基于并行子树构建的XML解析方法

陈荣鑫[1,2] 廖湖声[1] 陈维斌[3]
(北京工业大学计算机学院 北京 100124)[1] (集美大学计算机工程学院 厦门 361021)[2]
(华侨大学计算机学院 泉州 362021)[3]

摘 要 XML解析的高耗时特点制约着XML应用系统整体性能的提高,并行化是一种重要的优化手段。现有的并行XML解析算法存在的问题是需要通过预处理进行数据划分,才能实现分片并行完全解析处理。预处理往往很耗时,若进行优化处理,则实现复杂。提出的方法可实现对XML数据任意分片直接进行解析,并行构建各个片断中的子树,再通过子树合并获得全局的文档树。实验结果表明该方法能有效利用多核计算环境,并行实现XML解析。

关键词 XML解析,子树构建,并行,多核
中图法分类号 TP311.13 **文献标识码** A

XML Parsing Schema Based on Parallel Sub-tree Construction

CHEN Rong-xin[1,2] LIAO Hu-sheng[1] CHEN Wei-bin[3]
(College of Computer Science, Beijing University of Technology, Beijing 100124, China)[1]
(Computer Engineering College, Jimei University, Xiamen 361021, China)[2]
(Computer Science College, Huaqiao University, Quanzhou 362021, China)[3]

Abstract Since XML parsing is time-consuming operation which greatly affects the performance of XML application, parallelization is an important optimization measure. Existing parsing methods need pre-parsing stage to ensure proper data partition so as to complete XML segments parsing in parallel, however, pre-parsing tends to be long-running and difficult to be optimized. This paper presented a schema which supports parallel sub-tree construction upon arbitrary XML segments. Sub-trees were merged to form whole XML tree in final stage. Experiment results indicate that our schema can efficiently realize parallel XML parsing in multi-core environment.

Keywords XML parsing, Sub-tree construction, Parallel, Multi-core

1 引言

XML作为信息交换和存储标准被日益广泛应用。XML本身是半结构化数据,XML在大多数的实际应用中以文档形式存在,内部数据关系和内容的提取和利用首先需要进行解析,以获取XML树,应用程序通过对XML树的操作来实现XML访问。近年来也出现了以数据为中心的XML数据库,包括基于关系数据存储和原生XML存储。对XML数据的存储、转化和提取等操作都依赖XML解析工作。而解析是一种高耗时的操作,几乎成为大多数XML应用的性能瓶颈[1]。如何有效提高XML解析性能,进而提高XML应用系统的整体性能是个重要课题。

并行化是实现高速计算的一种重要途径。随着多核环境日益普及,很多问题求解都借助并行技术实现[2]。早期的XML数据并行化处理主要是对由XML数据转换成的关系数据进行数据划分,以支持并行查询,尚未涉及并行解析这个问题[3]。近年来,Wei Lu和Yinfei Pan等人做了很多XML并行解析的研究。Wei Lu等人提出一种XML DOM的并行解析方法[4,5],其包含预解析和并行解析两个主要阶段。预解析生成了框架(skeleton),作为数据划分的基础,同时为最后的合并提供指导。预解析设想比后续并行完全解析耗时少,不过串行执行还是比较低效。所以后来出现了针对框架生成的并行化优化方法,比如采用Meta-DFA法[6]和并发Transducers[7]法,这些预处理的并行化措施增加了实现的复杂性。此外,Yinfei Pan等人提出的基于SAX的并行化解析方法[8],整个处理划分为多个阶段,采用流水线连接。为获取块(Chunk)的划分框架,同样需要对整个XML数据进行读取,而其中某些串行执行阶段造成了性能瓶颈。总的说来,这些方法无法对任意划分的片段进行解析,数据片段划分需要一个框架生成预处理阶段,由于该阶段需要扫描整个XML数据文档,往往耗时较高,需要依靠某些优化措施,比如并行化处理,才能保证后续并行解析性能提高的效果不被抵消,这就造成预处理过于复杂。另外在解析结果的应用方面,目前大多方法的解析结果仅提供DOM接口,未考虑提供区域编码接口,而很多高效的twig查询算法是基于区域码[9]的。

本文提出的基于并行子树构建的解析方法,目的是利用

到稿日期:2010-04-22 返修日期:2010-09-08 本文受福建省自然科学基金项目(2008J04005)和北京市自然科学基金项目(4082003)资助。
陈荣鑫(1975—),男,博士生,讲师,主要研究方向为软件自动化与数据库技术,E-mail:ch2002star@163.com;廖湖声(1954—),男,博士生导师,教授,主要研究方向为软件自动化;陈维斌(1957—),男,教授,主要研究方向为数据库技术。

多核环境，通过并行方式提高 XML 解析的效率。该方法具备以下特点：

1)任意划分片解析：支持对 XML 任意划分后的非结构完整的片段进行解析，可以对每个片段独立解析，有效支持并行处理机制。

2)一次 XML 文档读取：由于仅需要轻量级的扫描进行片段边界确认，不需要任何框架生成预处理。整个处理实际仅需要一次扫描原始 XML 文档。

3)提供访问接口：获得的 DOM 关系以线性存储形式保存，使得包装的存取原语可以高效访问 XML。除了记录用于操纵 DOM 树的主要信息外，还记录了节点区域码，满足基于区域码的各种查询算法如 twig 查询的需求。

本文第 2,3,4 节分别讨论该方法所包含的 3 个主要工作阶段：数据分片、并行子树构建和子树合并；其中第 2 节给出了相关概念描述；第 3,4 节给出了主要算法，并给出了正确合并的相关证明；第 5 节进行实验验证与效果分析；最后总结并展望未来工作。

2 数据分片

数据分片是指按一定的规则把连续的 XML 文档划分成相互独立的数据片段，以便于进行数据并行处理。现有的并行 XML 解析方法依赖框架生成预处理，获取合理的数据分片。相比之下，任意数据划分是一种可按任意数据大小进行划分的非严格约束的划分方法。

2.1 基本数据描述

在 XML 数据任意划分处理中，涉及以下几种基本数据形式。

1)标签名称字符串：在最邻近的以“〈”开始和以“〉”结束之间的字符串。通常见到的标签名称字符串有以下 5 种：结束标签如〈/tagName〉；不带属性开始标签如〈tagName〉；空元素标签如〈tagName /〉；带属性开始标签如〈tagName att1 = ‘a001’…〉；以及带属性的空元素开始标签〈tagName att1 = ‘a001’…/〉。

2)失配字符串：指因片段划分，被分割到其他片段的标签名称字符串或者元素内容字符串。失配字符串可能出现在除第 1 个片段以外的片段起始位置和除最后片段以外的片段结束位置。

3)失配标签：指因片段划分，被分割到其他片段的不可配对标签。存在两种失配标签，即失配开始标签 mH 和失配结束标签 mT。某开始标签在本片段中找不到与之配对的结束标签，该开始标签叫 mH；同样可定义 mT。与 mH 配对的唯一 mT 可能出现在 mH 所在片段的所有后继片段中；同理，与 mT 配对的唯一 mH 可能出现在 mT 所在片段的所有前驱片段中。显然对整个结构良好的 XML 文档来说，全局失配标签一定是成对出现的。

4)子树：片段解析的局部结果，包括完整子树和残缺子树。完整子树指子树的所有信息都在同一个片段中；残缺子树指由于被截断，部分节点信息不在同一个片段中，根据完整开始标签建立节点，如果节点残缺则需要在后续调整工作中补全。解析一个 XML 片段将获得局部子树集合。

5)全局 XML 树：解析完成的整个 XML 文档树。包含 XML 节点关系信息与节点内容信息。目前主要考虑两种节点，即元素和属性节点。该数据经过进一步包装和存取原语设计，为各种存取操作提供接口。

2.2 任意数据分片

对独立的 XML 数据，一般可以根据给定的分片数，对数据等长划分。对 XML 文档任意划分片，采用文档位置逻辑定位而非对文档进行物理划分片段，只要文件将指针定位到各片段的位置开始读取即可。通过扫描直接划分的片段开始部分，而不是整个片段，即可获取片段边界信息，再进行片段边界调整。由于不必进行框架生成预处理，因此这是一种轻量级的划分方法。例如，对图 1 的 XML 文档任意划分，假设初始按(0,70,294,文件尾)字符位置划分 3 个数据片段，实线表示任意数据分割的位置；经过边界调整后将按(0,90,311,文件尾)划分，虚线表示调整后的划分边界。

图 1 XML 文档数据片段划分

对应的 XML 文档树将按图 2 分割，图中字串表示标签名，数字表示反映文档序的节点 ID；虚曲线表示分片对应的 XML 树分割的位置，不规则圈内的节点同属于一棵子树。

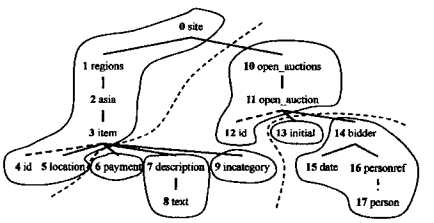

图 2 XML 树的划分

3 个片段各自构建子树：第 1 个片段：只有 1 棵子树，由节点 0—5 组成。第 2 个片段：有 4 棵子树，节点 6 组成片段内第 1 子树；节点 7,8 组成第 2 子树；节点 9 组成第 3 子树；节点 10—12 组成第 4 子树。第 3 个片段：有 2 棵子树，分别是片段内第 1 子树，含节点 13；第 2 子树，含节点 14—17。我们把属性节点当成特殊的元素节点，图中用虚直线表示，如节点 4,12 和 17。

分片解析过程将出现两类子树，一种是子树完整处在一个片段中，如图 2 中节点 7,8 构成的子树；另一种是残缺子树，如节点 10,11,12 构成的子树，该子树中存在元素节点未完整的情况，比如节点 10 的结束标签落在其它片段中。两种子树在分片解析时没有区别，只是在合并阶段需要对残缺子树进行特殊处理。我们处理残缺子树的办法就是把失配的结束标签补全，子树的实际结束位置取片段结束位置作参考值，真正子树的结束位置是在合并时，通过失配标签判定获得。

3 并行子树构建

该阶段对各个已划分的 XML 逻辑数据片段并行地构建

子树集合。需要维护一个片段内部的标签栈，以便进行失配标签识别和子树根节点识别等处理。片段内求取的子树信息，包含局部子树的节点关系、内容以及标签名。由于这阶段获得的信息尚不完全，需要在合并阶段通过调整工作完善。子树构建后还需要进行片段首尾失配字符串的处理。

3.1 树保存的信息

局部子树和全局 XML 树均需要保存以下信息项。区别是局部子树内的信息未完善，需要合并阶段的调整。

1)节点关系信息。节点 ID：唯一的标记，对应节点的存储位置。节点类型：主要是元素和属性两种节点。节点标签名 ID：通过 ID 可直接在节点标签名表里查找标签名。父节点位置：可用父节点 ID 表示。第一个子节点位置：当节点类型是属性时，为空。兄弟节点位置：指向右兄弟节点位置。第一个属性位置：当节点类型是节点时，为空。内容索引 ID：可用节点 ID 替代，通过 ID 直接在内容表里查找元素的内容文本或属性的值信息。扩展的信息(区域码)包括：层次(文档根节点是 0，其子节点值为 1，以此类推)、节点开始的文档位置和节点结束的文档位置。

2)节点内容信息。通过节点内容对照表存储元素节点内容和属性节点的值。对照表包括以下域：节点的内容索引(可用节点 ID 替代)；节点内容(对元素节点是元素内的文本，对属性节点是属性值)；是否压缩标记(对节点内容长度大于指定阈值的数据项进行压缩存储)。

3)节点标签名信息。为了避免节点标签名冗余，通过节点标签名对照表存储标签名，并可以通过 hash 查找快速定位。对照表包括两个域：标签名 ID(作为 hash 查找的关键字)和标签名信息。

3.2 子树构建

每个片段开始处理的节点一定是本片段内的第 1 棵子树的根节点；子树根节点只可能是元素节点。假定当前层次值为 L；ID 是记录当前节点 ID 的变量。处理节点构建子树时：若 $L=0$，对应处理节点为根节点，则设置 ID 为 0，节点类型为元素，其他信息待更新；若 $L\neq 0$，对应处理节点不是根节点，则设置 ID++，节点类型为元素或属性，父节点的 ID 为之前的节点 ID，其他信息待定。

算法 1 简要给出了子树构建框架：对某个数据片段连续读取标签直到片段结束，通过标签内容解析和关系识别，完成 XML 数据片段内的所有子树构建。

算法 1 子树构建框架

```
BuildSubTrees(segment)
1:pos←segment.startPos;
2:while(pos≠segment.endPos)
3:  pos←ReadCompletedTag(segment,pos,segment.endPos);
4:  ParseTagContent(completedTagString,subTrees);
5:ProcessEndUnmatched();
```

输入参数 segment 指定了某个数据片段。第 4 行 ParseTagContent 是标签名称字符串解析方法，其针对之前定义的 5 种标签字符串解析，获取子树节点包括属性节点的信息。第 5 行需要对失配标签进行记录，返回子树集合。

3.3 子树构建并行化

根据输入的 XML 逻辑数据片段，处理后获得该片段的子树集合以及各片段失配开始/结束标签序列。这部分工作交由不同线程并行完成，由于各分片的处理相互独立，解析操作属于易并行类型，可采用数据并行模式[10]处理。由于各个片段解析的数据没有依赖关系，只需分配 CPU 资源给各片段的解析线程。由于合并操作需要在所有片段完成解析后进行，因此并行任务采用计数器进行同步，见算法 2。

算法 2 子树并行构建

```
ParallelConstruct(segment[])
1:CountDownLatch latch←new CountDownLatch(sizeof(segment));
2:Executor exec←new Executor();
3:for each segment sg in segment[]
4:  exec.execute(BuildSubTreesTask(sg))
5:  latch.await();
6:exec.shutdown();
```

输入参数是各个数据分片信息。第 1 行创建一个以片段数目为参数的计数器，用于任务同步计数；第 2 行创建一个多线程执行服务，可以指定工作线程数目；第 4 行指派各个线程执行子树构建任务，BuildSubTreesTask 定义了以数据分片为参数的子树构建任务；第 5 行保证所有子树构建任务的同步；完成所有任务后，第 6 行关闭线程服务。

4 子树合并

合并之前需要进行预处理：根据各片解析获得的各子树信息，计算节点关系的存储空间大小，预开辟最终生成树空间。合并有两部分工作：一是节点关系信息调整，另一个是节点内容合并和节点标签名合并。

4.1 节点关系信息调整

由于各片解析的结果相互独立，未考虑全局情况，节点关系需要在合并节点关系时进行调整。调整工作包括以下几种类型。

1)调 0：除第 1 个片段以外其他片段的子树中各个节点信息中的指针域，以及节点 ID，需要加存储偏移量。实际只需对所有值不为空的指针域加偏移量。

2)调 1：找到子树根节点的父节点(挂接节点)，设置该节点的第一个孩子的指针域为子树根节点。由于子树根节点只可能是元素节点，挂接节点的属性指针域不受影响。

3)调 2：设置挂接节点的最末孩子的右兄弟指针域为子树根节点。

4)调 3：设置子树根节点的父节点指针域为挂接节点。

5)调 4：设置子树根节点的右兄弟指针域。由于调整按文档序进行，已经由调 2 处理时自动完成，不必另外做调整工作。

6)调 5：用子树根节点的挂接父节点的层次值加上当前层次值，作为子树节点的层次值。

7)调 6：调整被截断的节点的结束位置。根据失配标签信息，最后进行一次性调整。

以上调整 0—4 为基本节点关系调整；5，6 为区域码信息调整。

4.2 子树合并算法

子树合并的关键工作是完成节点关系的各种调整。算法 3 描述了合并过程中通过调 1，2，3 的工作，调整每棵子树的根节点，以及受影响的其他节点关系信息。输入参数 nodes 是所有节点，节点的偏移量的调整工作(调 0)之前已经完成；SubTrees 是并行构建获得的子树集合；offsetST 记录了各子树的偏移量。第 3 行由子树根节点求取挂接节点，调用的方

法参看算法 4，区别是为了表达简洁，这里忽略了两个全局参数。

算法 3 合并调整 1—3 工作

```
AdjustWork1to3(nodes[],SubTrees[],offsetST[])
1:for each subTree st in SubTrees[]
2:   pos←offsetST[st. treeID];
3:   parent← GetSubTreeRootNodeParent(st. rootNodeID)
4:   nodes[pos]. parent←parent. pos;//调 3
5:   if(nodes[parent. pos]. firstChild=UNKNOWN)
6:     nodes[pos]. firstChild←pos;//调 1
7:   else last←nodes[parent. pos]. firstChild
8:     while(nodes[last. pos]. nextSibling ≠ UNKNOWN)
9:       last←nodes[last. pos]. nextSibling
10:  nodes[last. pos]←pos //调 2
```

子树根节点的父节点是子树的挂接节点，根据 4.3 节的规则 1 可以实现子树的挂接节点求取。算法 4 给出了求取过程，通过跟踪栈保证失配标签正确配对和标签位置比较来进行挂接节点的求取。

算法 4 子树的挂接点的求取

```
GetSubTreeRootNodeParent(rootNode,missedTagList[],treeEndPos
[])
1:for each missedTag mt in missedTagList[]
2:   pos←treeEndPos [rootNode. nodeID]
3:   if(mt. pos >= pos)
4:     break;
5:else if(mt. type=END_TAG)
6:          traceStack. pop();
7:     else traceStack. push(mt);
8:return traceStack. pop();
```

参数 rootNode 表示子树根节点。missedTagList 是从各个分片收集的失配标签序列。treeEndPos 记录了子树结束在文档中的位置，判断子树的父节点时，存在两种情况：一是如果子树未被截，则 treeEndPos 记录的是根节点结束标签位置，判断挂接时，根节点开始标签在失配标签区间内；二是如果子树被截，则 treeEndPos 记录的是片段结束位置，判断挂接时，根节点开始标签刚好是某个失配开始标签位置。

4.3 子树合并的正确性

挂接节点的正确求取保证了子树合并的正确性，下文通过对规则 1 的证明，说明了算法 4 所描述的求取方法的正确性。

引理 1 失配开始标签所属节点一定是挂接节点，挂接节点至少可以挂接一棵子树。

证明：反证，假设失配开始标签所属节点不是挂接节点，那么说明该节点是完整节点，却带有失配标签，显然矛盾，故失配开始标签所属节点一定是挂接节点。如果失配开始标签所属节点没有挂接子树，将不可能完整，这就说明挂接节点至少要挂接一棵子树。另一方面，可以用实例说明一个挂接点可以有多个挂接子树，如图 2 中的节点 3。证毕。

定义 1(未配对失配标签) 指在失配标签序列的某个区间内没有匹配的标签。由于合并后的整个文档是结构良好的(well-formed)，故整个失配标签序列不存在未匹配的标签。

定义 2(前半区间) 指相对当前树根节点开始标签位置而言，在失配标签序列中，文档序靠前的所有失配标签。

规则 1 在失配标签序列前半区间中，离子树根节点开始标签最接近的未配对失配开始标签所在的节点就是该子树根节点的父节点。

证明：设失配开始标签序列 H 内的元素按文档序排列，则当 $i<j$，满足 $Hi.\ pos<Hj.\ pos$，其中用 i,j 表示元素在序列中的位置，用 pos 表示标签在文档中的位置。设失配开始标签所属的节点序列为 N，子树根节点为 r，其开始标签为 rh。在同一文档中，$rh.\ pos$ 与 $Hi.\ pos$ 具备可比性。根据引理 1，可以把挂接点和失配开始标签联系起来讨论。

情况 1 假设前半区间中，存在某已匹配失配标签节点 Nm，则其结束标签位置比 $rh.\ pos$ 靠前，那么 Nm 不可能包含 r，即说明已配对失配开始标签的节点不可能是挂接节点。

情况 2 假设前半区间中，存在未配对失配开始标签 Hi 和 Hj，分别属于节点 Ni,Nj，这两个节点可能为挂接节点，若 $i<j$，则 $Hi.\ pos<Hj.\ pos<rh.\ pos$，由 XML 结构良好的特点，只可能 Ni 包含 Nj，Nj 包含 r。如果 r 挂接在节点 Ni，则 r 与 Nj 之间只可能或者是兄弟关系，或者 r 包含 Nj，总之 r 不可能被节点 Nj 所包含，导致矛盾。只有 r 的被包含最小，才不产生矛盾，这个最小被包含对应最近未配对失配开始标签所在的节点。证毕。

图 3 说明子树挂接点的求取方法。分析由图 1 的实例分片解析获得的失配标签序列。图中第 1 行表示失配标签所在的节点 ID；第 2 行用‘〈’表示失配开始标签，‘〉’表示失配结束标签；虚竖线表示分片位置。对于节点 7 和 8 组成的子树，根节点是 7，对应的开始标签‘7〈’的位置如左边箭头所示，根据规则 1，它的最接近的未配对失配开始标签是‘3〈’，故挂接节点是节点 3。对于节点 10—12 组成的子树，根节点是 10，对应的开始标签‘10〈’的位置如右边箭头所示，根据规则 1，由于在前半区间，存在已配对标签对 1,2,3，则最接近的未配对失配开始标签应该是‘0〈’，故挂接节点是节点 0。

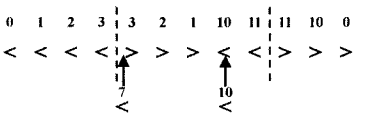

图 3 子树挂接点获取方法

4.4 节点内容合并和节点标签名合并

子树合并除了完成节点关系信息调整，还需要完成节点内容、标签名的合并工作。节点内容合并实现把各片段的节点 ID 和节点内容对照表合并入最终节点内容对照表。节点标签名合并实现把各片段的节点标签名合并入最终节点标签名 hash 表。通过重构节点标签名对照 hash 表完成合并。

5 实验与分析

为了验证解析方法在多核计算环境下的有效性。我们用 XMark 标准测试平台[11]，对不同文档尺寸和工作线程数量下的执行时间进行对比。用 XMark 的 XML 数据生成工具，设置参数 $-f$ 为 0.1，0.5，1 和 2，分别生成 11.5Mb，56Mb，115Mb 和 226Mb 4 个不同尺寸的 XML 文档。测试平台是 AMD Athlon II X4 620(2.60 Ghz) 4 核 PC，软件环境是用 Windows XP 系统运行 JDK1.6.0_18。图 4 是 4 个文档在不同工作线程数量下的执行解析总时间比较。很明显随着工作线程数量的增加，各文档的解析性能都有效地获得了提高。

(下转第 235 页)

[6] Chen Y X, Wang J Z, Krovetz R. CLUE: cluster-based retrieval of images by unsupervised learning[J]. IEEE Transactions on Pattern Analysis and Machine Intelligence, 2005, 14(8): 1187-1201
[7] Tong S, Chang E. Support vector machine active learning for image retrieval [C] // Proc. ACM Int. Conf. Multimedia. Ottawa, Canada: ACM Press, 2001: 107-118
[8] Brinker K. Incorporating diversity in active learning with support vector machines [C] // Proc. Int. Conf. Machine Learning. Washington, DC: AAAI Press, 2003: 59-66
[9] Wang L, Chan K L, Zhang Z. Bootstrapping SVM active learning by incorporating unlabelled images for image retrieval [C] // Proc. IEEE Int. Conf. Computer Vision and Pattern Recognition. Madison, Wisconsin: IEEE Press, 2003: 629-634
[10] Hoi S C H, Rong J, Zhu J K, et al. Semi-supervised SVM batch mode active learning and its applications to image retrieval [J]. ACM Transactions on Information Systems, 2009, 27(3): 1-29
[11] Huang T S, Dagli C K, Rajaram S, et al. Active learning for interactive multimedia retrieval [J]. Proceedings of IEEE, 2008, 96(4): 648-667
[12] Gosselin P H, Cord M. Active learning methods for interactive image retrieval [J]. IEEE Transactions on Image Processing, 2008, 17(7): 1200-1211
[13] Elkan C. Using the triangle inequality to accelerate k-means [C] // Proc. Int. Conf. Machine Learning. Washington, DC: AAAI Press, 2003
[14] http://www.csie.ntu.edu.tw/~cjlin/libsvm

(上接第 194 页)

在 4 个工作线程条件下，11.5Mb，56Mb，115Mb 和 226Mb 对应的平均解析总时间分别是 543ms，2323ms，4312ms 和 10810ms，反映了并行时，执行时间与数据大小接近线性比例，说明该方法有较好的扩展性。

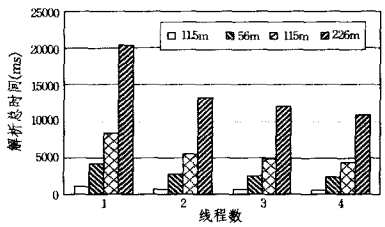

图 4 解析执行时间对比

通过解析执行时间计算加速比。以原有串行处理方法的解析时间 T_1 为基准，在 n 个线程下的解析时间为 T_n，则这时的加速比 $S_n = T_n / T_1$。对不同文档大小，4 个工作线程条件下，加速比接近 2。由于多线程子树构建过程中，存在并发存取问题，在大数据量条件下尤为突出，因此目前未进行数据加载优化，总体加速比未达理想状态，但反映了实际工作情况。

此外，我们测试了各种数据在多线程条件下各阶段时间的分布情况。图 5 给出了在 4 个工作线程下并行解析各阶段的耗时比例。比如 115Mb 数据的并行解析情况，其中并行子树构建的时间占总时间的 83.6%，而子树合并工作只占 13.9%，数据分片操作约占 2.5%。其他不同线程数量条件下的情况类似，子树构建的执行时间占大部分，这说明对其并行化处理的重要性。

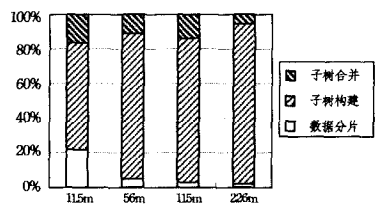

图 5 各阶段耗时比例

结束语 由于 XML 解析性能极大地影响着 XML 应用的整体性能，故 XML 解析优化是个研究重点。随着多核环境的普及，并行化是一个自然而有效的优化选择。相对现有的几种并行 XML 解析方法，本文的方法不需要进行耗时的预处理，而是通过对 XML 文档任意分片，对各数据分片并行处理，构建局部文档子树，最后合并成全局文档树。其关键是对耗时最多的分片解析进行数据并行。通过多核环境下的性能测试验证了整个方法的有效性。

未来工作考虑通过优化设计提高加速比，比如采用数据预读取方法避免并发存取竞争，考虑为每个片段处理串行加载文档的对应部分，以减少文档存取操作对各片段并行解析的影响，实验表明这样串行加载的时间代价小。另一方面，通过合理控制任务粒度，获取较好的负载均衡效果。在工作线程较多的条件下，考虑把 XML 数据划分成更小的片段，采用工作窃取[12]的调度模式使负载均衡容易实现。

参 考 文 献

[1] Matthias N, Jasmi J. XML parsing: a threat to database performance [C] // International Conference on Information and Know-ledge Management, Proceedings. 2003: 175-178
[2] Akhter S, Roberts J, Reinders J, et al. Multi-core programing: increasing performance through software multi-threading [M]. Hillsboro: Intel Press Business Unit, 2004
[3] Lu K, Zhu Y, Sun W, et al. Parallel processing XML documents [C] // Proceedings of International Database Engineering and Applications Symposium. 2002: 96-105
[4] Lu W, Chiu K, Pan Y. A parallel approach to XML parsing [C] // 7th IEEE/ACM International Conference on Grid Computing. 2006: 8-16
[5] Pan Y, Lu W, Zhang Y, et al. A static load-balancing scheme for parallel XML parsing on multicore CPUs [C] // Seventh IEEE International Symposium on Cluster Computing and the Grid (CCGrid '07). 2007: 356-365
[6] Pan Y, Zhang Y, Chiu K, et al. Parallel XML parsing using meta-DFAs [C] // 2007 3rd IEEE International Conference on e-Science and Grid Computing. 2008: 237-244
[7] Pan Y, Zhang Y, Chiu K. Simultaneous transducers for data-parallel XML parsing [C] // Proceedings of the 2008 IEEE International Parallel & Distributed Processing Symposium. 2008: 1143-1154
[8] Pan Y, Zhang Y, Chiu K. Hybrid parallelism for XML SAX parsing [C] // Proceedings of the IEEE International Conference on Web Services, ICWS 2008. 2008: 505-512
[9] Bruno N, Koudas N, Sprvastava D. Holistic twig joins: optimal XML pattern matching [C] // Proceedings of SIGMOD. 2002: 310-321
[10] Timothy M, Beverly S, Berna M. Patterns for parallel programming [M]. Massachusetts: Addison Wesley/Pearson, 2004
[11] CWI. An XML benchmark project [CP/OL]. http://www.xml-benchmark.org, 2009
[12] Lea D. A Java fork/join framework [C] // Proceedings of the ACM 2000 conference on Java Grande. 2000: 36-43