DATA FORMAT SUPPORT FOR PARALLEL NUMERICAL INTEGRATION

Wasim El-Hajj and Shujun Li
Department of Computer Science
Western Michigan University
Kalamazoo, MI 49008, U. S. A.
email: {welhajj, sli}@cs.wmich.edu

K. Kaugars and E. de Doncker
Department of Computer Science
Western Michigan University
Kalamazoo, MI 49008, U. S. A.
email: {kkaugars, elise}@cs.wmich.edu

ABSTRACT
This paper presents a data format for the parallel numerical integration package PARINT using XML. As with many other numeric computation programs, PARINT accepts a long list of arguments for describing the user's problem, the algorithm to be used and for specifying parallel run characteristics. Supporting XML input allows platform-independent creation and manipulation of input specifications and simplifies the addition of new integration algorithms. We discuss the purpose of each section in the proposed XML data format, and describe how new sections can be added to the XML data structure in order to support new computing paradigms. We also explain how data are processed efficiently and give some application examples. The format can serve more generally for various software packages.

KEY WORDS
distributed computation, EXtensible Markup Language, parallel integration

1 Introduction

PARINT is a package for parallel multivariate integration. The software package can be used in either a uniprocessing or in a cluster environment over MPI [1], allowing the solution of computationally intensive problems. The package handles multivariate vector integrand functions for integration over hyper-rectangular or simplex regions using a variety of integration algorithms. User-specified parameters control all aspects of operation (e.g. the desired accuracy of the answer and limits on the memory or cpu utilization allowed). Executing PARINT requires the user to supply a long list of command-line parameters which is both time-consuming and error-prone. Ideally, these types of systems should provide a mechanism whereby the user can easily enter parameters, save the entered parameters, include subsets of parameters for different runs, etc... These parameter management capabilities can be readily provided using an underlying platform-independent specification such as XML.

The EXtensible Markup Language (XML) utilizes tags formatted in a manner similar to HTML, but allows the user to define their own tags which more closely reflect the underlying structure of the data. In contrast to a pre-defined set of tags such as those used by HTML, the focus of XML is to describe data rather than present data or define processing rules for the data.

By developing an XML interface to PARINT, we avoid forcing the end user to provide a long list of command line parameters. All parameters are loaded from XML and the user can apply new algorithms or methods by small modifications to the file. The XML files can be manipulated with a variety of external programs, decoupling PARINT binaries from parameter-specification GUIs and allowing the integration of PARINT with emerging standards such as those of Web services [9]. In subsequent sections of this paper we will give a brief overview of PARINT and discuss the use of XML in Sections 2, 3 and 4. Section 5 describes how data are processed, and some application examples are given in the Appendix.

2 PARINT Overview

PARINT [2] is a project that targets the design, analysis and development of a set of coarse grain parallel and distributed algorithms for multivariate numerical integration. The purpose is to compute an approximation Q to the multivariate integral

$$I = \int_D f(\vec{x}) d\vec{x} \tag{1}$$

and an error bound E such that

$$|I - Q| \leq E \leq \max\{\varepsilon_a, \varepsilon_r\,|I|\}, \tag{2}$$

for given absolute and relative error tolerances ϵ_a and ϵ_r, respectively. The integration domain D is a hyper-rectangular or simplex region or a set of such regions; D may also be a general product region, which is handled via integral transformations.

PARINT is layered over MPI. It runs on a single processor or on a user-specified number of processors of its host system. Generally we use one process per processor. The algorithms used for solving the integration problems include:

An adaptive method used for low dimension problems (say, ≤ 10 dimensions). It utilizes load balancing techniques that handle localized integrand difficulties.

A Monte Carlo (MC) algorithm for higher dimensions and possibly erratic integrand functions and domains.

A non-adaptive Quasi-Monte Carlo (QMC) technique used also for higher dimensions but fairly smooth integrands. Load balancing strategies are applied in this method as well.

Specialized applications based on QMC methods for the calculation of multivariate normal and Student-t distribution integrals.

The adaptive method uses p processors one of which functions as a controller and keeps track of the global result (Q) and error estimate (E). The controller may also act as a worker. The algorithm starts by partitioning the given domain among the workers. Each step of the worker process algorithm starts with the selection of the next region to subdivide (based on the local region error estimate). The region selected is subdivided and its subregions evaluated. After a number of subdivisions, an update message is sent to the controller to report changes in the worker's integral and error estimate. The controller periodically sends a message to the worker with the current value of the approximate error tolerance,

$$\varepsilon = \max\{\varepsilon_a, \varepsilon_r\,|Q|\}.$$

When the integration domain D is subdivided initially, some workers may receive a difficult region (with integrand problems such as singularities or peaks) and some workers may receive an easy region. The workers with a hard region need more time to complete their part than those with an easy region, because they have to subdivide the region many times. The workers with an easy region soon become idle, and thus load balancing is applied to maintain efficient use of the processors. Workers report their status (busy / idle) to the controller. When a worker is idle, the controller directs a busy worker to transfer some work to the idle one.

Despite its slow convergence, the Monter Carlo MC technique provides a way to obtain numerical results for ill-behaved integrands of which no a-priori knowledge about their behavior is available. The parallel MC algorithm in PARINT utilizes a specific parameter, *utime*. Every worker is required to work *utime* seconds before reporting its result to the controller. This strategy yields good speedup results, even on heterogeneous systems [4].

Quasi-Monte Carlo (QMC) methods be used in higher dimensions, and generally have better convergence properties than MC under certain conditions on the integrand. The QMC algorithm in PARINT generates a sequence of randomized (Korobov) lattice rule approximations in parallel,

$$K_N(\beta) = \frac{1}{N}\sum_{i=1}^{N} f(\{\frac{i}{N}v + \beta\}), \tag{3}$$

where v is the lattice generator vector, and β is a uniformly distributed random vector. Computing the rule for q random β_j vectors, the integral is approximated by

$$\bar{K}_N = \frac{1}{q}\sum_{j=1}^{q} K_N(\beta_j), \tag{4}$$

which allows for a standard error estimation

$$E_N^2 = \frac{1}{q(q-1)}\sum_{j=1}^{q}(K_N(\beta_j) - \bar{K}_N)^2. \tag{5}$$

The lattice rule sequence uses an increasing number N of integration points to calculate $\bar{K}_N$, until either an answer is found according to the accuracy the user specified or until the function count limit is reached. The parallel methods in PARINT rely on a non-trivial splitting of the work to avoid load imbalances and braking loss resulting from the increasing rule size [7, 3].

PARINT can be executed as a stand alone executable program, with the user's integrand function stored in a library of functions. Command line parameters are used to specify the function to be integrated, along with other parameters to determine the accuracy, etc. The user may also call PARINT functions within other applications. The integrand function in this situation is passed in as a function pointer.

For problems where a large number of integrals need to be computed, a hierarchical process structure was implemented and tested as resported in [5, 6], where a global controller gives out integrals to subgroups of processes.

3 EXtensible Markup Language

XML (EXtensible Markup Language) can be thought of as a generalized form of HTML but without some of the restrictions of SGML. The language allows the tagging of data elements with text strings enclosed in "<" and ">" brackets. In addition, tags may also contain data attributes further describing the data. Tags are organized into a tree structure to allow data grouping. In contrast to HTML, the tags are not pre-defined language elements but instead are application dependent values. XML data is stored in files as text, sacrificing compactness for portability and ease of modification. XML is considered a meta-language, i.e., a language for describing other languages.

An XML file is optionally associated with a Document Type Definition (DTD) or Schema [11] which describes the structure of a particular set of XML tags. The DTD or Schema is used by XML parsers to validate a document for conformance to the particular application's XML specification. Standard formatting for a particular XML document can be provided by style sheets which provide layout and display instructions for tags.

XML is system independent and allows application developers to define a set of tags which are meaningful for the data associated with a particular application. Data is not restricted to English-language strings and tags can be named so that the data files are self-explanatory or comments may be added to the file to provide explanations for tags as is done in the example presented here. Parsing the XML document results in a tree of nested tags along with associated attributes and data, allowing efficient subset processing within program modules. Tags and attributes are easily added to the specifications, and software modules are free to ignore any tags they do not recognize.

4 Integration Parameters

This section discusses aspects of Appendix A, which provides an example of the XML data format for numerical integration programs. Since a main characteristic of XML is its extensibility, developers are free to add extensions or remove redundant elements from the XML description. Except for the element *parint*, all elements are intended to be optional, so that the developers and users will have maximum flexibility. Although the sequence of elements is arbitrary, their relationship must be preserved. For example, function name *f* should be a child of *problem*, which is a child of *parint*.

4.1 Problem Definition Parameters

The entries in the integration problem section provide the data of the problem to be solved and the target accuracy to be achieved. The major elements include the integrand function body, the plug-in function library, the function name, the dimension, the integration region(s), the expected accuracy specified by error tolerances, etc.

There are two main types of region elements, hyper-rectangular and simplex regions, each of which consists of one or more regions. For an n-dimensional integration problem, a hyper-rectangular region can be represented by two points (the upper and lower bounds); a simplex is determined by $n + 1$ points (the simplex vertices). Each point has n coordinates, which are enclosed inside the tag *pt*. For simplicity, individual coordinates are not tagged.

An integrand function may be provided by either specifying a function body in the XML file or by selecting a pre-compiled function from a function library. Either approach is supported by the problem specification through distinct tags, and the application selects which form to use at runtime. Function body specification within the XML file causes a minor difficulty: XML uses the “<” character to initiate a tag and the “&” character to initiate an escape sequence. The writer of the XML document may choose to either escape all occurrences of these special characters or may enclose text within a special tag. The latter approach is illustrated in the sample – the function body is enclosed within a “<![CDATA[” starting element and a “]]>” ending element, indicating to XML parsers that the enclosed text should not be interpreted, allowing a “natural” representation of the C code of the function body.

4.2 Algorithm Parameters

The *algorithm* tag is used as the root of a tree where common algorithmic parameters are direct children and method-specific parameters are enclosed in subtrees named by the algorithm. The direct parameters include the function evaluation limit (*lf*), number of processing units (*np*), a flag for whether the controller also works (*caw*), a flag for load balancing (*lb*), etc.

The *adaptive*, *qmc*, and *mc* tags are all children of *algorithm* and represent parameter sets used for a specific integration method. The *adaptive* parameters are for cubature rules, the *qmc* parameters are used for the quasi-Monte Carlo method and *mc* parameters apply to Monte Carlo integration.

4.3 Adding New Sections

New sections may need to be added to the XML data structure to support new algorithms or new methods of execution. The developers simply define a new set of tags which are processed by the new methods. If the new tags are related to some existing method, they can be added as children of the appropriate tag, or if they represent a distinct method they can be added as children of the root *parint* node.

As an example of these additions, the developers recently implemented an experimental version of PARINT running on multiple clusters worldwide using Web services technology, and collecting runtime data back to one machine for real-time computation visualization. Therefore, an element called *webService* was added as a child of *parint*. The remote clusters were represented by endpoints specified by a Uniform Resource Identifier (URI). The *endpoints* tag was therefore coded as a child of the *webService* element and the parent of multiple individual *endpoint* tags. A *visualization* tag, whose parent is *webService*, had visualization specific children, such as the number of grid cells in each direction and the frequency of grid updates.

5 Data Processing

The input file is initially passed to an XML parser which loads and splits tags to build an XML document object. The programming interface to the document object is called the XML Document Object Model (DOM) [10] and provides a tree view of the XML document. A document object can be modified, built without using an input file, passed from a process to another process, or saved to a disk file. The specific programmatic interfaces for manipulating a tree depend on the language or related package used. In order to simplify programming and provide a package-independent interface for PARINT development, we provide several C/C++ functions for parameter retrieval.

The function *parseParameters* accepts the XML file name as its input parameter and builds the document tree. To retrieve a parameter, say *dimension*, the caller can just pass the tag name, *"dimension"*, to a function called *getParameter*, which will return the text content of the problem dimension. The full path, *"parint/problem/dimension"* or any trailing part of the path are also valid. This is useful when multiple tags have the same name. For example, a hyper-rectangular region is defined by its lower and upper bounds (in all coordinate directions), while a simplex region is defined by $n + 1$ vertices, where n is the dimension. They share the tag name *pt*. The calls *getParameter("rect/rgn/pt[1]")* and *getParameter("rect/rgn/pt[2]")* return the lower bounds and the upper bounds, respectively; *getParameter("splx/rgn/pt[1]")* gives the first vertex of the simplex region. These functions are supported by XPath [12], which allows searching through a document for nodes that match specified criteria.

The three functions *goTo*, *hasNext*, and *next* are used together to retrieve the region data efficiently, especially when the number of regions is large. *goTo* accepts a path to an internal node as its input parameter; *hasNext* checks whether there are still regions for processing; *next* returns the region data.

6 Summary and Future Work

We described a method of organizing numerical integration data using the facilities of XML. XML is a good choice because of its extensibility, ease of use and ability to organize input data. We described the XML tag structure used by PARINT and illustrated how new sections can be added to the XML tag structure in order to support new computational models and algorithms. The XML file is read as an input file and then parsed to produce a tree-like structure for the XML document. Functions such as *goTo*, *hasNext* and *next* are used together to retrieve the region data efficiently, particularly when the number of regions is large.

The input data can be generated manually with a text editor. In a future project we plan to configure Pion [8], which provides a Web-based PARINT execution environment, to generate the input files.

The input data can either be returned to the user for further processing, or sent (by a remote function call) to an integration service provider [9], which performs the integration and sends the results back. We are currently experimenting with distributed computation visualization, for which new elements are added to the input. The new PARINT release will be the first that supports XML. The XML format will become a standard method of data exchange in our work on distributed numerical integration. It can also serve more generally for other software packages.

Acknowledgments

The PARINT project has been supported in part under National Science Foundation grants ACR-0000442, EIA-0130857 and ACI-0203776.

References

[1] http://www-unix.mcs.anl.gov/mpi/index.html, MPI web site.

[2] http://www.cs.wmich.edu/parint, PARINT web site.

[3] L. CUCOS AND E. DE DONCKER, *Distributed QMC algorithms: New strategies and performance evaluation*, in Proceedings of the High Performance Computing Symposium (HPC'02), 2002, pp. 155–159.

[4] E. DE DONCKER, L. CUCOS, R. ZANNY, AND K. KAUGARS, *Parallel multivariate integration: Paradigms and applications*, in Joint Statistical Conferences (JSM 2002) CD-ROM Proceedings, 2002.

[5] E. DE DONCKER, A. GUPTA, AND R. ZANNY, *Large scale parallel numerical integration*, Journal of Computational and Applied Mathematics, 112 (1999), pp. 29–44.

[6] E. DE DONCKER, A. K. GUPTA, AND L. CUCOS, *On the scalable computation of large sets of integrals*, in ISCA 16th International Conference on Parallel and Distributed Systems (PDCS), 2003, pp. 144–150.

[7] E. DE DONCKER, R. ZANNY, M. CIOBANU, AND Y. GUAN, *Asynchronous Quasi Monte-Carlo methods*, in Proceedings of the High Performance Computing Symposium (HPC'00), 2000, pp. 130–135.

[8] S. LI, E. DE DONCKER, AND K. KAUGARS, *Pion: A problem solving environment for parallel multivariate integration*, 2003. Submitted, draft at http://www.cs.wmich.edu/∼elise (Publications).

[9] S. LI, K. KAUGARS, AND E. DE DONCKER, *Massive scale distributed integration using web service*, in The Hawaii International Conference on Computer Sciences, 2003. CDROM Proceedings.

[10] W3C, *Document Object Model*. `http://www.w3.org/DOM/`.

[11] ——, *Extensible Markup Language*. `http://www.w3.org/XML/`.

[12] ——, *XPath*. `http://www.w3.org/TR/xpath`.

A Sample XML File

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!-- parameter file for integration programs -->
<parint>

<!-- parameters that describe the problem -->
<problem>
  <!-- plugin name, the binary file that contains
the function -->
    <L>stdfuncs.ppl</L>
  <!-- The function name, which must be the same
as the function name in body section
        below, if present -->
    <f>fcn7</f>
  <!--A description of the integrand function,
        which is usually optional -->
    <description>
      The function is f(x) = 1 / (x0 + x1 + x2)ˆ2
    </description>
  <!-- Dimension -->
    <dimension>3</dimension>
  <!-- Absolute error tolerance-->
    <ea>1E-6</ea>
  <!-- Relative error tolerance-->
    <er>1E-6</er>
  <!-- Relative error tolerance as # of digits
   -->
    <ed></ed>
  <!-- Number of integrands, usually 1 -->
    <nfcns>1</nfcns>
  <!-- The region(s) used may also be a default,
         or determined by the rule number below
   -->
    <rgns>
      <!-- Define a multiple region set -->
      <rect>
        <!-- First region -->
        <rgn>
          <pt> 0.0 0.0 0.0 </pt>
          <pt> 1.0 1.0 1.0 </pt>
        </rgn>
        <!-- Second region -->
        <rgn>
          <pt> 1.0 1.0 1.0 </pt>
          <pt> 2.0 2.0 2.0 </pt>
        </rgn>
      </rect>
        <!-- Third region -->
      <splx>
        <rgn>
          <pt> 0.0 0.0 0.0 </pt>
          <pt> 1.0 0.0 0.0 </pt>
          <pt> 0.0 1.0 0.0 </pt>
          <pt> 0.0 0.0 -1.0 </pt>
        </rgn>
        <!-- Fourth region -->
        <rgn>
          <pt> 0.0 0.0 0.0 </pt>
          <pt> -1.0 0.0 0.0 </pt>
          <pt> 0.0 -1.0 0.0 </pt>
          <pt> 0.0 0.0 -1.0 </pt>
        </rgn>
      </splx>
    </rgns>

  <!-- Function body, contains an integrand
        function (see also the ParInt manual)
   -->
    <body>
    <![CDATA[
      int fcn7(int *ndims, pi_base_t *x,
       int *nfcns, pi_base_t *funvls){
        pi_base_t z = x[0] + x[1] + x[2];
        z *= z;
        funvls[0] = (z != 0.0
              ? 1.0 / z
              : 0.0);
        return 0;
      }
    ]]>
    </body>
</problem>

<algorithm>
  <!-- Function evaluation limit,
can be very large -->
    <lf>400000</lf>
  <!-- Number of runs, usually 1 -->
```

```
    <onr>1</onr>
  <!-- Load balancing? 1, yes; 0, no -->
    <lb>1</lb>
  <!-- Controller also works? 1, yes; 0, no -->
    <caw>1</caw>
  <!-- Number of processors -->
    <np>1</np>

  <!-- Adaptive algorithm parameters -->
  <adaptive>
    <!-- Integration rules
    PI_IRULE_DIM2_DEG13 = 1;
    PI_IRULE_DIM3_DEG11 = 2;
    PI_IRULE_DEG9_OSCIL = 3;
    PI_IRULE_DEG7 = 4;
    PI_IRULE_DQK15 = 5;
    PI_IRULE_DQK21 = 6;
    PI_IRULE_DQK31 = 7;
    PI_IRULE_DQK41 = 8;
    PI_IRULE_DQK51 = 9;
    PI_IRULE_DQK61 = 10;
    PI_IRULE_QMC = 11;
    PI_IRULE_SIMPLEX_DEG3 = 12;
    PI_IRULE_SIMPLEX_DEG5 = 13;
    PI_IRULE_SIMPLEX_DEG7 = 14;
    PI_IRULE_SIMPLEX_DEG9 = 15;
    -->
    <!-- Integration rule, must be a number
     -->
      <r>2</r>
    <!-- Error ratio threshold for load
            balancing, >= 1.0 -->
      <ohr>1.2</ohr>
    <!-- Worker granularity -->
      <ons>5</ons>
    <!-- Maximum heap size. 0, no limit -->
      <ohs>0</ohs>
    <!-- Max # of region evaluations -->
      <lr> </lr>
    <!-- Percentage threshold controlling
   eps messages -->
      <oet>0.01</oet>
    <!-- Percentage threshold controlling
   update message -->
      <out>0.1</out>
  </adaptive>

  <!--- QMC algorithm parameters -->
  <qmc>
    <!-- qmc_table_length -->
      <oqcl>10</oqcl>
    <!-- qmc_table_width -->
      <oqcm>6</oqcm>
    <!-- Start row -->
      <b></b>
    <!-- Number of columns -->
      <c></c>
    <!-- split each cell into <count>
           pieces -->
      <k></k>
  </qmc>

  <!-- MC algorithm parameters -->
  <mc>
    <!-- Work unit time (in seconds) -->
      <t>0.1</t>
  </mc>
</algorithm>

<hierarchical>
  <!-- Turn on hierarchical method and set
the number of groups -->
```

```
    <hiergroup></hiergroup>
  <!-- Set function parameter file for
hierarchical run -->
    <hierfile></hierfile>
</hierarchical>

<!-- IO settings -->
<io>
  <!-- Less verbose? -->
    <lessverb>0</lessverb>
  <!-- Display temporary results every n evals
   -->
    <tr></tr>
  <!-- Output file name (records tagged values)
   -->
    <o>result.txt</o>
  <!-- Output file for logging -->
    <plfile>temp.txt</plfile>
  <!-- Read the regions set from this file -->
    <InRgnfile>file.in</InRgnfile>
  <!-- Write regions to this file -->
    <OutRgnfile>file.out</OutRgnfile>
  <!-- Input file system type used for inRgnfile,
     where:
      0 - read files from NFS mounted partition;
      1 - read files from a partition local to
    each worker;
      2 - the files are read only by the
    controller -->
    <InFS></InFS>
  <!-- Output file system type used for
      outRgnfile, where:
    1 - each worker writes a separate file;
    2 - the controller (only) writes the files.
   -->
    <OutFS>1</OutFS>
  <!-- Output file type: b - binary, t - text -->
    <OutRgnFT>b</OutRgnFT>
</io>

<webService>
  <!-- Service to be requested:
       0 -  computation;
       1 - max points;
       2 - result and error -->
    <operationType>2</operationType>
  <!-- Endpoints for Web service -->
    <endPoints>
      <endPoint>http://aegis.cs.wmich.edu:8080/
        axis/services/ParIntService
      </endPoint>
      <endPoint>http://lupata.cs.wmich.edu:8080/
        axis/services/ParIntService
      </endPoint>
      <endPoint>http://localhost:8080/
        axis/services/ParIntService</endPoint>
      <endPoint>http://demeter.cs.wmich.edu:8080/
         axis/services/ParIntService
      </endPoint>
    </endPoints>

  <!-- Remote computation visualization -->
  <visualization>
    <speed>100</speed>
    <grid>
      <cells>128</cells>
      <cells>128</cells>
      <cells>10</cells>
    </grid>
  </visualization>

  <!--number of partitions, for pre-splitting-->
    <numPartitions>8</numPartitions>
  </webService>
  </parint>
```