)结果的引用。

•**exec(executor，args，n)**使用给定的 args 同步运行指定的 executor。executor 将产生 n 个输出。返回值是对这些输出的 n 个引用的列表。

•**spawn_exec(executor，args，n)**生成并行任务以使用给定的 args 运行指定的 executor。与 exec()一样，返回值是对这些输出的 n 个引用的列表。

•**间接引用（一元 - *）运算符**可以应用于任何引用;它将引用的数据加载到 Skywriting 执行上下文中，并评估结果数据的结构。

在下文中，我们将描述 Skywriting 如何映射到 CIEL 基元。我们描述了如何创建任务，如何使用引用来促进依赖于数据的控制流，以及 Skywriting 与其他框架之间的关系。

4.1 创建任务

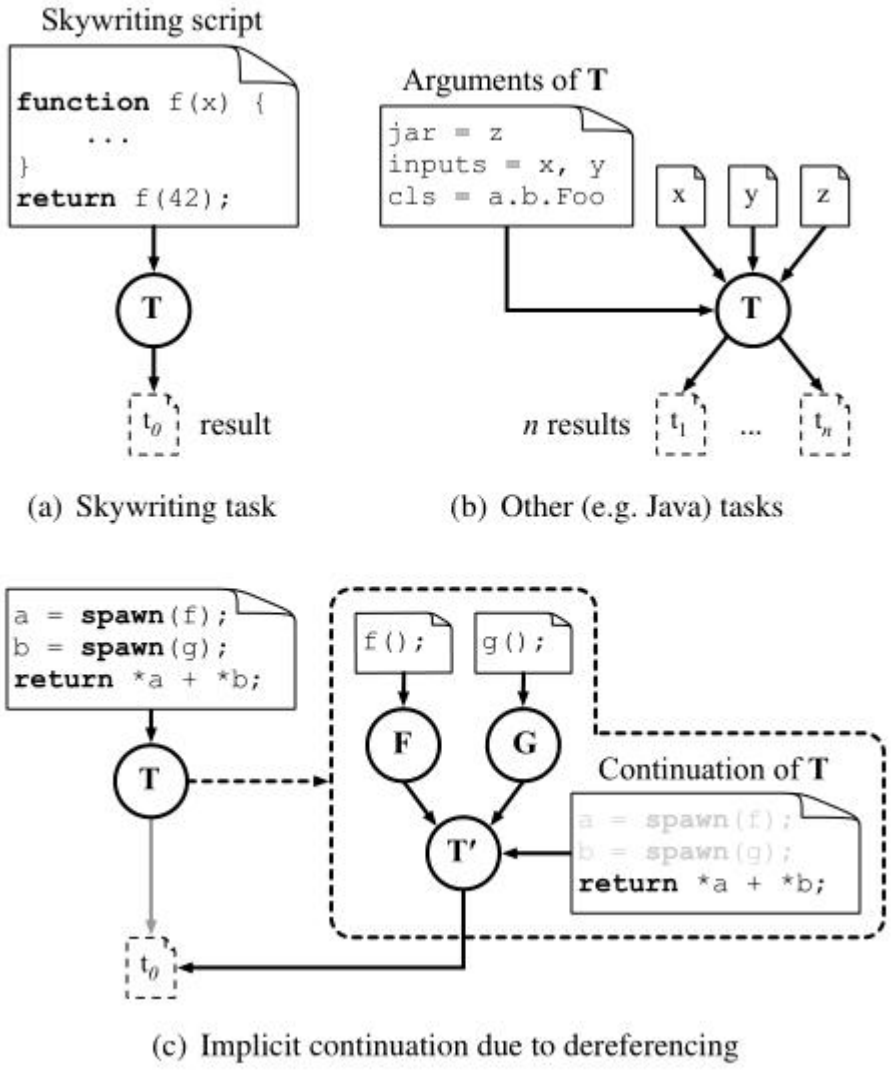


Figure 5:Skywriting 中的任务创建。可以使用(a)spawn()，(b)spawn_exec()和(c)间接引用(*)运算符创建任务。

Skywriting 的独特之处在于它能够在执行作业的过程中产生新任务。该语言提供了两个显式机制来生成新任务(spawn()和 spawn_exec()函数)和一个隐式机制(*-operator)。Figure 5 总结了这些机制。

spawn()函数创建一个新任务来运行给定的 Skywriting 函数。为此，Skywriting 运行时首先创建一个包含新任务环境的数据对象，包括要执行的函数的内容以及传递给函数的任何参数的值。此对象称为 Skywriting 后续，因为它**封装了计算的状态**。然后，运行时为新任务创建任务描述符，其中包括新任务的依赖。最后，它为任务结果分配一个引用，并将其返回给调用脚本。Figure 5(a)显示了创建任务的结构。

spawn_exec()函数是一种较低级别的任务创建机制，允许调用者调用不同语言编写的代码。通常，不直接调用此函数，而是通过相关 executor 的包装器（例如内置的 java()库函数)。

当调用 spawn_exec()时，运行时将参数序列化为数据对象并创建依赖于该对象的任务（Figure 5(b)）。如果 spawn_exec()的参数包含引用，则运行时会将这些引用添加到新任务的依赖项中，以确保 CIEL 在其所有参数都可用之前不会调度任务。同样，运行时为任务输出创建引用，并将它们返回给调用脚本。

如果任务尝试间接引用尚未创建的对象，例如，调用 spawn()的结果，当前任务必须阻塞。但是，**CIEL 任务是非阻塞**的：所有同步（和数据流）必须在动态任务图（第 3.1 节）中显式化。为了解决这个矛盾，运行时隐式地创建一个用于继续执行的任务，该任务依赖于间接引用的对象和当前任务的后续操作（即当前的 Skywriting 执行堆栈）。因此，新任务仅在生成间接引用的对象时运行，这提供了必要的同步。Figure 5(c)显示了当任务间接引用 spawn()的结果时产生的依赖图。

任务在到达 return 语句时终止（或者在 future 对象的引用上阻塞）。 Skywriting 任务有一个输出，它是 return 语句中表达式的值。 在终止时，运行时将**输出存储在本地对象库**中，**发布对该对象的具体引用，并按创建顺序将生成的任务列表发送给 master**。

Skywriting 确保动态任务图保持非循环。在评估由任务创建的函数时，任务的依赖关系是固定的，这意味着在评估函数之前，它们只能包含存储在本地 Skywriting 范围中的引用。因此，任务不能依赖于自身或其任何后续任务。请注意，Skywriting 任务可以在其返回值或随后的任务创建函数调用中传递引用。这使脚本能够创建任意非循环依赖图，例如 MapReduce 依赖图（第 4.3 节）。

4.2 数据相关控制流

Skywriting 旨在协调以数据为中心的计算，这意味着计算中的对象可以分为两个空间：
数据空间。包含大小可能高达几千兆字节的大型数据对象。
协调空间。包含**确定控制流**的小对象（如整数，布尔值，字符串，列表和词典）。

通常，数据空间中的对象由编程语言处理，以实现比 Skywriting 提供的更好的 I/O 或计算性能。在现有的分布式执行引擎（例如 MapReduce 和 Dryad）中，数据空间和协调空间是不相交的，这阻止了这些系统支持依赖于数据的控制流。

为了支持依赖于数据的控制流，数据必须能够从数据空间传递到协调空间，以便它可以帮助确定控制流。在 Skywriting 中，*操作符将引用从数据空间对象转换为协调空间的值。
生成的任务也许会由任何 executor 运行，必须以 Sky-writing 可识别的格式编写引用的对象；为此，我们使用 JavaScript Object Notation（JSON）。此序列化格式**仅用于**传递给 Skywriting 的引用，并且大多数 executor 使用适当的二进制格式作为其数据。

4.3 其他语言与框架

```
function apply(f, list) {
  outputs = [];
  for (i in range(len(list))) {
    outputs[i] = f(list[i]);
  }
  return outputs;
}

function shuffle(inputs, num_outputs) {
  outputs = [];
  for (i in range(num_outputs)) {
    outputs[i] = [];
    for (j in range(len(inputs))) {
      outputs[i][j] = inputs[j][i];
    }
  }
  return outputs;
}

function mapreduce(inputs, mapper, reducer, r) {
  map_outputs = apply(mapper, inputs);
  reduce_inputs = shuffle(map_outputs, r);
  reduce_outputs = apply(reducer, reduce_inputs);
  return reduce_outputs;
}
```

Figure 6:在 Skywriting 中实现 MapReduce 编程模型。用户提供输入列表，mapper 函数，reducer 函数和要使用的 reduce 个数。

像 MapReduce 这样的系统至少在某些方面变得流行，因为它们的界面简单：开发人员只需要一对 map()和 reduce()函数就可以指定整个分布式计算。为了证明 Skywriting 接近这种简洁程度，Figure 6 展示了 MapReduce 执行模型的实现，它取自于 Skywriting 标准库。

mapreduce()函数首先将 mapper 应用于输入的每个元素。mapper 是一个 Skywriting 函数，它返回 r 个元素的列表。然后对 map 操作后的输出进行 shuffle，使得每个 map 的第 i 个输出变为第 i 个 reduce 的输入。最后，将收集到的 reduce 的输入传到 reducer 函数运行 r 次。在典型的使用中，mapreduce()的输入是包含已拆分的数据对象，mapper 和 reducer 函数调用 spawn_exec()来执行另一种语言的计算。

请注意，mapper 函数负责在 reducer 之间对数据进行分区，reducer 函数必须合并它接收的输入。如果需要，mapper 的实现还可以包含组合器。为了简化开发，我们已将 Hadoop MapReduce 框架的一部分移植到 CIEL 任务中，并提供辅助函数来分区，合并和处理 Hadoop 文件格式。

任何编译为 DAG 任务的高级语言也可以编译为 Skywriting 程序，并在 CIEL 集群上执行。例如，可以为 Pig 和 DryadLINQ 开发 Skywriting 后端，从而提高了扩展这些语言的可能性，

并支持无限迭代。

5 实现所遇问题

CIEL 和 Skywriting 的当前实现包含大约 9,500 行 Python 代码，以及执行器绑定中的几百行 C，Java 和其他语言。本节的其余部分描述了我们实现的三个有趣特性：记忆化（https://en.wikipedia.org/wiki/Memoization），主容错和流。

5.1 确定性命名和记忆化

回想一下，CIEL 集群中的所有对象都具有唯一的名称。在本小节中，我们将展示如何通过适当的名称选择来实现记忆化功能。

我们最初的 C IEL 实现使用全局唯一标识符（UUID）来标识所有数据对象。虽然这是一个概念上简单的方案，但它使容错变得复杂，因为 master 必须记录生成的 UUID 以支持任务失败后重新运行。

这促使我们重新考虑名称选择。为了支持容错，现有系统假设单个任务是确定的，并且 CIEL 做出相同的假设（§3.1）。因此，**具有相同依赖关系的两个任务**（包括作为依赖关系的可执行代码）**将具有相同的行为**。因此，使用以下 Skywriting 语句创建 n 个输出的任务

```
result = spawn_exec(executor, args, n);
```

将完全由 executor，args，n 及其标志决定。因此，我们可以通过将 executor，args，n 和 i 通过适当的分隔符连接来构造第 i 个输出的名称。但是，由于 args 本身可能包含引用，因此名称可能会变得难以管理。因此，我们使用抗冲突哈希函数 H 来计算 args 和 n 的 digest，从而得到结果名称：

executor	:	$\mathcal{H}(\text{args}\|\|\text{n})$	:	i

我们目前使用 160 位 SHA-1 哈希函数来生成 digest。

回想一下§3.2 中的惰性评估算法：任务的预期输出可以解决另一个阻塞任务的依赖，只有这样任务才会被执行。如果先前任务已经生成了新任务的输出，则根本不需要执行新任务。因此，由于确定性命名，CIEL 会记住任务的结果，这可以提高执行重复任务的作业的性能。

我们的记忆化的目标与最近的 Nectar 系统类似。 Nectar 对 DryadLINQ 查询执行静态分析，以识别先前在同一数据上计算的子查询.Nectar 在 DryadLINQ 级别实现，这使得它能够对每个任务的语义和缓存中间结果的成本/收益比做出假设。例如，如果先前的查询在当前查询的输入的前缀上操作，则 Nectar 可以重复使用来自先前查询的交换和关联聚合的结果。CIEL 工作的表现力使得运行这些分析变得更具挑战性，我们正在研究 Skywriting 程序中如何通过简单注释在我们的系统中提供类似的功能。

5.2 容错

分布式执行引擎必须在面对网络和计算机故障时继续运行。随着工作时间变得更长（并且，由于 CIEL 允许无限次迭代，它们可能变得非常长）经历故障的概率增加。因此，CIEL 必须容忍计算中涉及的任何机器的故障：客户端，worker 和 master。

客户端容错是微不足道的，因为 CIEL 本身支持迭代作业并从头到尾管理作业执行。客户端唯一的角色是提交作业：如果客户端随后失败，则作业将继续而不会中断。相比之下，为了在非支持迭代框架上执行迭代作业，客户端必须运行一个驱动程序，该程序执行所有依赖于数据的控制流（例如收敛测试）。由于驱动程序在框架外执行，它不会从透明容错中受益，开发人员必须手动提供此功能，例如通过检查执行状态。在我们的系统中，一个 Skywriting 脚本取代了驱动程序，CIEL 可靠地执行整个脚本。

CIEL 中的 **worker 容错**类似于 Dryad。master 接收来自每个 worker 的定期心跳消息，并且如果（i）它在指定的超时之后没有发送心跳，并且（ii）它不响应来自 master 的反馈消息，则认为 worker 已经故障。此时，如果已为 worker 分配任务，则认为该任务已失败。

当任务失败时，CIEL 会自动重新执行它。但是，如果由于其输入存储在失败的 worker 中而失败，则该任务不可再运行。在这种情况下，CIEL 递归地重新执行先前的任务，直到满足失败任务的所有依赖关系。为实现此目的，master 使对象表中每个丢失输入的位置无效，并采取懒惰策略重新评估丢失的输入。依赖于来自故障 worker 中数据的其他任务也将失败，并且这些任务同样由 master 重新执行。

CIEL 还支持 **master 容错**。在 MapReduce 和 Dryad 中，如果一个作业的 master 进程失败，它就会完全失败;在 Hadoop 中，如果 JobTracker 失败，所有作业都会失败;并且 master 故障通常会导致多次重复提交作业的驱动程序失败。但是在 CIEL 中，所有 master 状态都可以从活动中的作业集中派生出。至少，持久化存储每个存活作业的根任务允许创建新的 master 并立即恢复执行。CIEL 提供了三种扩展 master 容错的互补机制：持久日志记录，从 master 和对象表重构。

创建新作业时，master 会为作业创建日志文件，并将其根任务描述符同步写入日志。默认情况下，它将日志写入本地辅助存储上的日志目录，但它也可以写入网络文件系统或分布式存储服务。创建新任务时，其描述符将异步附加到日志文件，并定期写入到磁盘。作业完成后，将对其结果的具体引用写入日志目录。重新启动后，master 会在其日志目录中**扫描没有匹配结果的作业**。对于这些作业，它会重新生成日志，重建动态任务图。处理完所有日志后，master 会通过懒惰策略评估其输出来重新启动作业。

或者，master 可以将状态更新记录到从 master。在从 master 向主 master 注册后，主 master 将所有任务表和对象表更新转发到从 master。每个新作业都是同步发送的，以确保在客户端收到确认之前将其记录在从 master 中。此外，从 master 记录了向主 master 注册的每个 worker 的地址，以便它可以在故障转移方案中联系到 worker。从 master 定期向主 master 发送心跳;当它检测到主 master 发生故障时，从 master 指示所有 worker 重新向其注册。我们在 §6.5 中评估这种情况。

如果主 master 失败并随后重新启动，worker 可以使用其本地对象存储库的内容帮助重建对

象表。如果主 master 没有响应请求，则认为主 master 故障了。此时，worker 切换到重新注册模式，并且不再发送心跳消息，而是定期发送注册请求到同一网络位置。当 worker 最终联系新的主 master 时，主 master 使用基于 GFS 主 master 恢复的协议来提取 worker 中数据对象的列表。

5.3 流

我们之前对任务的定义（第 3.1 节）声明任务生成的数据对象是作为其结果的一部分。这个定义意味着对象生成是原子性的：对象要么完全存在要么根本不存在。但是，由于数据对象可能非常大，因此通常有机会在任务之间流式传输部分写入的对象，这可能导致流式并行。

如果生成的任务具有流式输出，则它向主 master 发送预发布的消息，其中包含每个流式输出的流引用。这些引用用于更新对象表，并可能阻塞其他任务：如流对象消费者。流对象消费者像以前一样执行，但执行的代码从命名管道中而不是本地文件读取其输入。担任消费者的 worker 进程会启动单独的线程从担任生产者的 worker 进程中取得输入信息块，并将它们写入管道。当生产者成功终止时，它会提交其输出，该输出向消费者发出信号，告知不再有数据需要读取。

在当前实现中，流消息生产者还将其输出数据写入本地磁盘，因此，如果流消费者失败，则流生产者不受影响。如果生产者在拥有消费者时故障，则生产者回滚任何部分写入的输出。在这种情况下，消费者将因缺少输入而失败，并触发生产者的重新执行（第 5.2 节）。我们正在研究更复杂的容错和调度策略，这些策略允许生产者和消费者通过直接 TCP 流进行通信，如 Dryad 和 Hadoop Online Prototype。但是，正如我们在下一节中所示，对流式传输的支持为某些应用程序带来了有用的性能优势。

##

6 评估

我们开发 CIEL 的主要目标是开发一个系统，该系统支持比现有分布式执行引擎更强大的计算模型，而不会在性能方面产生高成本。在本节中，我们将评估运行 Skywriting 中实现的各种应用程序的 CIEL 的性能。我们调查以下问题：

1. CIEL 的性能与生产使用的系统（即 Hadoop）相比如何？ （§6.1，§6.2）
2. CIEL 在执行迭代算法时提供了哪些好处？ （第 6.2 节）
3. CIEL 对计算密集型任务施加了哪些开销？ （§6.3，§6.4）
4. 主要故障对端到端工作性能有何影响？ （§6.5）

对于我们的评估，我们选择了一组算法来回答这些问题，包括 MapReduce 风格，迭代和计算密集型算法。我们选择动态编程算法来演示 CIEL 执行算法的能力，这些算法具有不转换为 MapReduce 模型的数据依赖性。

本节中介绍的所有结果均使用 Amazon EC2 云计算平台上的 m1.small 虚拟机进行收集。在撰写本文时，m1.small 实例具有 1.7 GB 的 RAM 和 1 个虚拟核心（相当于 2007 AMD Opteron 或 Intel Xeon 处理器）。在所有情况下，操作系统都是 Ubuntu 10.04，在 32 位模式下使用 Linux 内核版本 2.6.32。由于虚拟机是单核的，我们每台机器运行一个 CIEL 工作器，并配

置 Hadoop 为每个 TaskTracker 使用一个 map 槽。
##

8 结论

我们设计 CIEL，以提供现有分布式执行引擎提供的功能集合。使用 Skywriting，可以以命令式方式编写迭代算法，并通过透明的容错和自动分发来执行它们。但是，CIEL 还可以执行任何 MapReduce 作业或 Dryad 图，并且对迭代的支持允许它执行 Pregel 和 Piccolo 样式的计算。

我们的下一步是将 CIEL 原语与现有的编程语言集成。目前，只有 Skywriting 脚本可以创建新任务。这并不限制普遍性，但它要求开发人员在 Skywriting 中重写他们的驱动程序。所有与调度相关的控制流决策必须最终通过解释代码，这可能会对 Skywriting 运行时施加压力。 Skywriting 的主要好处是它掩盖了间接引用算子背后的连续传递风格的复杂性（§4.2）。我们现在寻求一种方法将这种抽象扩展到主流编程语言。

CIEL 可以扩展到数百台机器，但其他扩展挑战仍然存在。例如，目前还不清楚如何在一台机器中最好地利用多个内核，我们目前将此问题传递给执行器，后者可以充分利用单个机器。这使应用程序开发人员能够以更高的复杂性为代价，精确控制程序的执行方式。但是，如果任务本质上是顺序的并且有多个核可用，则会限制效率。此外，通过在单个主机上并置流生产者及其使用者而节省的 I/O 可能会超过 CPU 争用的成本。找到最佳调度时间是一个难题，我们正在研究简单的注释方案和启发式方法，以提高常见情况下的性能。最近关于集群操作系统和调度算法的工作使人们希望这个问题能够得到一个优雅的解决方案。
有关 CIEL 和 Skywriting 的更多信息，包括源代码，语言参考和教程，可从项目网站获得：
http://www.cl.cam.ac.uk/netos/ciel/