GB-500: Introdução a Workflows Científicos e suas Aplicações

Professores: Luiz Gadelha, Kary Ocaña

Programa de Verão do LNCC, 2017
Laboratório Nacional de Computação Científica

5 de abril de 2018

Modelos de Computação para Workflows Científicos

- Diversos modelos de computação são adotados por SGWCs:
  - Álgebra relacional: SciCumulus.

    Ogasawara, E. et al. (2011). An Algebraic Approach for Data-Centric Scientific Workflows. Proceedings of the VLDB Endowment, 4(12), 1328–1339. http://www.vldb.org/pvldb/vol4/p1328-ogasawara.pdf.
  - Modelo de atores: Kepler.

    Bowers, S., Ludascher, B. (2005). Actor-oriented design of scientific workflows. In Conceptual Modeling - ER 2005. - Lecture Notes in Computer Science (Vol. 3716, pp. 369–384). https://doi.org/10.1007/11568322_24.
  - Programação funcional: Swift.

    Wilde, M. et al. (2011). Swift: A language for distributed parallel scripting. Parallel Computing, 37(9), 633–652. https://doi.org/10.1016/j.parco.2011.05.005.

Programação funcional

- Paradigmas de programação:
  - Imperativo (C)
  - Funcional (Lisp, Haskell)
  - Lógico (Prolog)
- Paradigma ortogonal: orientação a objetos (pode ser combinado com os demais: C++, Scala, F-logic).

Martin Oderski. Functional Programming Principles in Scala. `https://www.coursera.org/learn/progfun1`.

Paul Chiusano, Rúnar Bjarnasson. Functional Programming in Scala. Manning, 2015.

Chris Okasaki. Purely Functional Data Structures. Cambridge University Press, 1998.

Programação funcional

- Na programação imperativa
  - modificam-se variáveis mutáveis através de atribuições;
  - são utilizados controles de fluxo.
- Paradigma imperativo vs. Modelo de von Neumann:
  - variáveis mutáveis $\Leftrightarrow$ células de memória
  - dereferenciamento de variáveis $\Leftrightarrow$ instrução *load*
  - atribuição de variáveis $\Leftrightarrow$ instrução *store*
  - estruturas de controle $\Leftrightarrow$ saltos

Programação funcional

- Problemas do paradigma imperativo:
  - escalabilidade (baixo nível de abstração).
  - como evitar especificar programas instrução por instrução.
- John Backus argumentava que necessitamos de abstrações de alto nível:
  - coleções,
  - polinômios,
  - formas geométricas,
  - cadeias de caracteres,
  - etc.
- E que *teorias* fossem desenvolvidas sobre estas abstrações.

J. Backus, Can programming be liberated from the von Neumann style?: a functional style and its algebra of programs, Communications of the ACM, vol. 21, pp. 613–641, aug 1978. http://doi.org/10.1145/359576.359579.

Programação funcional

- Teorias são dadas por:
  - um ou mais tipos de dados,
  - operações e leis sobre esses tipos de dados.
- Teorias excluem mutabilidade (mudar dados sem mudar a identidade dos mesmos) por definição.

Programação funcional

- A *programação funcional* pode ser definida por:
  - definição de teorias para dados e operações (definidas como funções),
  - evitar mutabilidade,
  - mecanismos para abstrair e compor as funções,
  - uma função pode ser definida em vários lugares e pode ser passada como argumento para outras funções.
- Definição estrita: programação sem variáveis mutáveis, atribuições, laços ou outros estruturas de controle imperativas.
- Definição no sentido amplo: ênfase em funções, que produzem valores que podem ser consumidos ou compostos.

H. Abelson, G. J. Sussman, and J. Sussman, Structure and Interpretation of Computer Programs. MIT Press, second ed., 1996.

Programação funcional

- Benefícios: modelo mais simples para formalizar e fazer inferências, melhor modularidade, bom modelo para paralelismo (de *multicore* a computação na nuvem).
- Funções não têm efeitos colaterais, calculam apenas um valor e retornam o resultado.
- Se uma função é chamada com os mesmos parâmetros, retorna os mesmos resultados (transparência referencial).
- Linguagens de programação funcionais oferecem meios para:
  - definir expressões primitivas,
  - combinar expressões,
  - abstrair expressões.

Eriksen, M. (2016). Functional at scale. Communications of the ACM, 59(12), 50–55. https://doi.org/10.1145/2980985

Parallel and Concurrent Programming in Haskell. Simon Marlow. O'Reilly, 2013.

Programação funcional

- Linguagens funcionais:
  - Lisp (1959), Scheme (1977), Clojure (2007),
  - ML (1975), Standard ML (1986), OCaml (2000), F# (2005),
  - Haskell (1990),
  - Erlang (1990),
  - Scala (2003).

Swift

- Swift (`http://www.swift-lang.org`) permite gerenciar workflows científicos em ambientes paralelos e distribuídos.
- Usabilidade e produtividade com abstrações de alto nível para especificação e paralelização de workflows científicos.
- Alta taxa de execução de tarefas (500 tps no Swift/K e 500 Ktps no Swift/T).

M. Wilde, M. Hategan, J. Wozniak, B. Clifford, D. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):634-652, 2011.

Wozniak, J. M. et al. (2013). Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing. Proc. 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing (CCGrid 2013), pp. 95–102.

Armstrong, T. G. et al. (2014). Compiler Techniques for Massively Scalable Implicit Task Parallelism. In Proc. International Conference for High Performance Computing, Networking, Storage and Analysis (SC 2014), pp. 299–310.

Programming in the small × *programming in the large*

- *Programming in the small*:
  - Desenvolvimento de módulos relativamente "pequenos", com escopo bem-definido, como aplicações científicas e conversores de formato de dados.
  - Geralmente utiliza linguagens como C/C++, Fortran e R.
- *Programming in the large*:
  - Utilização de linguagens para interconexão dos módulos, que especificam a interação entre os mesmos para a construção de sistemas mais complexos.
  - Caso típico de utilização de sistemas de gerenciamento de workflows científicos, como o Swift.

De Remer, F., Kron, H. (1975). Programming-in-the large versus programming-in-the-small. ACM SIGPLAN Notices, 10(6), 114–121.

Programação Funcional e Workflows Científicos

- *Programação funcional* é um paradigma de programação onde a computação é orientada à definição e avaliação de funções.
- *Efeitos colaterais* (p.ex. mudanças de estado de variáveis) são desencorajados.

Swift

- É composto por:
  - Uma linguagem de alto nível para especificação de workflows científicos como scripts.
  - Ambiente de execução com suporte nativo a:
    - execução local,
    - execução paralela (p.ex. PBS, SGE),
    - execução distribuída (p.ex. SSH, Condor, Globus).
  - Sistema de gerência de proveniência.

Gadelha, L. M. R., Wilde, M., Mattoso, M., Foster, I. (2012). MTCProv: a practical provenance query framework for many-task scientific computing. Distributed and Parallel Databases, 30(5-6), 351–370.

Swift

- Desenvolvido no *Argonne National Laboratory* e *University of Chicago*
- Há uma colaboração com o LNCC no desenvolvimento do componente de gerência de proveniência.
- Exemplos de workflows científicos implementados no LNCC:
  - swift-blast (alinhamento paralelo de sequências).

    `https://github.com/lgadelha/swift-blast`
  - SwiftGECKO (genômica comparativa).

    `https://github.com/mmondelli/swift-gecko`
  - sdm-workflows (modelagem de distribuição de espécies).

    `https://github.com/sibbr/sdm-workflows`

Swift

- Cenário típico: composição de várias tarefas (*many-task computing*).
- As tarefas podem ser:
  - fracamente acopladas (*bag of tasks*),
  - fortemente acopladas:
    - locais (p.ex. paralelismo com memória compartilhada),
    - distribuídas (p.ex. paralelismo com memória distribuída).

Swift

- Paralelismo implícito:
  - Toda expressão de um script é avaliada em paralelo, exceto quando há dependências de dados.
  - $\Rightarrow$ Não é possível assumir uma ordem de execução dessas expressões.
- Independência de localidade:
  - Os scripts não expressam transferências de dados e seleção de sítios de execução.

Swift: Linguagem de Especificação de Workflows

- Aplicações que compõem um workflow são associadas a funções (*app functions*) em um script Swift.
- Variáveis em um script Swift podem ser associadas a:
  - tipos de dados primitivos,
  - dados persistentes armazenados em arquivos,
  - tipos de dados compostos (coleções).
- Suporte a laços e controles condicionais de fluxo.

Swift: Linguagem de Especificação de Workflows

- Sintaxe semelhante ao C, Java.
- Estrutura geral de um script Swift:
  1. Declarações de tipos de dados.
  2. Declarações e atribuições de variáveis.
  3. Declarações e chamadas de funções associadas a aplicações.
  4. Declarações e chamadas de funções compostas.

Swift: Linguagem de Especificação de Workflows

- Semelhanças com paradigma funcional:
  - Computação orientada a avaliação de funções.
  - Chamadas sem *efeitos colaterais*.
  - *Avaliação preguiçosa* de expressões.
  - Variáveis de *atribuição única*.

Swift: Linguagem de Especificação de Workflows

Exemplo:

```
type query;
type output;
string num_partitions=@arg("n", "8");
...
app (output o) blastapp(query i, fastaseq d, string p, string e, string f,
                        database db)
{
  blastall "-p" p "-i" @filename(i) "-d" @filename(d) "-o" @filename(o)
           "-e" e "-T" "-F" f;
}
...
foreach part,i in partition {
  formatdbout[i] = formatdb(part);
  out[i]=blastapp(query_file, part, program_name, expectation_value,
                  filter_query_sequence, formatdbout[i]);
}
```

Workflow especificado em Swift (`http://www.swift-lang.org`)

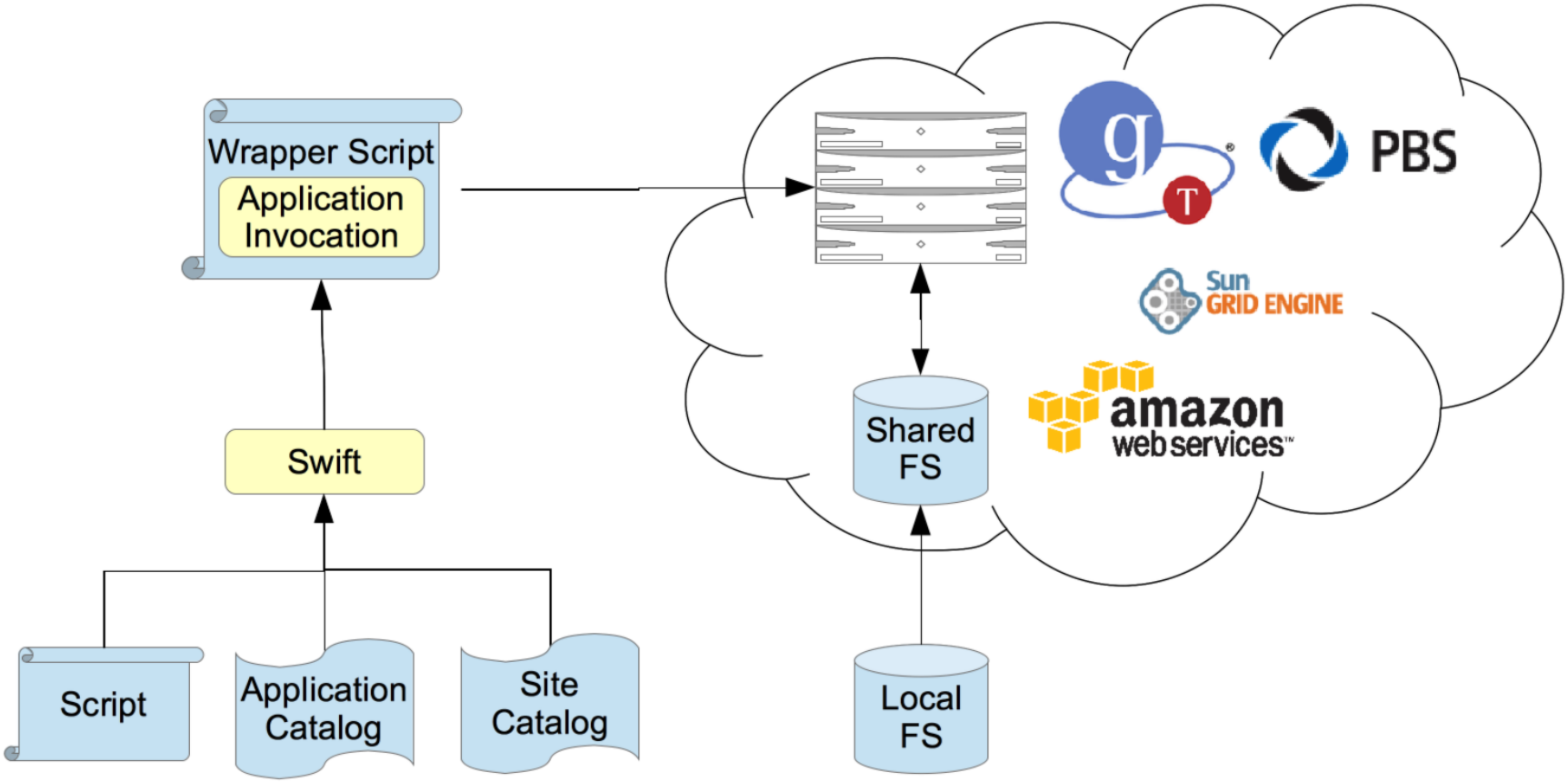
Swift: Ambiente de Execução

- Execução independente de localidade.
- Paralelização automática de execuções de aplicações componentes que não tenham dependências de dados.
- Heurística de balanceamento de carga das aplicações entre os recursos disponíveis.
  - Pontuação de desempenho proporcional à taxa de execução histórica de tarefas.
- Tolerância a falhas:
  - redundância,
  - resubmissão de execuções que falharam,
  - re-execução de script sem repetição de computações.

Swift: Ambiente de Execução

- Responsável submissão de execução de aplicações componentes e transferência de arquivos.
- Provedores de dados e de execução.
- Catálogo de aplicações componentes.
- Catálogo de recursos computacionais.

Swift: Ambiente de Execução



Swift: Ambiente de Execução

Configuração do ambiente de execução (`swift.conf`):

```
site.localhostsge {
   execution {
       type: "coaster"
       jobManager: "local:sge"
       URL : "localhost"
       options {
          maxJobs: 16
          nodeGranularity: 1
          maxNodesPerJob: 1
          tasksPerNode: 8
          jobQueue: "linux.q"
          maxJobTime: "24:00:00"
          jobOptions.pe: "mpi8"
       }
   }
   staging: local
   workDirectory: ${env.HOME}"/swiftwork"
   maxParallelTasks    : 128
   initialParallelTasks: 128
   app.ALL { executable: "*" }
}
```

Exemplo de paralelização do BLAST com Swift

- Código disponível em:
  - `https://github.com/lgadelha/swift-blast`
- Requisitos: `fastasplit` e `blastmerge` de D. Mathog (Caltech).

Exemplo de paralelização do BLAST com Swift

```
type fastaseq;
...
type database
{
  headerfile phr;
  indexfile pin;
  seqfile psq;
}
...
string num_partitions=arg("n", "8");
string program_name=arg("p", "blastp");
fastaseq dbin <single_file_mapper;file=arg("d", "database")>;
...
fastaseq partition[] <ext;exec="splitmapper.sh",n=num_partitions>;
```

Exemplo de paralelização do BLAST com Swift

```
app (fastaseq out[]) split_database (fastaseq d, string n)
{
  fastasplitn filename(d) n;
}
app (database out) formatdb (fastaseq i)
{
  formatdb "-i" filename(i);
}
app (output o) blastapp(query i, fastaseq d, string p, string e,
                        string f, database db){
  blastall "-p" p "-i" filename(i) "-d" filename(d) "-o" filename(o)
           "-e" e "-T" "-F" f;
}
app (output o) blastmerge(output o_frags[])
{
  blastmerge filename(o) filenames(o_frags);
}
```

Exemplo de paralelização do BLAST com Swift

```
partition=split_database(dbin, num_partitions);

database formatdbout[] <ext; exec="formatdbmapper.sh",n=num_partitions>
output out[] <ext; exec="outputmapper.sh",n=num_partitions>;

foreach part,i in partition {
  formatdbout[i] = formatdb(part);
  out[i]=blastapp(query_file, part, program_name, expectation_value,
                  filter_query_sequence, formatdbout[i]);
}

blast_output_file=blastmerge(out);
```

Obrigado!

E-mail: lgadelha@lncc.br