ITAT 2013 Proceedings, CEUR Workshop Proceedings Vol. 1003, pp. 75–81
http://ceur-ws.org/Vol-1003, Series ISSN 1613-0073, © 2013 Z. Falt, M. Kruliš, J. Yaghob

Bobolang – jazyk pro systém Bobox

Zbyněk Falt, Martin Kruliš, Jakub Yaghob*

Univerzita Karlova, Praha, Česká republika,
{falt,krulis,yaghob}@ksi.mff.cuni.cz

Abstrakt: Paralelní zpracování dat je v současné době velmi aktuální téma. Jeden z používaných postupů je převod vstupních dat na datové proudy a zpracování těchto proudů pomocí operátorů, které mohou být vyhodnocovány paralelně. Protože pro specifikaci vzájemného propojení operátorů jsou běžné programovací jazyky nevhodné, vznikla pro tento účel celá řada doménově specifických jazyků. Jazyk Bobolang je jedním z nich. Kromě běžných vlastností, ale přidává navíc některé syntaktické a sémantické prvky, které značně usnadňují intra-operátorovou paralelizaci. Díky tomu je možné snadno vytvářet vysoce škálovatelné aplikace zpracovávající proudová data.

1 Úvod

Paralelní systémy jsou v dnešní době téměř standardem. Z tohoto důvodu se neustále hledají nové cesty jak co nejefektivněji využít tyto prostředky. Bohužel, vývoj paralelních aplikací je náročný a náchylný k chybám. Proto vznikají různé knihovny, nástroje, systémy a metody, jak tuto činnost maximálně usnadnit a zefektivnit. Některé tyto nástroje se snaží být co nejobecnější, tzn. programátorovi pomáhají s vytvářením a synchronizací vláken [5], některé poskytují množství knihovních funkcí pro snazší paralelizaci některých typů algoritmů [19], některé rozšiřují samotný jazyk o direktivy určené pro snazší vývoj [7]. Kromě toho existují doménově specifické nástroje, které jsou určeny pro konkrétní druhy aplikací. Systémy pro zpracování proudových dat jsou jedním z nich [20, 18, 21, 8, 4, 17, 2].

Tyto systémy pracují s tzv. datovými proudy, tj. v podstatě s posloupnostmi n-tic. Tyto proudy jsou průběžně (tj. tak jak n-tice přichází) zpracovávány operátory, které transformují vstupní proudy na výstupní. Vývoj aplikací pro takové systémy se skládá ze dvou fází:

1. Implementace potřebné množiny operátorů, tj. jak přetransformovat vstupní proud/proudy na výstupní proud/proudy.
2. Vytvoření exekučního plánu, tj. pospojování operátorů orientovanými hranami, které určují datové proudy mezi nimi.

Zatímco implementaci operátorů lze provést v téměř libovolném běžném programovacím jazyku (C/C++, Java, C# apod.), pro specifikaci exekučního plánu se tyto jazyky příliš nehodí. Exekuční plán totiž odpovídá orientovanému grafu, tj. je zadán seznamem vrcholů a hran. To sice lze snadno vyjádřit v běžných programovacích jazycích, ale takový kód je obtížně čitelný a modifikovatelný. Proto vznikají jazyky, které vytváření exekučního plánu usnadňují, ať už speciální syntaxí, nebo grafickou vizualizací plánu.

Bobolang se zaměřuje na druhou fázi vývoje. Kromě čitelné syntaxe se ale snaží i o pomoc při intra-operátorové paralelizaci, kdy provádí některé transformace zvyšující paralelismus exekučního plánu automaticky. Tím se odlišuje od ostatních podobných jazyků.

Zbytek článku je rozdělen následovně: Kapitola 2 představuje používané jazyky určené pro systémy zpracování proudových dat, kapitola 3 popisuje systém Bobox, pro který vznikla pilotní implementace jazyka Bobolang. V kapitole 4 rozebíráme možnosti a postupy paralelizace operátorů. Stručný popis jazyku Bobolang je uveden v kapitole 5 a několik příkladů jeho použití uvádíme v kapitole 6. Celý článek pak shrnujeme v kapitole 7.

2 Související práce

Současné jazyky zaměřující se na proudové zpracování dat se dají rozdělit do několika skupin podle jejich zaměření.

Brook [3, 4], StreaMIT [20] a StreamC [9] se zaměřují na vývoj vysoce výkonných aplikací pracujících převážně s multimediální daty (kodeky, filtry, transformace apod.). Tyto jazyky jsou založeny na syntaxi jazyka C/C++ a pokrývají jak fázi implementace jednotlivých operátorů, tak jejich vzájemné propojení ve výsledné aplikaci. Překladače těchto jazyků provádějí některé optimalizace, které zvyšují výkon nebo nebo mapují operátory na určité výpočetní jednotky systému (CPU, GPU, FPGA[1]).

Lucid [1] je další jazyk určený pro programování proudových aplikací. Tento jazyk sám o sobě není určen pro paralelní aplikace, proto vznikl jazyk Granular Lucid (GLU) [16], který umožňuje do plánu zařadit operátory implementované v jazyku C, které mohou být pouštěny paralelně.

Jazyk X-Language [14] je moderní jazyk vyvinutý pro systém Auto-Pipe [6]. Tento jazyk slouží pro vzájemné propojení již připravených operátorů. Svým charakterem je tedy podobný jazyku GLU, ale propojení operátorů

*Článek byl podporován Grantovou agenturou Univerzity Karlovy, projekt č. 277911, Grantovou agenturou ČR GACR P103/13/08195S a grantem SVV-2013-267312.

[1] Field-programmable gate array

vyjádřeno explicitněji (syntaxe je podobná jazyku Bobolang). Podporuje rovněž vytváření operátorů z již existujících operátorů. Na rozdíl od Bobolangu ale nedochází k automatické modifikaci plánu za účelem zvýšení paralelismu.

3 Bobox

Systém Bobox je jedna z implementací systému pro zpracování proudových dat. Bobox poskytuje běhové prostředí pro vyhodnocování exekučních plánů v paralelním prostředí. Systém podporuje jak acyklické exekuční plány, tak i plány obsahující cykly. Protože jazyk Bobolang je navržen právě pro Bobox a využívá některé jeho vlastnosti, uvádíme v této kapitole některé technické detaily tohoto systému.

Datové proudy v Boboxu jsou reprezentované jako proud tzv. datových obálek. Každá obálka obsahuje seznam tzv. datový sloupců. Tyto datové sloupce obsahují samotná data. Každý sloupec musí obsahovat data pouze jednoho typu, ale jedna obálka může obsahovat sloupce různých typů. Dále platí, že všechny sloupce v jedné obálce mají vždy stejnou délku, takže se můžeme na obálku dívat jako na posloupnost n-tic. Kromě datových obálek existují tzv. otrávené obálky, jejichž úkolem je signalizovat konec datového proudu.

V současné době podporuje Bobox pouze shared-memory systémy, takže operátory si mohou vzájemně posílat pouze ukazatele na obálky. Tato implementace značně urychluje operátory jako např. `broadcast` (viz kapitola 4), neboť data nejsou nijak klonována a v paměti se nacházejí pouze v jedné instanci. Navíc je celkový počet řádků v obálce je zvolen s ohledem na velikost vyrovnávacích pamětí v procesoru, tak aby komunikace mezi operátory probíhala bez nutnosti přístupu do hlavní paměti.

Každý exekuční plán, musí obsahovat dva speciální operátory:

- `init` – tento operátor je vždy první v topologickém uspořádání exekučního plánu. Jeho úkolem je nastartovat výpočet tím, že na svůj výstup odešle otrávenou obálku.
- `term` – tento operátor je vždy poslední v topologickém uspořádání. Ve chvíli, kdy přijme na svém vstupu otrávenou obálku, oznámí systému, že exekuční plán byl vyhodnocen.

Důležitou součástí systému je plánovač, jehož úkolem je přidělovat výpočetní čas jednotlivým operátorům. Obecně se plánovač řídí dostupností datových obálek na vstupech operátorů, tj. pokud má operátor neprázdnou frontu vstupních obálek, je vložen do fronty operátorů připravených ke spuštění. Na základě různých kritérií [13] vybírá plánovač z této fronty operátory a spouští jejich kód v některém z připravených vláken. Důležité je, že jeden operátor může být spuštěn v nejvýše jednom vlákně. Toto omezení má dva důsledky:

- Programátor operátorů nemusí řešit synchronizaci vláken, takže vývoj operátorů je značně usnadněn.
- Pro zpracování jedné obálky nelze použít více než jedno vlákno, což zdánlivě omezuje možnosti paralelizace.

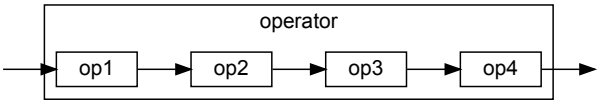
Ale i přes jednovláknovost operátorů, je možné dosáhnout paralelního vyhodnocování exekučního plánu, neboť nezávislé operátory mohou být spouštěny paralelně. Rozlišujeme tři typy paralelismů [15]:

- pipelinový paralelismus – zdroj datového proudu pracuje paralelně s jeho konzumentem
- taskový paralelismus – nezávislé datové proudy jsou zpracovávány paralelně
- datový paralelismus – nezávislé části jednoho proudu mohou být zpracovány paralelně

Taskový paralelismus je pevně zakódovaný v exekučním plánu, takže přináší pouze omezenou škálovatelnost. Datový a pipelinový paralelismus lze ale za určitých okolností zvýšit a tím zvýšit škálovatelnost celého systému.

4 Intra-operátorový paralelismus

Pipelinový paralelismus můžeme zvýšit (resp. zavést) tím, že určitý operátor rozdělíme na posloupnost dílčích operátorů, kde každý vykoná nad proudem část práce (viz Obrázek 1). Bohužel ne všechny operátory lze takto dekomponovat a u těch, u kterých to lze, to může být nevýhodné z důvodu zvýšení režie nutné pro přenos datového proudu.



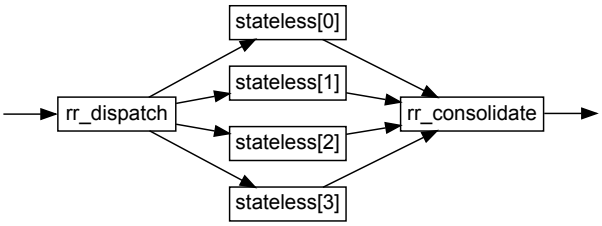
Obrázek 1: Rozklad operátoru pro zvýšení pipelinového paralelismu.

Datový paralelismus můžeme do plánu zavést tak, že vstupní proud rozdělíme na několik dílčích proudů, ty zpracujeme paralelně a poté je opět spojíme do výsledného proudu. V následujících dvou podkapitolách rozebereme dva postupy, jak toho dosáhnout.

4.1 Bezestavové operátory

Bezestavové operátory si neudržují vnitřní stav. To znamená, že zpracování jedné n-tice je zcela nezávislé na ostatních. Typickou ukázkou je např. operátor `filter`, který z proudu n-tic odstraní ty, které nesplňují určitou podmínku, neboť vyhodnocení podmínky pro jednu n-tici nezávisí na jiných n-ticích.

Protože zpracování jedné n-tice nezávisí na ostatních, nezávisí ani zpracování celé obálky na ostatních obálkách. Můžeme tedy použít schéma naznačené na Obrázku 2. Operátor `rr_dispatch` jednoduše přeposílá vstupní obálky na své výstupy metodou round-robin, operátor `rr_consolidate` metodou round-robin odebírá výsledné obálky z jednotlivých operátorů a vytváří tak výsledný proud.



Obrázek 2: Paralelizace bezestavového operátoru.

Protože `rr_dispatch` a `rr_consolidate` pouze manipulují se ukazateli na obálky (viz kapitola 3), pracují oba operátory velmi rychle a celý výpočet zpomalují pouze zanedbatelně.

4.2 Paralelizovatelné operátory

Se stavovými operátory je situace složitější, neboť abychom zpracovali jednu obálku, musíme znát stav odvozený z obsahu předchozích obálek. U některých operátorů můžeme použít postup naznačený v této podkapitole. Předpokládejme, že tělo stavového operátoru vypadá obecně takto:

$S \leftarrow$ iniciální stav
while not konec **do**
  zpracuj vstupní data pomocí S a zároveň aktualizuj S
end while

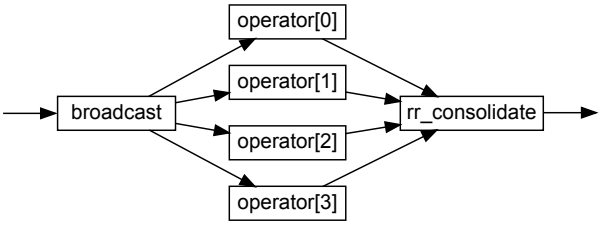
Občas ale lze toto schéma upravit do následující podoby:

$S \leftarrow$ iniciální stav
while not konec **do**
  zpracuj vstupní data pomocí S
  aktualizuj S
end while

Pokud je aktualizace stavu S rychlejší než zpracování dat, můžeme vytvořit N paralelních operátorů a očíslovat je čísly 0 až $N-1$ (toto číslo budeme v dalším textu označovat jako *PID* – Parallel ID). Každý operátor pak bude pracovat následujícím způsobem:

$S \leftarrow$ inciální stav
fáze $\leftarrow 0$
while not konec **do**
  if fáze $\bmod N = PID$ **then**
    zpracuj část vstupu pomocí S
  end if
  aktualizuj S
  fáze $\leftarrow$ fáze + 1
end while

V tuto chvíli se operátory střídají ve zpracování vstupních dat, takže pro paralelizaci můžeme použít schéma znázorněné na Obrázku 3. Efektivita paralelizace závisí na složitosti aktualizace stavu. Je zřejmé, že by měla být alespoň N-krát rychlejší než zpracování dat. Problém nastává, pokud je aktualizace stavu netriviální, neboť v takovém případě se počítá N-krát totéž. Řešením je dedikovat samostatný operátor pro aktualizaci stavu, který by všem ostatním posílal aktuální stav.



Obrázek 3: Paralelizace paralelizovatelného operátoru.

Ukázka paralelizovatelného operátoru Velmi jednoduchý příklad operátoru, který lze paralelizovat schématem popsaným v části 4.2 je defragmentace obálek. Některé operátory generují obálky mnohem menší než doporučené velikosti. To má za následek snížení výkonu systému, neboť příliš malé obálky zvyšují celkovou režii potřebnou na plánování operátorů.

Má-li obálka doporučenou velikost L n-tic, je základní algoritmus následující:

while not konec **do**
  překopíruj a přeskoč L n-tic do výstupní obálky
end while

Podle výše uvedeného postupu, můžeme kód operátoru upravit do následující podoby:

fáze $\leftarrow 0$
while not konec **do**
  if fáze $\bmod N = PID$ **then**
    překopíruj L n-tic do výstupní obálky
  end if
  přeskoč L n-tic
  fáze $\leftarrow$ fáze + 1
end while

Protože přeskakování n-tic je velmi rychlá operace (můžeme přeskakovat celé obálky nebo jejich části), je tato paralelizace velmi účinná.

5 Bobolang

5.1 Úvod

Jazyk Bobolang vznikl pro účely pohodlnějšího zápisu exekučních plánů. Tomu odpovídá syntaxe, kdy programátor vyrobí instance operátorů (podobně jako se vytváří proměnné v jazycích C/C++) a poté je pomocí operátoru `->` vzájemně pospojuje.

Pomocí jazyka je rovněž možné z množiny hotových operátorů (naimplementovaných v jazyku C++ nebo v jazyku Bobolang) poskládat samostatný operátor, který lze poté použít v exekučním plánu. K tomu slouží následující syntaxe:

```
operator process(int)->(int)
{
  prepropcess(int)->(int) pre;
  postprocess(int)->(int) post;

  input -> pre;
  pre -> post;
  post -> output;
}
```

Řádek `operator process(int)->(int)` říká, že chceme vytvořit operátor se jménem `process`, který transformuje proud celých čísel na proud celých čísel. Následuje tělo operátoru, které se skládá z instancí operátorů `preprocess` a `postprocess`. Kromě explicitně uvedených instancí operátorů (`pre` a `post`), obsahuje každé tělo implicitně operátory `input` a `output`. Ty reprezentují vstup/výstup celého operátoru. Takže řádek `input -> pre` říká, že vstup operátoru `process` je přeposílán na vstup operátoru `pre`. Podobně funguje operátor `output`.

Exekuční plán se specifikuje stejnou syntaxí. Jak bylo uvedeno v kapitole 3, exekuční plán se skládá ze dvou speciálních operátorů `init` a `term` a těla exekučního plánu. Na tělo exekučního plánu se tedy můžeme dívat jako na operátor, který má jeden vstup (k němu je připojen operátor `init`) a jeden výstup (k němu je připojen operátor `term`).

Aby interpretr jazyka poznal, který operátor reprezentuje exekuční plán, musí být vždy pojmenován jako `main`.

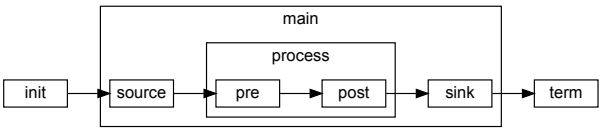
Pokud bychom chtěli vyrobit aplikaci, která zpracovává posloupnost celých čísel, napíšeme následující kód:

```
operator main()->()
{
  source()->(int) source;
  process(int)->(int) op;
  sink(int)->() sink;

  input -> source;
  source -> op;
  op -> sink;
  sink -> output;
}
```

Pokud předáme systému Bobox tento kód, interpretr Bobolangu instanciuje operátor `main`, vytvoří operátory `init` a `term` a vytvoří exekuční plán, který je znázorněný na Obrázku 4.

Pokud má operátor více vstupů/výstupů, jsou tyto operátory číslovány od nuly a číslo vstupu/výstupu musí být uvedeno. Pokud má vstup/výstup pouze jeden, nemusí být toto číslo uvedeno. Viz např. použití operátoru `merge`, kdy

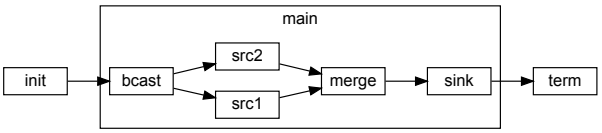


Obrázek 4: Ukázka plně instanciovaného exekučního plánu.

navíc musí rozeslat otrávenou obálku z operátoru `init` do operátorů `source` (viz Obrázek 5).

```
operator main()->()
{
  broadcast()->(),() bcast;
  source()->(int) src1, src2;
  merge(int),(int)->(int) merge;
  sink(int)->() sink;

  input -> bcast;
  bcast[0] -> src1 -> [0]merge;
  bcast[1] -> src2 -> [1]merge;
  merge -> sink;
  sink -> output;
}
```



Obrázek 5: Ukázka práce s operátory, které mají více vstupů/výstupů.

5.2 Násobnost vstupů/výstupů

Každý operátor může mít libovolný nenulový počet vstupů a výstupů. Kromě toho ale může být každý vstup/výstup tzv. násobný. Implicitně je každý vstup/výstup jednonásobný, násobnost se musí zapisovat explicitně, tj. např.:

```
broadcast()->(){N} bcast;
```

kde *N* značí násobnost. *N* může být buď přirozené číslo nebo znak ∗. Číslo *N* přesně určuje násobnost vstupu/výstupu, zatímco ∗ nechává toto rozhodnutí na Bobolangu, který dosadí vhodné číslo (v pilotní implementaci shodné s počtem vláken v systému).

Bobolang umožňuje vzájemně propojit výstup libovolné násobnosti na vstup libovolné násobnosti, pokud taková operace nevede k logické chybě.

Pokud je připojen vícenásobný výstup na jednonásobný vstup, dojde k automatické replikaci cílového operátoru podle násobnosti výstupu a připojení výstupů na jednotlivé vstupy replikovaných operátorů.

Spojení jednonásobného výstupu s jednonásobným vstupem způsobí, že cílový operátor je replikovaný právě

tolikrát, kolikrát je replikovaný zdrojový operátor a jednotlivé výstupy jsou napojeny na jednotlivé vstupy.

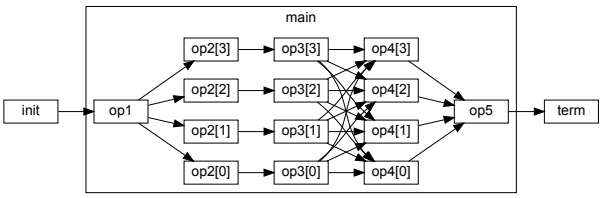
Aby spojení jednonásobného výstupu na vícenásobný vstup bylo korektní, musí být zdrojový operátor replikovaný. Pokud je tato podmínka splněna, je vytvořena jedna instance cílového operátoru a jednotlivé výstupy jsou připojeny na vstup tohoto operátoru.

Spojení vícenásobného výstupu a vícenásobného vstupu je rovněž povoleno, pokud je zdrojový operátor replikovaný. V takovém případě je cílový operátor replikovaný tolikrát, kolikrát je replikovaný zdrojový operátor. V případě operátoru `->` je 1. operátor napojen na 1. podvstup cílových operátorů, 2. operátor na 2. podvstup, atd.

Následující zdrojový kód, který pokrývá všechny uvedené možnosti, bude interpretován tak, jak je znázorněno na Obrázku 6.

```
operator main()->()
{
  op()->(){*} op1;
  op()->() op2;
  op()->(){*} op3;
  op(){*}->() op4;
  op(){*}->() op5;

  input -> op1 -> op2 -> op3;
  op3 -> op4 -> op5 -> output;
}
```



Obrázek 6: Ukázka exekučního plánu s násobnými vstupy/výstupy.

5.3 Klíčové slovo `typename`

Aby bylo možné vytvářet znovupoužitelné operátory (např. operátor třídící celá a desetinná čísla bude mít pravděpodobně identickou vnitřní strukturu), obsahuje Bobolang klíčové slovo `typename`. To je inspirované stejným slovem v jazyku C++ a je možné jej použít v deklaraci operátoru např. v případě třídění takto:

```
operator sort(typename T)->(T)
{
  some_operator(T)->(T) op;
  ...
}
```

V těle operátoru pak můžeme používat typ `T`, jako jakýkoliv jiný běžný typ. Pokud instanciujeme tento operátor např. následujícím způsobem:

```
sort(int,int)->(int,int)
```

dosadí se za `T` typ `(int,int)`, tj. dvojice celých čísel. Pokud by vstupní a výstupní typ byl různý, dojde k chybě při vyhodnocování.

5.4 Intra-operátorová paralelizace

Zápis intra-operátorové paralelizace je nyní snadný. Pokud máme bezestavový operátor, stačí zapouzdřit jej následovně:

```
operator parallel_stateless
  (typename T)->(typename U)
{
  rr_dispatch(T)->(T){*} disp;
  stateless_operator(T)->(U) op;
  rr_consolidate(U){*}->(U) cons;

  input -> disp -> op;
  op -> cons -> output;
}
```

Instanciací operátoru dostaneme stejné schéma jako v podkapitole 4.1 (viz Obrázek 2).

Paralelizovatelný operátor má identické schéma, jenom místo operátoru `rr_dispatch`, použijeme operátor `broadcast`. Aby programátor nemusel bezestavové a paralelizovatelné operátory takto paralelizovat ručně, provádí tuto úpravu Bobolang sám. Stačí označit operátor jako bezestavový (klíčovým slovem `stateless`) nebo paralelizovatelný (klíčovým slovem `parallel`). V ostatních případech se žádná modifikace neprovádí.

Pokud se jedná o komplexnější paralelizaci operátoru, je nutné zapsat schéma operátoru ručně, nicméně Bobolang tuto činnost značně usnadňuje, viz kapitola 6.

6 Příklady aplikací

6.1 Nested-loops join

Nested-loops join je velmi snadný algoritmus pro paralelizaci. Máme-li naimplementovaný operátor, který vykonává nested-loops join nad vstupními daty, můžeme paralelizovat operátor tak, že vytvoříme N instancí toho operátoru a do jednoho vstupu operátorů přepošleme celý první vstup, zatímco do druhého pouze jednu N-tinu druhého vstupu (N-tiny musí být samozřejmě disjunktní). Výsledný proud pak získáme jako sjednocení výsledku replikovaných operátorů.

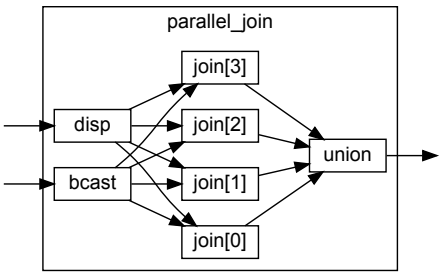
V Bobolangu tento algoritmus zapíšeme následujícím způsobem (`dispatch` má z úkol rozdělit proud na N částí, `union` spojit N proudů do jednoho)

```
operator parallel_join
  (typename L),(typename R)
  -> (typename T)
{
  broadcast(L)->(L){*} bcast;
```

```
  rr_dispatch(R)->(R){*} disp;
  nested_loops_join(L),(R)->(T) join;
  union(T){*}->(T) union;

  input[0] -> bcast -> [0]join;
  input[1] -> disp -> [1]join;
  join -> union -> output;
}
```

Instanciovaný operátor je zobrazen na Obrázku 7. Více detailů včetně experimentů lze nalézt v článku [10].



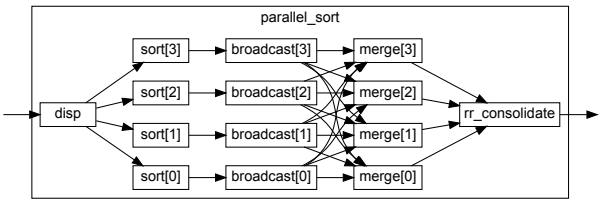
Obrázek 7: Paralelizovaný nested-loops join.

6.2 Třídění

Problémem třídění v systémech proudového zpracování dat jsme se zabývali v předchozí práci [11]. Základní ideou algoritmu je rozdělit vstupní proud na několik podproudů, ty setřídit paralelně a tyto setříděné podproudy paralelně slít. Tato myšlenka vede k následujícímu kódu:

```
operator parallel_sort(typename T)->(T)
{
  rr_dispatch(T)->(T){*} disp;
  sort(T)->(T) sort;
  parallel merge(T){*}->(T) merge;
  input -> disp -> sort;
  sort -> merge -> output;
}
```

Protože je `merge` označen jako `parallel`, vloží se před tento operátor automaticky `broadcast` a za něj `rr_consolidate`. Pokud použijeme operátor `parallel_sort` v exekučním plánu, rozvine se do tvaru znázorněného na Obrázku 8.



Obrázek 8: Paralelizovaný třídicí operátor.

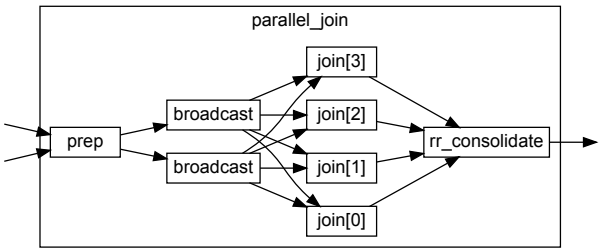
6.3 Merge join

Základní myšlenkou paralelního merge joinu pro systém Bobox je modifikovat a vzájemně párovat vstupní obálky tak, aby bylo možné spojovat tyto páry paralelně. To vede k následujícímu schématu:

```
operator parallel_join
  (typename L),(typename R)
  -> (typename T)
{
  preprocess(L),(R)->(L),(R) prep;
  parallel join(L),(R)->(T) join;

  input[0] -> [0]prep[0] -> [0]join;
  input[1] -> [1]prep[1] -> [1]join;
  join -> output;
}
```

Instanciovaný operátor je znázorněn na Obrázku 9. Bližší podrobnosti včetně detailní implementace operátoru `preprocess` a `join` lze nalézt v článku [12].



Obrázek 9: Paralelizovaný merge join.

7 Závěr a budoucí práce

V tomto článku jsme představili jazyk Bobolang, který je určený pro použití v systémech pro zpracování proudových dat. Kromě specifikace exekučních plánů má vlastnosti, které umožňují snadno popsat vnitřní strukturu paralelizovaných operátorů. Interpret jazyka na základě těchto popisů instanciuje exekuční plán tak, aby při jeho vyhodnocování v paralelním prostředí k maximálnímu využití hardwarových prostředků. Uvedli jsme i několik příkladů jeho reálných aplikací.

Do budoucna plánujeme rozšířit Bobolang tak, aby podporoval rovněž distribuované systémy. Bude tedy možné snadno specifikovat, jak rozdistribuovat exekuční plán mezi více uzlů, případně nechat interpret jazyka rozdistribuovat plán automaticky.

Reference

[1] E.A. Ashcroft, A.A. Faustini, R. Jagannathan, and W.W. Wadge. *Multidimensional programming*. Oxford University Press, 1995.

[2] David Bednarek, Jiri Dokulil, Jakub Yaghob, and Filip Zavoral. Bobox: Parallelization Framework for Data Processing. In *Advances in Information Technology and Applied Computing*, 2012.

[3] Ian Buck. Brook: A streaming programming language, 2001.

[4] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. *ACM Transactions on Graphics*.

[5] David R Butenhof. *Programming with POSIX threads*. Addison-Wesley Professional, 1997.

[6] Roger D Chamberlain, Mark A Franklin, Eric J Tyson, James H Buckley, Jeremy Buhler, Greg Galloway, Saurabh Gayen, Michael Hall, EFBerkley Shands, and Naveen Singla. Auto-pipe: Streaming applications on architecturally diverse systems. *Computer*, 43(3):42–49, 2010.

[7] R. Chandra. *Parallel programming in OpenMP*. Morgan Kaufmann, 2001.

[8] Charles Consel, Hedi Hamdi, Laurent Réveillère, Lenin Singaravelu, Haiyan Yu, and Calton Pu. Spidle: A DSL approach to specifying streaming applications. In *Proceedings of the 2nd international conference on Generative programming and component engineering*, GPCE '03, pages 1–17, New York, NY, USA, 2003. Springer-Verlag New York, Inc.

[9] Abhishek Das, William J. Dally, and Peter Mattson. Compiling for stream processing. In *Proceedings of the 15th international conference on Parallel architectures and compilation techniques*, PACT '06, pages 33–42, New York, NY, USA, 2006. ACM.

[10] Zbynek Falt, David Bednarek, Miroslav Cermak, and Filip Zavoral. On Parallel Evaluation of SPARQL Queries. In *DBKDA 2012, The Fourth International Conference on Advances in Databases, Knowledge, and Data Applications*, pages 97–102. IARIA, 2012.

[11] Zbynek Falt, Jan Bulanek, and Jakub Yaghob. On Parallel Sorting of Data Streams. In *ADBIS 2012 - 16th East European Conference in Advances in Databases and Information Systems*, 2012.

[12] Zbynek Falt, Miroslav Cermak, and Filip Zavoral. Highly Scalable Sort-Merge Join Algorithm for RDF Querying. In *The Second International Conference on Data Management Technologies and Applications*, 2013. [accepted].

[13] Zbynek Falt and Jakub Yaghob. Task scheduling in data stream processing. In *Proceedings of the Dateso 2011 Workshop*, pages 85–96. Citeseer, 2011.

[14] M.A. Franklin, E.J. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer. Auto-pipe and the X language: A pipeline design tool and description language. In *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*. IEEE, 2006.

[15] Michael I. Gordon, William Thies, and Saman Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XII, pages 151–162, New York, NY, USA, 2006. ACM.

[16] Rangaswamy Jagannathan, Chris Dodd, and Iskender Agi. Glu: A high-level system for granular data-parallel programming. *Concurrency - Practice and Experience*, 9(1):63–83, 1997.

[17] Ujval J. Kapasi, William J. Dally, Scott Rixner, John D. Owens, and Brucek Khailany. Programmable stream processors. *IEEE Computer*, 36:282–288, 2003.

[18] William R. Mark, R. Steven, Glanville Kurt, Akeley Mark, and J. Kilgard. Cg: A system for programming graphics hardware in a c-like language. *ACM Transactions on Graphics*, 22:896–907, 2003.

[19] J. Reinders. *Intel threading building blocks*. O'Reilly, 2007.

[20] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*, pages 179–196. Springer, 2002.

[21] Dan Zhang, Zeng zhi Li, Hong Song, and Long Liu. A programming model for an embedded media processing architecture. In *SAMOS*, pages 251–261, 2005.