Universidade de São Paulo

Biblioteca Digital da Produção Intelectual - BDPI

Departamento de Física e Ciência Interdisciplinar - IFSC/FCI Artigos e Materiais de Revistas Científicas - IFSC/FCI

2009

HieraAnalyses: a tool for hierarchical analysis of parallel programs

International Journal of High Performance Systems Architecture,Olney : Indersciences Publishers,v. 2, n. 1, p. 58-67, 2009

http://www.producao.usp.br/handle/BDPI/49324

Downloaded from: Biblioteca Digital da Produção Intelectual - BDPI, Universidade de São Paulo

HieraAnalyses – a tool for hierarchical analysis of parallel programs

Thatyana de Faria Piola Seraphim* and Enzo Seraphim

Engineering of Systems and Technologies of the Information Institute,
Federal University of Itajubá,
BPS Av., 1303, Itajubá–MG, Brazil
Fax: +55-35-36291187
E-mail: thatyana@unifei.edu.br
E-mail: seraphim@unifei.edu.br
*Corresponding author

Gonzalo Travieso

Institute of Physics of São Carlos,
University of São Paulo,
Trabalhador São-carlense Av.,
400, São Carlos–SP, Brazil
Fax: +55-16-33722218
E-mail: gonzalo@ifsc.usp.br

Abstract: Detailed information for performance analysis of parallel programs can be collected through trace files. Generally, trace files contain a register of individual events that occurred during program execution. Considering that the events traced are commonly of low level, like communication operations in a parallel system, and that it is increasingly common for the application programmer to use higher level abstractions (e.g., a parallel eigenvalues routine), a semantic gap exists between the collected information and the concepts used for the development of the application, hindering an effective use of that information. In this work, a new approach to trace files is proposed, where the files retain information about the different hierarchical levels in the application. The files follow an XML format, where routines are XML tags, with auxiliary routines called during its execution as child tags. The approach is demonstrated by its implementation for the MPI library level and the OOPS level, this last one being an object-oriented framework with higher level abstractions for the development of parallel programs that uses MPI for its implementation. To complement the work, some analysis tools using the file format are presented.

Keywords: trace; performance analysis; parallel programming.

Reference to this paper should be made as follows: Seraphim, T.F.P., Seraphim, E. and Travieso, G. (2009) 'HieraAnalyses – a tool for hierarchical analysis of parallel programs', *Int. J. High Performance Systems Architecture*, Vol. 2, No. 1, pp.58–67.

Biographical notes: Thatyana de Faria Piola Seraphim is an Associate Professor at the Engineering of Systems and Technologies of the Information Institute at the Federal University of Itajubá, Itajubá–MG, Brazil. She received her MSc (Applied Physics, 2003) and PhD (Applied Physics, 2007) from the Institute of Physics of São Carlos (IFSC/USP), University of São Paulo, São Carlos–SP, Brazil. Her research interests include high performance computing and computer architecture.

Enzo Seraphim is an Associate Professor at the Engineering of Systems and Technologies of the Information Institute at the Federal University of Itajubá, Itajubá–MG, Brazil. He received his MSc (Computer Science, 2000) and PhD (Computer Science, 2006) from the Institute of Mathematics and Computing (ICMC/USP), University of São Paulo, São Carlos–SP, Brazil. His research interests include database management systems (DBMS), data structure and high performance computing.

Gonzalo Travieso is an Associate Professor at the Institute of Physics of São Carlos, at the University of São Paulo, São Carlos–SP, Brazil. He is an Electronic Engineer from Escola de Engenharia de São Carlos, University of São Paulo and received his MSc and PhD on Applied Physics from the Institute of Physics of São Carlos, University of São Paulo, São Carlos–SP, Brazil. His main research interests are on distributed systems and parallel programming.

Copyright © 2009 Inderscience Enterprises Ltd.

1 Introduction

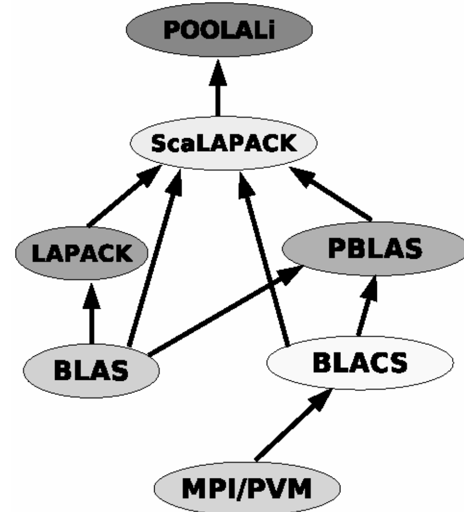
Many applications require performances that cannot be achieved by a single processor, making parallel processing indispensable. Parallel machines ranging from multi-core CPUs to grid computing, from clusters of off-the-shelf computers to dedicated parallel systems are available to supply the needs of these applications. Nevertheless, due to the complexity of parallel software development, the use of parallel systems is mostly restricted to applications that can be easily decomposed in independent tasks or applications where their importance justifies the higher investment in resources needed.

One of the reasons for this situation is the importance of performance for parallel programs, as performance is commonly the factor that justifies switching to a parallel implementation of the program, despite the resulting additional code complexity and hardware cost. Given a parallel application, it is therefore important to be able to analyse the factors that determine its performance. Many tools have been proposed to help this evaluation, like Automated Instrumentation and Monitoring System (AIMS) (Fineman et al., 1997), Pablo (Reed et al., 1993) and Vampir (Moore et al., 2001), among other (see also Section 2). The technique of *trace files* is used to register events that occur during program execution. Through the analysis of these events and their times, conclusions can be drawn about performance bottlenecks or sections of the code where optimisations might be useful.

Parallel programs can be developed using the communication libraries, like Message Passing Interface (MPI) (MPI Forum, 1994) or PVM (Geist et al., 1994), but higher level solutions like ScaLAPACK (Blackford et al., 1997) are being used because they provide abstractions that are closer to the application domain and can be optimised by experts to achieve high performance for a wide range of platforms. The use of higher level abstractions creates a semantic gap problem when working with trace files because these are generally based on lower level events, like communication operations. For concreteness, consider the example of the user of the POOLALi library (Rodrigues, 2004), an object-oriented wrapper to the ScaLAPACK eigenvalues/eigenvectors routines. POOLALi is based on ScaLAPACK, that is based on PBLAS; the last is based on BLACS, that is implemented using MPI (or PVM) (see Figure 1). For the user of POOLALi, trace events related with MPI communication operations are useless. It is important that events at the level of POOLALi method calls be registered. But in some cases, further analyses require access to events of a lower level abstraction. A full trace file-based approach to performance analysis should therefore include information on events over all abstraction levels.

This article presents the *Hierarchical Analyses* tool, which enables performance evaluation at different abstraction levels. The reminder of this articles is organised as follows: Section 2 discusses some related performance analysis tools; Section 3 presents the tool proposed in this work; Section 4 shows the results of some experiments with the proposed tool, using a molecular dynamics application implemented in MPI and in the Object-Oriented Parallel System (OOPS) framework (Sonoda and Travieso, 2006); and the conclusions are presented in Section 5.

Figure 1 Different abstractions levels in the POOLALi parallel library



2 State of the art

Performance evaluation aims at identifying performance bottlenecks. Tools are used to help understand the behaviour of parallel programs, load balancing, amount of communications and other issues closely related with the performance of the application. Without trying to be comprehensive, some performance-related tools are presented below.

The traditional tool `gprof` helps identify procedures or lines of code where the program spends most of its time (Graham et al., 1982), collecting information about the time taken in each routine and the number of calls. This information is useful for identifying optimisation or parallelisation candidates. There is no explicit support for parallelism in `gprof`.

Multiprocessing Environment (MPE) is related with the MPICH implementation of MPI, but can be used in other implementations. It supports facilities including profiling and visualisation tools. The profiling library works with the profiling interface of MPI (Moore et al., 2001).

Pablo (Browne et al., 1998; Reed et al., 1993), Paraver (Labarta et al., 2001) and Vampir (Moore et al., 2001; Browne et al., 1998) are environments for collection, analysis and visualisation of performance data of parallel programs. Events registered correspond to communication and I/O operations of MPI. Paraver works also with OpenMP and Java. Vampir has a mechanism limit the quantity of recorded events, by choosing the most appropriate events to the desired analysis.

Paradyn (Miller et al., 1995) and *AIMS* (Yan, 1994) enable real-time monitoring of parallel programs. In Paradyn, instrumentation is dynamically adjusted during

program execution. The user specifies the performance data to collect (like CPU time, communication or synchronisation operations) and the parts of the program to instrument. There is no need to recompile the program to change the instrumentation behaviour. In AIMS, the program behaviour can be visualised through animations.

In IPS (Miller et al., 1990; Hollingsworth et al., 1991), instrumentation code is automatically inserted during compilation, with collection of events like procedure call and return, synchronisation operations, I/O, process creation, among other.

SCALEA (Truoung and Fahringer, 2003) is a performance instrumentation, measurement, analysis and visualisation tool for parallel programs that supports post-mortem performance analysis. It supports profiling and tracing for parallel and distributed programs and sensor managers for capturing and managing performance data of individual computing nodes of parallel and distributed machines. The SCALEA profiling and tracing library collects timing, event and counter information, as well as hardware parameters [determined through an interface with a PAPI library (Browne et al., 2000)]. The *Scalea Instrumentation System* (SIS) provides the user with three alternatives to control instrumentation, which includes command-line options, SIS directives and a high level instrumentation library combined with an OpenMP, MPI, HPF front-end and unparser. All of these alternatives support the specification of performance metrics and code regions of interest for which SCALEA automatically generates instrumentation code and determines the desired performance values during or after program execution.

The *UAH Logging, Trace Recording and Analysis* (ULTRA) instrumentation system (Cohen et al., 2007) provides an accurate and low cost mean of collecting traces of MPI program execution. These traces preserve the original parallel program's data-dependencies by recording each MPI operation performed, the message source, destination and size, and the number of application instructions preceding the operation. The instrumentation introduces a small amount of overhead when an MPI communication library function is called, allowing data to be collected on large production runs of parallel programs. The instrumentation uses wrappers inserted between the application code and the functions that implement the MPI operations.

3 Hierarchical analyses tool

From the above presented performance evaluation tools, none is structured to take into account the various abstraction levels used in the development of the application. The following sections describe the *HieraAnalyses* tool, developed to demonstrate the feasibility of the approach proposed in this work. For the tool development was used software instrumentation in library routines of static form, by the facility of instrumentation and does not need a dedicated hardware. The tool is composed of two modules: a collector module, described in Section 3.1, and a transformation module, described in Section 3.2.

3.1 Data collection

The data to be used for performance analysis is collected and stored by the hieraCollector module. An *eXtensible Markup Language* (XML) (W3C, 2009) format is used which reflects the logical organisation of procedure calls in a tree structure, with a procedure call being child of the procedure call that resulted in its execution.

Each library routine that should have its execution monitored must be adapted by inclusion of instrumentation code. This is done at present manually by the library developer or someone else with access to the source code. Collection operations where developed for the MPI library, using its profiling interface and for the *OOPS* framework (Sonoda and Travieso, 2006), a class library with high level abstractions for the development of parallel applications. As OOPS uses MPI for its implementation, it is possible, through the hierarchical collection system to analyse the performance at the level of OOPS method calls or MPI communication operations.

The grammar of the generated XML file is defined by a *Document Type Definition* (DTD) file. The DTD used for MPI and OOPS is presented below.

```
1   <!ELEMENT processor(hieraMPI | hieraOOPS)*>
2   <!ATTLIST processor
3       rank        ID          #REQUIRED
4       init        CDATA       #REQUIRED
5       finalize    CDATA       #REQUIRED>
6   <!ELEMENT hieraMPI EMPTY>
7   <!ATTLIST hieraMPI
8       operation (address|allgather|allgatherv|allreduce|
9           alltoall|alltoallv|barrier|broadcast|bsend|
10          bsend_init|buffer_attach|buffer_detach|cancel|
11          comm_create|comm_dup|comm_split|gatherv|
12          gather|get_count|get_elements|ibsend|
13          intercomm_create|intercomm_merge|iprobe|
14          irecv|irsend|isend|issend|pack|pack_size|probe|
            receive|
15          recv_init|reduce|reduce_scatter|request_free|rsend|
16          rsend_init|scan|scatter|scatterv|send|send_init|
17          sendrecv|ssend|sendrecv_replace|ssend_init|start|
18          startall|test|testall|testany|test_cancelled|testsome|
19          type_commit|type_contiguous|type_extent|
20          type_free|type_hindexed|type_hvector|
21          type_indexed|type_lb|type_size|type_struct|
22          type_ub|type_vector|unpack|wait|waitall|waitany|
23          waitsome) #REQUIRED
24      file        CDATA       #REQUIRED
25      line        CDATA       #REQUIRED
```

```
26      start_time    CDATA     #REQUIRED
27      finish_time   CDATA     #REQUIRED
28      count         CDATA     #IMPLIED
29      type          CDATA     #IMPLIED
30      dest          CDATA     #IMPLIED
31      tag           CDATA     #IMPLIED
32      com           CDATA     #IMPLIED
33      …
34  >
35  <!ELEMENT hieraOOPS (hieraOOPS|hieraMPI|
36  EMPTY)*>
37  <!ATTLIST hieraOOPS
38      class (unknown|distributionBlocked|matrix
39          distributionCyclic|distributionNone|vector|
40          vectorRepl|vectorSequ|group|partner|topology|
41          topologyGrid|topologyLinear|topologyPlain|
42          topologyPipe|topologyTorus|workgroup)
43          #REQUIRED
44      method (allreduce|barrier|bcast|col|
45          distributionBlocked|distributionCyclic|
46          distributionNone|fromEast|fromNext|fromNorth|
47          fromSouth|fromWest|fromNE|fromNW|fromSE|
48          fromSW|gather|gatherv|globalToLocal|isInGroup|
49          localSize|localToGlobal|matrix|max|min|partner|
50          prod|recv|reduce|row|scatter|scatterv|send|split|
51          store|subGroup|sum|syncGhostsCart|topologyGrid|
52          topologyPipe|toEast|toNorth|toPrevious|toSouth|
53          toWest|toNE|toNW|toSE|toSW|vector|vectorRepl|
54          vectorSequ) #REQUIRED
55      file          CDATA     #REQUIRED
56      line          CDATA     #REQUIRED
57      start_time    CDATA     #REQUIRED
58      finish_time   CDATA     #REQUIRED
59      …
60  >
```

The root element `processor` holds information of process ID (like MPI rank) and times of start and finish of the execution. All operations executed, of types `hieraMPI` or `hieraOOPS`, are a child of this element. These operations may be point-to-point or collective operations in MPI or method calls in OOPS, with the information carried by each element dependent on the element; common information are operation name, file name and line of the call, start and finish time of the operation. For OOPS elements, class and method names are registered.

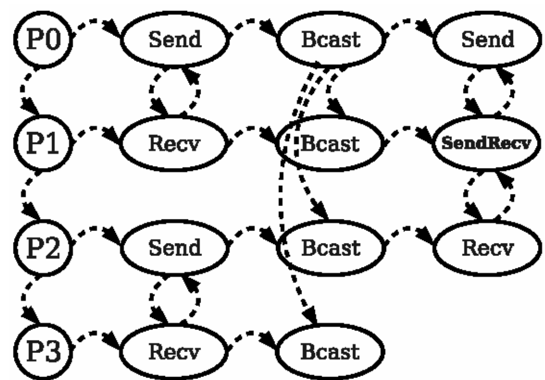
During execution of the instrumented code, two types of files are generated: a configuration file and one trace file for each process. The configuration file holds information about all trace files.

3.2 Analysis

The *hieraTransform* module reads the collected data and builds a memory representation from which measurements can be computed for the performance analysis of the program execution. It can thus be understood as operating in two phases: transformation and measurements.

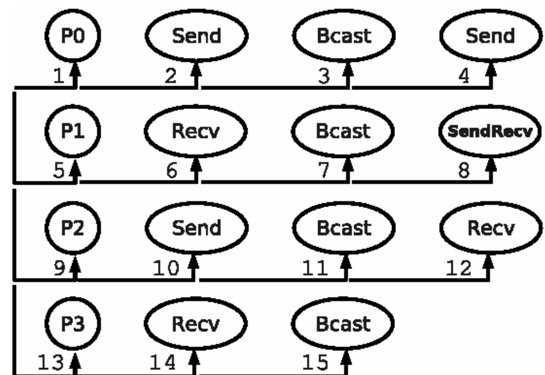
The representation phase reads the XML files generated by *hieraCollector* and build a graph whose vertexes represent the operations (the elements in the XML representation) and whose edges represent relations between them. Figure 2 shows an example with four processors (P0, … , P3) and where, for example, P0 executed the operations `send`, `bcast` and `send`. Related communication operations have their respective vertexes linked by edges.

Figure 2 Example of the graph generated by *hieraTransform*



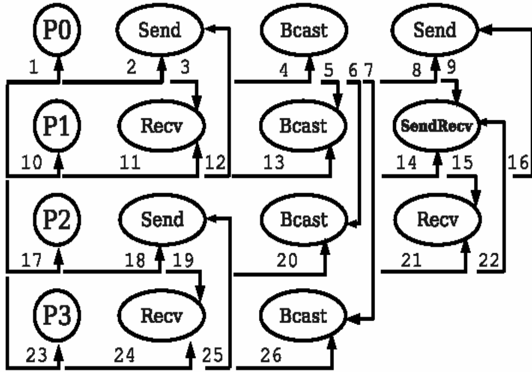
The edges of the graph enable various navigation procedures to be deployed for performance evaluation. One possibility as shown in Figure 3 is to traverse the graph one operation at a time, going through each operation just once and each process one after the other. The operations in the figure are therefore traversed in the order 1, 2, 3, 4, … , 15.

Figure 3 Graph traversal based on the operations



Another possibility is to traverse the partner operations (like the corresponding receive to a send) before going on to the next operation of the same process, as shown in Figure 4. In this case, each operation may be visited many times. For the example in the figure, the traversal is 1, 2, 6, 3, 7, 11, 15, 4, 8, 5, 6, 2, 7, 8, 4, 12, 9, 10, 14, 11, 12, 8, 13, 14, 10, 15.

Figure 4 Traversal based on partner operations



In the measurement phase, the first task is to choose what to measure. Due to the large amount of data collected during the execution of parallel programs, the measurements are generally of the statistical nature, like number of operations, averages, standard deviations or histograms. The measurements are evaluated by traversing the graph in an appropriate way, e.g., using the traversal by operations of Figure 3 to count the number of each operation type executed.

4 Experiments

As already said, *hieraCollector* was implemented for MPI and OOPS. *MPI* was chosen due to its widespread use by the parallel programming community and its availability for a wide range of machines, enabling code portability while retaining execution efficiency.

The *OOPS* framework is a class library aiming to support the development of regular scientific applications with extensive use of distributed matrices and vectors. It supports higher level abstractions for the development of the parallel code, without completely hidden the parallelism. Its implementation is based on MPI. For this reason, it is well-suited as a testbed for the tool proposed in this work, as the user of OOPS will develop the code based on OOPS abstractions, instead of the underlying MPI abstractions.

To test the tool in a real application scenario, a program that computes molecular dynamics of Lennard-Jones particles using the force decomposition algorithm of Plimpton (1995) was implemented and evaluated. A given number of particles are distributed in a tri-dimensional box subject to periodic boundary conditions and initial position and velocities for the particles are specified. Afterwards, the particles evolve according to the Lennard-Jones interaction among them. The computation of the interaction forces between each pair of particles is decomposed among the available processors, with the particles distributed in blocks to the processors, the processors arranged in a two-dimensional processor grid and each processor being responsible for the interaction of particle in the same row with particles in the same column. See Plimpton (1995) for a complete description of the algorithm. The algorithm was implemented in an MPI version and an OOPS version.

Execution times reported below refer to the execution on an eight node cluster of Pentium 4, 3.0 GHz machines running GNU/Linux.

4.1 HieraCollector for MPI and OOPS

The MPI and OOPS versions of the program were executed with four processes. The configuration file generated is similar for the two versions and shown in the frame below. The root is a `hieraCollector` element with the application name and number of processes used for the execution. The children are `collect_file` elements with the information about the files that have the collected data from each process. For instance, line 4 says that the data collected from the process with rank 0 is stored in the file named `trace0.xml`.

```
1  <?xml version="1.0"?>
2  <!DOCTYPE hieraCollector SYSTEM
   "hieraCollector.dtd">
3  <hieraCollector application="dinamica"
   count_proc="4">
4     <collect_file rank="P0">trace0.xml</collect_file>
5     <collect_file rank="P1">trace1.xml</collect_file>
6     <collect_file rank="P2">trace2.xml</collect_file>
7     <collect_file rank="P3">trace3.xml</collect_file>
8  </hieraCollector>
```

Part of the contents of file `trace0.xml` for the MPI program version is shown in the box below. It shows the root element `processor` with process identification `rank=P0`, start and finish times. Children of `processor` are the various MPI operations executed, all of type `hieraMPI` and corresponding operation fields (`broadcast`, `receive`, `send`, etc.) and fields for information about the operation, like file and line number, start and finish time, etc.

```
1  <?xml version="1.0" encoding="ISO-8859-1"?>
2  <!DOCTYPE processor SYSTEM "trace.dtd">
3  <?xml-stylesheet type="text/xsl"
4     href="visual.xsl"?>
5  <processor rank="P0" init="0.01229"
6     finalize="1.96906">
7  <hieraMPI operation="broadcast" file="md.c"
8     line="37"start_time="0.01992"
9     finish_time="0.02001"count="1"
10    type="MPI_INT" root="0"
11    comm="MPI_COMM_WORLD"/>
12 <hieraMPI operation="receive" file="auxfmd.c"
13    line="51"start_time="0.02540"
14    finish_time="0.02542"count="1"
15    type="MPI_INT" rem="1" tag="0"
16    comm="MPI_COMM_WORLD"/>
```

```
17  <hieraMPI operation=”send” file=”auxfmd.c”
18      line=”58”start_time=”0.02645”
19      finish_time=”0.02649” count=”1536”
20      type=”MPI_FLOAT” dest=”1” tag=”1”
21      comm=”MPI_COMM_WORLD”/>
22  <hieraMPI operation=”type_contiguous”
23      file=”md.c” line=”92” start_time=”0.04312”
24      finish_time=”0.04313”count=”3”
25      oldtype=”MPI_FLOAT” newtype=”P6_dtype”/>
26  <hieraMPI operation=”type_commit” file=”md.c”
27      line=”93”start_time=”0.04316”
28      finish_time=”0.04316”type=”P6_dtype”/>
29  <hieraMPI operation=”scan” file=”md.c” line=”95”
30      start_time=”0.04319” finish_time=”0.04321”
31      count=”1”type=”MPI_INT” op=”MPI_SUM”
32      comm.=”MPI_COMM_WORLD”/>
33  <hieraMPI operation=”allgatherv” file=”md.c”
34      line=”137”start_time=”0.05551”
35      finish_time=”0.05568”scount=”512”
36      stype=”MPI_INT” rcount=”512”
37      rtype=”MPI_INT” comm=”P5_comm”/>
38  <hieraMPI operation=”reduce” file=”md.c”
39      line=”171”start_time=”0.07606”
40      finish_time=”0.07609”count=”1536”
41      type=”MPI_FLOAT” op=”MPI_SUM”
42      root=”1” comm=”P5_comm”/>
43  …
44  </processor>
```

The following box shows part of the `trace1.xml` file generated by the execution of process rank 1 of the OOPS version of the molecular dynamics code. The root element is again `processor`, with rank information `rank="P1"` and initial and final time for the process. The children are `hieraOOPS` elements for the methods called during the execution, with information about class and method. For instance, line 6 shows the call for the constructor of class `TopologyPipe`, implemented in line 17 of file `TopologyPipe.cc`, called with arguments `next` and `previous` as specified.

```
1   <?xml version=”1.0” encoding=”ISO-8859-1”?>
2   <!DOCTYPE processor SYSTEM ”trace.dtd”>
3   <?xml-stylesheet type=”text/xsl” href=”visual.xsl”?>
4   <processor rank=”P1” init=”0.000064”
5       finalize=”134.766716”>
6   <hieraOOPS file=”TopologyPipe.cc” line=”17”
7       class=”OOPS::TopologyPipe”
8       method=”TopologyPipe”previous=”0” next=”2”
9       start_time=”0.014255”finish_time=”0.014377”/>
10  <hieraOOPS file=”TopologyGrid.cc” line=”4273”
11      class=”OOPS::TopologyGrid” method=”split”
12      color=”0” key=”1” start time=”0.014401”
13      finish_time=”0.023591”>
14      <hieraOOPS file=”Topology.cc” line=”2221”
15          class=”OOPS::Group” method=”split” color=”0”
16          key=”1”start_time=”0.014440”
17          finish_time=”0.023584”/>
18  </hieraOOPS>
19  <hieraOOPS file=”TopologyGrid.cc” line=”4274”
20      class=”OOPS::TopologyGrid” method=”split”
21      color=”1”key=”1” start_time=”0.023638”
22      finish_time=”0.031642”>
23      <hieraOOPS file=”Topology.cc” line=”2221”
24          class=”OOPS::Group” method=”split” color=”1”
25          key=”1”start_time=”0.023672”
26          finish_time=”0.031634”/>
27  </hieraOOPS>
28  <hieraOOPS file=”molecularDynamicsOOPS.cc”
29      line=”0”class=”OOPS::TopologyGrid”
30      method=”TopologyGrid”gsize=”4” cols=”2”
31      start_time=”0.014385”finish_time=”0.031741”/>
32  <hieraOOPS file=”molecularDynamicsOOPS.cc”
33      line=”45” class=”OOPS::TopologyGrid”
34      method=”bcast” root=”0” start_time=”0.031749”
35      finish_time=”0.031878”>
36      <hieraOOPS file=”Topology.cc” line=”2335”
37          class=”OOPS::Group” method=”bcast”
38          root=”0” start_time=”0.031788”
39          finish_time=”0.031872”>
40          <hieraMPI operation=”broadcast”
            file=”Topology.cc”
41              line=”2335” count=”1” type=”MPI_INT”
42              root=”0” comm=”OOPS_Comm1”
43              start_time=”0.031788”
                finish_time=”0.031866”/>
44      </hieraOOPS>
45  </hieraOOPS>
46  <hieraOOPS file=”molecularDynamicsOOPS.cc”
47      line=”46” class=”OOPS::TopologyGrid”
48      start_time=”0.031885” finish_time=”0.032002”>
49      <hieraOOPS file=”Topology.cc” line=”2335”
50          class=”OOPS::Group” method=”bcast” root=”0”
51          start_time=”0.031918” finish_time=”0.031995”>
52          <hieraMPI operation=”broadcast”
53              file=”Topology.cc” line=”2335” count=”1”
54              type=”MPI_INT” root=”0”
                comm=”OOPS_Comm1”
55              start_time=”0.031918”
                finish_time=”0.031989”/>
56      </hieraOOPS>
```

```
57  </hieraOOPS>
58  <hieraOOPS file="molecularDynamicsOOPS.cc"
59      line="47" class="OOPS::TopologyGrid"
60      method="bcast" root="0" start_time="0.032008"
61      finish_time="0.032123">
62      <hieraOOPS file="Topology.cc"line="2335
63          class="OOPS::Group" method="bcast"
64          root="0" start_time="0.032043"
65          finish_time="0.032118">
66          <hieraMPI operation="broadcast"
67              file="Topology.cc" line="2335" count="1"
68              type="MPI_INT" root="0"
                comm="OOPS_Comm1"
69              start_time="0.032043"
                finish_time="0.032112"/>
70      </hieraOOPS>
71  </hieraOOPS>
72  …
73  </processor>
```

The hierarchical structure of the file can be observed in lines 32 to 40. The call to the method `bcast` of class `TopologyGrid` (line 32) results in a call of method `bcast` of class `Group` (line 36), that calls the `hieraMPI` element `broadcast` (line 40).

4.2 HieraTransform for MPI and OOPS

To demonstrate analysis tools based on the described graph structure generated from the collected files, some applications were developed to extract some statistical performance data. One of them counts the number of operations of each type in each hierarchical level; another computes the number of the operations of each type discriminating by processor and between MPI and OOPS operations; a third application simply collects the total number of each operation executed; and lastly, an application that computes the total communication and computation times for each processor.

4.2.1 Evaluating the MPI molecular dynamics program

Table 1 shows the number of operations in hierarchical level 0 of the MPI application (the only one present in this case). The process column shows the processes involved in the application execution. The level column shows the level in the hierarchy, in this case, there is just one level 0. The count column shows the number of operations in the level 0. All processes execute about the same number of operations, with process 0 executing about 10% more than the others.

Table 2 gives more information, listing the type of collection (all MPI in this case), the operation executed, file and line number of the call and number of times the operation was called. Due to the amount of collected information, only operations of process 0 are shown in the table. The reduce operations in lines 171 and 175 of file `md.c` are the most executed operations by process 0.

Table 1 Number of operations in the MPI version of the molecular dynamics program

Process	*Level*	*Count*
0	0	2,396
1	0	2,144
2	0	2,144
3	0	2,144

Table 2 Number of operations listed by individual operation in the MPI version of the molecular dynamics code

Category	*Operation*	*File*	*Line*	*Count*
mpi	allgather	md.c	107	1
mpi	allgather	md.c	108	1
mpi	allgatherv	md.c	137	1
mpi	allgatherv	md.c	139	1
mpi	allgatherv	md.c	150	200
mpi	allgatherv	md.c	156	200
mpi	broadcast	md.c	37	1
mpi	broadcast	md.c	38	1
mpi	broadcast	md.c	39	1
mpi	broadcast	md.c	40	1
mpi	broadcast	md.c	41	1
mpi	broadcast	md.c	42	1
mpi	broadcast	md.c	43	1
mpi	broadcast	md.c	44	1
mpi	broadcast	md.c	45	1
mpi	comm_split	md.c	101	1
mpi	comm_split	md.c	102	1
mpi	receive	auxfmd.c	51	9
mpi	receive	auxfmd.c	90	180
mpi	receive	auxfmd.c	92	180
mpi	reduce	md.c	171	400
mpi	reduce	md.c	175	400
mpi	reduce	md.c	183	200
mpi	reduce	md.c	184	200
mpi	reduce	md.c	185	200
mpi	reduce	md.c	186	200
mpi	scan	md.c	95	1
mpi	send	auxfmd.c	58	9
mpi	type_commit	md.c	93	1
mpi	type_contiguous	md.c	92	1

Table 3 shows the same information (for all processes), but collapsed by operation type. It can be seen that the code relies heavily on reduction operations.

Table 3 Number of operations of each type, for the MPI version of the molecular dynamics code

Operation	*Count*
allgather	8
allgatherv	1,608
broadcast	36
comm_split	8
receive	378
reduce	6,400
scan	4
send	378
type_commit	4
type_contiguous	4

Finally, Table 4 shows communication and total execution times for the different processes. Note that communication time is a significant fraction of total time.

Table 4 Total execution times and communication times (in seconds) for the four processes of the MPI molecular dynamics code

Process	*Communication*	*Total*
0	0.495700	1.784300
1	0.915030	1.753440
2	0.991050	1.753510
3	0.972300	1.753490

To give an idea of the influence of the instrumentation on execution time, Table 5 presents the total execution times for the application with and without instrumentation code. The difference is about 6%.

Table 5 Influence of the instrumentation code on execution time of the MPI version of the molecular dynamics code (times in seconds)

Process	*Instrumented*	*Not instrumented*
0	1.784300	1.676900
1	1.753440	1.655580
2	1.753510	1.655580
3	1.753490	1.655630

4.2.2 Evaluating the OOPS molecular dynamics program

Now, the same analysis is made for the OOPS version of the molecular dynamics code (the same algorithm, but implemented using OOPS primitives).

Table 6 shows the number of operations executed in each of the abstraction levels 0, 1 and 2. Note the much higher number of operations than the MPI code. This is due to the fact that the current version of OOPS has no reduce operation over sections of arrays (like present in MPI), and therefore, the reductions must be executed in a loop for each particle.

Table 6 Number of operations in each hierarchical level for the OOPS molecular dynamics code

Process	*Level 0*	*Level 1*	*Level 2*
0	101,756	165,476	100,811
1	101,756	100,948	100,811
2	101,756	100,948	100,811
3	101,756	100,948	100,811

Table 7 discriminates the number of operations for process 2 by operation type and point of call. Note how most operations are MPI reduce operations (a total of 100,000), executed at the request of the `sum` operations at lines 181 and 188 of `molecularDynamicsOOPS.cc`.

The results for all processes summarised by operation type are shown in Table 8. Note how the operation count is dominated by operations `sum` (and the corresponding `reduce`), followed by the `localSize` and `localToGlobal` operations, responsible for the verification of the sizes of local parts of arrays and conversion from local to global array indexes, respectively.

Finally, the effect of instrumentation on execution time of the code is shown in Table 9. It can be seen that the overload is of about 0.8%. Note that this is a small value, despite the high number of registered operations.

5 Conclusions

Due to the increasing use of higher level abstractions for the development of parallel codes, it is important that performance tools consider these levels and are able to generate information at the level understood by the application developer.

This article presented the *HieraAnalyses* tool, developed to probe the feasibility of such kind of tools. The tool is composed of a collector model *hieraCollector* and a transformation module *hieraTransform*. The collector module writes performance information as an XML file with hierarchical structure following the hierarchical call structure of the execution. The transformation module builds a graph from the collected data, upon which various performance analyses may be carried out.

The article described further the application of the tool to the analysis of a parallel molecular dynamics code written in two versions: one using only MPI operations and other using the OOPS framework, a high level framework implemented using MPI. It was shown that important information about the program execution and the parts of the code that require attention can be deduced from the graph structure that represents the performance information.

A deeper use of the collected hierarchical information involves the visualisation of the information following the hierarchical structure present in the data. This may enable the user to find the important sections of the code in a top-down approach. This is a suggestion for future work.

Table 7 Discrimination of the number of operations for process 2 of the OOPS molecular dynamics code

Category	*Operation*	*File*	*Line*	*Count*
mpi	allgather	Topology.cc	2,655	2
mpi	allgatherv	Topology.cc	2,622	2
mpi	allreduce	Topology.cc	3,672	200
mpi	allreduce	Topology.cc	3,837	600
mpi	broadcast	Topology.cc	2,335	5
mpi	broadcast	Topology.cc	2,423	3
mpi	broadcast	Topology.cc	2,434	1
mpi	reduce	Topology.cc	3,892	100,000
mpi	scan	Topology.cc	4,202	1
oops	broadcast	molecularDynamicsOOPS.cc	45	1
oops	broadcast	molecularDynamicsOOPS.cc	46	1
oops	broadcast	Topology.cc	2,423	3
oops	broadcast	Topology.cc	2,434	1
oops	distributionBlocked	Application	0	1
oops	gather	Topology.cc	2,622	2
oops	gather	Topology.cc	2,655	2
oops	gather	Topology.h	3,019	60
oops	load	Vector.h	1,606	4
oops	localSize	Vector.h	1,285	4
oops	localSize	Vector.h	1,531	4
oops	scan	molecularDynamicsOOPS.cc	105	1
oops	scan	Topology.cc	4,202	1
oops	scatter	Vector.h	1,531	4
oops	scatter	Topology.h	3,045	4
oops	split	Topology.cc	2,221	2
oops	split	TopologyGrid.cc	4,273	1
oops	store	Vector.h	1,613	60
oops	sum	molecularDynamicsOOPS.cc	181	50,000
oops	sum	molecularDynamicsOOPS.cc	188	50,000

Table 8 Operation count for all processes of the OOPS version of the molecular dynamics code

Operation	*Count*	*Operation*	*Count*
allgather	8	load	16
allgatherv	8	localSize	32,800
allreduce	3,200	localToGlobal	32,016
broadcast	36	scatter	32
reduce	400,000	split	16
scan	12	store	240
broadcast	72	sum	806,400
distribBlocked	4	topologyGrid	4
gather	3,712	topologyPipe	12

Table 9 Execution times (in seconds) with and without instrumentation for the OOPS version of the molecular dynamics code

Process	*Instrumented*	*Not instrumented*
0	239.407456	237.457
1	239.288433	237.458
2	239.288600	237.459
3	239.288144	237.471

Acknowledgements

The work of T.F.P. Seraphim was supported by National Council for Scientific and Technological Development (CNPq) under Grant Number 140453/2003-2.

References

Blackford, L.S., Dongarra, J.J. and Whaley, R.C. (1997) 'ScalaPACK user's guide', *Siam – Society for Industrial and Applied Mathematics*, May, ISBN: 0-89871-397-8.

Browne, S., Dongarra, J. and London, K. (1998) 'Revier of performance analysis tools for MPI parallel programs', *NHSE Review*, Vol. 3, No. 1.

Browne, S., Dongarra, J., Garner, N., London, K. and Mucci, P. (2000) 'A scalable cross-plataform infrastructure for application performance tuning using hardware counters', *Proceedings of the ACM/IEEE Conference on Supercomputing*, IEEE Computer Society, p.42.

Cohen, W.E., Garrett, W.D. and Gaede, R.K. (2007) 'Parallel program traces for accurate prediction of proposed cluster performance', *CiteSeerX – Scientific Literature Digital Library and Search Engine*.

Fineman, C., Frumkin, M., Hontalas, P., Hribar, M. and Jin, H. (1997) *The Automated Instrumentation and Monitoring System*, available at http://www.nas.nasa.gov/Groups/Tools/Projects/AIMS/manual/TableTest.html (accessed on April/09, January).

Geist, A., Beguelin, A., Dongarra, J., Jiang, W., Manchek, R. and Sunderam, V. (1994) *PVM Parallel Virtual Machine, A User's Guide and Tutorial for Networked Parallel Computing*, MIT Press.

Graham, S.L., Kessler, P.B. and Mckusic, M.K. (1982) 'gprof: a call graph execution profiler', *SIGPLAN Symposium on Compiler Construction*, June, pp.120–126.

Hollingsworth, J.K., Irvin, R.B. and Miller, B.P. (1991) 'The integration of application and system based metrics in a parallel program performance tool', *Proceedings on the 3rd ACM SIGPLAN Symposium on Principles & Practice of Prallel Programming*, SINGPLAN Notices, April, Vol. 26, No. 7, pp.189–200.

Labarta, J., Gimenez, J., Caubet, J. and Escale, F. (2001) 'Paraver: parallel program visualization and analysis tool', *European Center for Parallelism of Barcelona*, October, Version 3.1.

Miller, B.P., Callaghan, M.D., Cargille, J.M., Hollingsworth, J.K., Bruce, R., Karen, I., Karavanic, L., Kunchithapadam, K. and Newhall, T. (1995) 'The paradyn parallel performance measurement tools', *IEEE Computer*, pp.37–46.

Miller, B.P., Clark, M., Hollingsworth, J.K., Kierted, S., Lim, S. and Torzewski, T. (1990) 'IPS-2: the second generation of a parallel program measurement system', *IEEE Transactions on Parallel and Distributed Systems*, February, Vol. 1, No. 2, pp.206–217.

Moore, S., Cronk, D., London, K. and Dongarra, J. (2001) 'Review of performance analysis tools for MPI parallel programs', *8th European PVM/MPI Users' Group Meeting*, pp.241–248.

Message Passing Interface (MPI) Forum (1994) 'MPI: a message-passing interface standard', University of Tennessee, Knoxville, TN, USA, June.

Plimpton, S. (1995) 'Fast parallel algorithms for shortrange molecular dynamics', *Journal of Computational Physics*, March, Vol. 117, No. 1, pp.1–19.

Reed, D.A., Aydt, R.A., Noe, R.J., Shields, K.A., Schwartz, B.W. and Tavera, L.F. (1993) 'Scalable performance analysis: the Pablo performance analysis environment', *Proceedings of the Scalable Parallel Libraries Conference*, IEEE Computer Society, pp.104–113.

Rodrigues, F.A. (2004) 'Técnicas de Orientacão ao objeto para computacão científica paralela', Master's thesis, Instituto de Física de São Carlos – Universidade de São Paulo, April.

Sonoda, E. and Travieso, G. (2006) 'The OOPS framework: high level abstractions for the development of parallel scientific applications', *OOPSLA'06: Companion to the 21st ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pp.659–660.

Truoung, H. and Fahringer, T. (2003) 'SCALEA: a performance analysis tool for parallel programs', *Concurrency and Computation: Practice and Experience*, Vol. 15, Nos. 11–12, pp.1001–1025.

W3C (2009) *W3C: World Wide Web Consortium*, available at http://www.w3c.org (accessed on April/09).

Yan, J.C. (1994) 'Performance tuning with AIMS – an automated instrumentation and monitoring system for multicomputers', *Proceedings of the 27th Hawaii International Conference on System Sciences*, January, Vol. II, Nos. 4–7, pp.625–633.