GASP: A Performance Tool Interface for Global Address Space Languages & Libraries

Adam Leko1, Dan Bonachea2, Hung-Hsun Su1, Bryan Golden1, Hans Sherburne1, Alan D. George1

1 Electrical Engineering Dept., University of Florida
2 Computer Science Dept., UC Berkeley

Outline

- Introduction
- GASP Overview
- GASP Interface
- GASP Overhead
- Conclusions
- Future Directions
Global Address Space (GAS) Languages/Libraries

- Unified Parallel C (UPC), Co-Array Fortran, Titanium, SHMEM, etc.

Properties:
- Provides a shared address space abstraction
- Includes one-sided communication operations (put/get)

- Available for a wide range of system architectures (both shared-memory machines and distributed systems) in the form of compiler or library
- Implementation’s internals may vary greatly from one system to another for the same language
- Performance can be comparable with MPI code
- But, generally requires hand-tuning
  - Performance tool support would help greatly

Motivation for Tool Interface (1)

- Minimal performance tool support for GAS programs. Why is that?
  - Newer languages/libraries
  - Complicated compilers
    - Take UPC for example, several different implementation strategies
      - Direct compilation (GCC-UPC, Cray UPC)
      - Translator + Library approach (Berkeley UPC w/GASNET, HP UPC)
    - One-sided memory operations tracking support
    - Shared-data tracking support
Motivation for Tool Interface (2)

- Performance tool support strategies
  - Direct source instrumentation
    - May prevent compiler optimizations/reorganization
    - How to deal with relaxed memory model?
  - Binary instrumentation
    - Not available on some architectures
    - Difficult to relate back to source code
  - Intermediate libraries
    - Wrappers for functions/procedures
    - Does not work for “pure compilers”
  - Performance interface
    - Defines basic interaction between compiler and performance tool
    - Up to compiler developers to decide how to best incorporate the interface (wrapper, translation, etc.)

Overview (1)

- Global Address Space Performance (GASP) interface
  - Event-based interface
    - GAS compiler/runtime communicate with performance tools using standard interface
    - Performance tool is notified when particular actions happen at runtime
    - Implementation-agnostic
  - Notification structure
    - Function “callback” to tool developer code
    - Use a single function name (gasp_event_notify)
      - Pass in event ID and source code location
      - Use varargs for rest of arguments (like printf)
    - Notifications can come from compiler/runtime (system events) or from code (user events)
    - Allows calls to the source language/library to make model-specific queries
Overview (2)

```
enum gasp_event_type {gasp_event_type_start, gasp_event_type_end,
gasp_event_type_atomic};

void gasp_event_notify(
    unsigned int event_id,
    enum gasp_event_type event_type,
    const char* source_file,
    unsigned int source_line,
    unsigned int source_col,
    ...);
```

System Event Types (1)

- All system events have symbolic names defined in `gasp_[language].h`
- Startup and shutdown
  - Initialization called by each process after GAS runtime has been initialized
  - Exit called before all threads stop (two types of events: collective exit & non-collective exit)
- Synchronization
  - Fence, notify, wait, barrier start/end
  - Lock functions
System Event Types (2)

- Work sharing
  - Forall start/end
- Collective events
  - Broadcast, scatter, gather, etc.
- Shared variable access
  - Direct (through variable manipulations)
  - Indirect (through bulk transfer functions)
  - Non-blocking operations
- User functions
  - Beginning and end of desirable user functions

User-Defined Event Type

- Allow user to give context to performance data
- Can be used for
  - Instrumenting individual loops or regions in user code
  - Phase profiling
  - Hand instrumentation
- Simple language-independent API
  - `gasp_create_event()` creates an event with a description
  - `gasp_event_start()`, `gasp_event_end()` notify tool of region entry/exit
  - Event start/end functions also take a variable number of arguments (printf-style display) inside performance tool
Instrumentation & Measurement Control

- Provide finer instrumentation and measurement control
- **--profile** flag
  - Instructs compiler to instrument all events for use with performance tool
  - Compiler should instrument all events, except
    - Shared local accesses
    - Accesses that have been privatized through optimizations
- **--profile-local** flag
  - Instruments everything as in --profile, but also includes shared local accesses
- **#pragma pupc [on / off]** directive
  - Controls instrumentation during compile time, only has effect when **--profile** or **--profile-local** have been used
  - Instructs compiler to avoid instrumentation for specific regions of code, if possible
- `pupc_control(int on);` function call
  - Controls measurement during runtime done by performance tool

Vendor Support

- **UPC**
  - Berkeley UPC
    - GASP implemented within runtime library
    - Supported with Berkeley UPC 2.4+
      - **--enable-profile** configure-time option
  - HP UPC
    - HP verbally agreed to support GASP at UPC ’05 workshop
    - Unfortunately, GASP work has been pushed back at the moment
  - Cray UPC, MuPC, GCC-UPC
    - GASP support pending
- **Others**
  - Titanium GASP support is in the pipeline
  - Support for other languages/libraries pending
Berkeley UPC GASP Tracing Overhead
(Splash-2 LU)

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>I (Local)</th>
<th>I+M</th>
<th>I+M (PAPI)</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Overhead</td>
<td>0</td>
<td>5</td>
<td>10</td>
<td>15</td>
</tr>
</tbody>
</table>

I – instrumentation only, empty calls (with or without instrumentation for local events)
I+M – actual events are recorded through a preliminary measurement layer
PAPI – measurement layer records PAPI hardware counter events

* Overhead expected to be much lower when replacing gettimeofday() with a high-performing timer

All tests executed on a dual 2.4GHz opteron cluster (32 nodes) with Quadrics interconnect

Berkeley UPC GASP Profiling Overhead

<table>
<thead>
<tr>
<th></th>
<th>CG</th>
<th>EP</th>
<th>FT</th>
<th>IS</th>
</tr>
</thead>
<tbody>
<tr>
<td>% Overhead</td>
<td>0.5</td>
<td>1</td>
<td>1.5</td>
<td>2</td>
</tr>
</tbody>
</table>

*All results are for NAS benchmark 2.4 class A unless noted otherwise
Conclusions

- GASP specifies a standard event-based performance interface for GAS languages/libraries
- Preliminary version of GASP includes UPC support (w/ low overhead, working implementation available for Berkeley UPC)
- Performance interface should be an integral part of a language/library design & implementation effort
  - Fairly straightforward for compiler developers to add support
  - Avoid interference with compiler optimization
  - But how do we get language/library implementer’s support?
- GASP is integrated with a new performance analysis tool (Parallel Performance Wizard) we are currently developing for UPC and SHMEM
- Specifications for other GAS languages/libraries are forthcoming
  - May even extend beyond GAS languages/libraries to include other parallel languages/libraries such as OpenMP, MPI-2, X10, Fortress, Chapel, etc.
- For more info, see
  - http://docs.hcs.ufl.edu/upc/gasp/
  - http://docs.hcs.ufl.edu/upc/gasp/ChangeLog
Future Directions

- GASP serves as a starting point for generic parallel performance analysis
- We are currently investigating the possibility of a generic parallel performance analysis approach that deals with event types rather than language constructs/library functions
  - Execution model does not differ significantly between parallel languages/libraries
  - Once a generic set of analyses is developed, it should be applicable to all languages/libraries
  - Adding performance analysis support for a new language/libraries simplifies to
    - Enabling instrumentation and measurement of events (i.e. GASP)
    - Creating a mapping of language constructs/library functions to event types
    - Small modification to visualizations to better present the result
  - Analysis of program involving multiple languages/libraries is possible

<table>
<thead>
<tr>
<th>Language construct / Library function (ex: upc_memget)</th>
<th>Mapping of constructs/functions to pre-defined event types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Event Type (ex: Data transfer: asyn.)</td>
<td>Perform analysis based on event type only</td>
</tr>
<tr>
<td></td>
<td>Substitute back-in the language construct library function</td>
</tr>
<tr>
<td>Analysis result</td>
<td></td>
</tr>
<tr>
<td>Visualization/User interface</td>
<td></td>
</tr>
<tr>
<td>Programming model independent</td>
<td></td>
</tr>
<tr>
<td>Programming model specific</td>
<td></td>
</tr>
</tbody>
</table>

Event ID

| Event type (ex: data transfer) | Sub-event type (ex: asyn_get) | Flags |