OpenMP-centric Performance Analysis of Hybrid Applications

Karl Fürlinger [1,2], Shirley Moore [1]

[1] *Innovative Computing Laboratory*
EECS Department, University of Tennessee, Knoxville
37996 Knoxville, TN, U.S.A.
[2] *Computer Science Division*
EECS Department, University of California, Berkeley
94720 Berkeley, CA, U.S.A.
[1] {karl,shriley}@eecs.utk.edu [2] fuerling@eecs.berkeley.edu

***Abstract*—Several performance analysis tools support hybrid applications. Most originated as MPI profiling or tracing tools and OpenMP capabilities were added to extend the performance analysis capabilities for the hybrid parallelization case. In this paper we describe our experience with the other path to support both programming paradigms. Our starting point is a profiling tool for OpenMP called ompP that was extended to handle MPI related data. The measured data and the method of presentation follow our focus on the OpenMP side of the performance optimization cycle. For example, the existing overhead classification scheme of ompP was extended to cover time in MPI calls as a new type of overhead.**

I. Introduction

The widespread adoption of multicore CPU designs in current and upcoming parallel computing systems of all sizes has far-reaching effects on how those systems can optimally be utilized by domain scientists to advance their research.

For application developers, multicore means that they are faced with an increasingly large number of computing cores arranged in hierarchies (e.g., core, chip, SMP node, NUMA node, and cluster) that share different types of resources. It has been suggested that a flat MPI model might not be the best fit for such machines with deep hierarchies and that instead the programming paradigm should follow the hierarchical principle too. The most promising choices for such a hierarchical combination are MPI as a distributed computing paradigm and OpenMP for shared memory parallelism.

Several performance analysis tools support hybrid applications. Most originated as MPI profiling or tracing tools and OpenMP capabilities were added to extend the performance analysis capabilities for the hybrid parallelization case. In this paper we describe our experience with the other path to support both programming paradigms. Our starting point is a profiling tool for OpenMP called ompP that was extended to handle MPI related data. The measured data and the method of presentation follow our focus on the OpenMP side of the performance optimization cycle. For example, the existing overhead classification scheme of ompP was extended to cover time in MPI calls as a new type of overhead.

The rest of this paper is organized as follows. In Sect. II-A we first describe the existing capabilities of ompP. The extensions for MPI monitoring and data presentation are then discussed in Sects. II-B1 and II-B3, respectively. Sect. III shows examples of how the delivered data can be utilized in various hybrid programming scenarios. We discuss related work in Sect. IV and give an outlook on planed future extensions in Sect. V.

II. Performance Analysis of Hybrid Applications

A. OpenMP Profiling with ompP

ompP is a profiling tool for OpenMP applications designed for Unix-like systems. ompP differs from other profiling tools like gprof [1] or OProfile [2] in primarily two ways. Firstly, ompP is a measurement-based profiler and does not use program counter sampling. The instrumented application invokes ompP monitoring routines that enable a direct observation of program execution events (like entering or exiting a critical section). An advantage of the direct approach is that the results give exact counts, rather than sampled values, and hence they can also be used for correctness testing, for example to check that a critical section is indeed executed a prescribed number of times.

The second difference lies in the way of data collection and representation. While general profilers work on the level of functions, basic blocks, or statements, ompP collects and displays performance data in the user model of the execution of OpenMP events [3]. For example, the data reported for critical section contains not only the execution time but also lists the time to enter and exit the critical construct (enterT and exitT, respectively) as well as the accumulated time each thread spends inside the critical construct (bodyT) and the number of times each thread enters the construct (execC). An example profile for a critical section extended with MPI-level profiling data is given in Fig. 1.

B. Adding Support for MPI monitoring in ompP

As ompP is a tool for OpenMP, profiling data is gathered on a per-thread basis for OpenMP constructs, functions and user-defined regions. The entire application is implicitly assumed to consist of one process that hosts the threads of execution. For MPI, the execution model prescribes that the

```
R00002 main.c (20-23) (unnamed) CRITICAL
 TID    execT    execC    bodyT    enterT    exitT
   0     1.00        1     1.00      0.00     0.00
   1     3.01        1     1.00      2.00     0.00
   2     2.00        1     1.00      1.00     0.00
   3     4.01        1     1.00      3.01     0.00
 SUM    10.02        4     4.01      6.01     0.00
```

Fig. 1: Profiling data delivered by ompP for a critical section.

parallel application consists of a number of processes which communicate with messages. Since there are already a number of tools (e.g., TAU [4], Vampir [5], Scalasca [6], mpiP [7]) that can give the user a great deal of information about the messaging characteristics of the application, we have decided to limit ourselves to MPI extensions that represent an added benefit for an OpenMP-centric performance analysis instead of replicating the functionality of the existing tools. Our goals is to give rough MPI-level data that is anchored in the OpenMP-level of thinking of application development, more sophisticated MPI tools can then be used to drill deeper into the MPI data if this is desired.

From the OpenMP standpoint, a hybrid application can be seen as being comprised of a number of copies of an OpenMP-parallel processes that happen to use MPI mechanisms for work distribution, synchronization, and result gathering. Questions such as

- "Do the application OpenMP process copies behave uniformly or are there significant differences between them?",
- "If there are differences, where do they come from?",
- "Can the variance be related to the MPI-level work assignment (domain decomposition) or does it have other reasons?",
- "Which OpenMP level overheads do the application processes exhibit and what is their source?",

are the interesting in this context and hard to answer with existing tools.

1) Monitoring Support for MPI: Similar to other MPI tools, ompP uses the profiling interface to interpose the monitoring library between the application code and the MPI library. OmpP monitors data like the number of calls, the send and receive volume and the time spent in MPI calls. To enable a thread safe monitoring of the MPI activity, we have chosen to follow the following approach: much like there is a continuously increasing value of a "time" variable that is sampled on entry (t_{start}) and exit (t_{end}) from routines, and used to increment the accumulated overall execution time by (t_{end}-t_{start}), ompP keeps thread local MPI-related counters that are incremented in the MPI monitoring calls.

ompP determines the receive and send volume (more on this below) and the time spent in the MPI calls. Then, as shown in an example in Fig. 2, the counters for sent data volume in, received data volume, MPI time and execution count are updated for the thread that executes the MPI call.

The OpenMP profiling code reading wall-clock timestamps

```
int MPI_Barrier(MPI_Comm comm) {
  ...
  tid = THREADID();

  t_start = TIMESTAMP();
  res = PMPI_Barrier(comm);
  t_end = TIMESTAMP();

  n_coll[tid]++;
  t_MPI[tid] += t_end-t_start;

  return res;
}
```

Fig. 2: MPI related counter variables (n_{coll}, t_{MPI}) are incremented in the MPI monitoring call of ompP.

```
void region_enter(reg_t reg) {
  ...
  tid = THREADID();
  reg.t_exec^enter[tid]=TIMESTAMP();
  reg.t_MPI^enter[tid]=t_MPI[tid];
  reg.vol_send^enter[tid]=vol_send[tid];
  reg.vol_recv^enter[tid]=vol_recv[tid];
  ...
}

void region_exit(reg_t reg) {
  ...
  tid = THREADID();
  reg.t_exec[tid]+=(TIMESTAMP()-
                    reg.t_exec^enter[tid]);
  reg.t_MPI[tid]+=(TIMESTAMP()-
                    reg.t_MPI^enter[tid]);
  reg.vol_send[tid]+=vol_send[tid]-
                    reg.vol_send^enter[tid];
  ...
}
```

Fig. 3: The OpenMP monitoring code of ompP was extended to read the MPI variables (vol_{send}, t_{MPI},...) in addition to reading timestamps and hardware counter data.

and hardware counter values was subsequently modified to also read the MPI related counters. An example of this is shown in Fig. 3. In the region_enter call the current values are recorded in region-local variables such as t_{MPI}^{enter}[tid]. In the region_exit call the recorded enter values are subtracted from the current values and the difference is used to update the region statistics.

As the same thread either executes an MPI_* call (where data is updated or incremented) or a call related to the OpenMP monitoring (where data is read) there is no danger of a race condition.

The technique described above allows the user to correlate

MPI activity to the constructs that are monitored by ompP. Those are all OpenMP constructs (like parallel regions and critical sections) and functions, other regions can be instrumented manually by the user to show up in ompP's profiling report.

2) Determination of the Data Volume: Almost all existing MPI performance analysis tools have the capability to report sent and received data volume on a per process basis. For point to point communication operations the derivation of these quantities is straight forward, but for collectives, the *actual* amounts of data sent or received depend on the implementation and are not observable from the PMPI profiling layer interface. Consider for example an MPI_Bcast call of n bytes in a communicator with p ranks. Clearly, each process (except the root) has to receive the n bytes, but how much data is sent by each process? MPI_Bcast could be implemented by $p-1$ sends originating at the root rank, in which case the root rank would have a send volume of $n \times (p-1)$. In another scenario, MPI_Bcast could be implemented in a pipeline fashion, where each process sends the data to exactly one neighbor processor until it reaches the last one. In this case each rank would only have a send volume of n bytes.

With most MPI library implementations, the broadcast operation will actually be implemented using a tree, but at the level of the MPI profiling layer such details are not visible. Practically, the only two options for determining send and receive volumes for collective operations are a *naive* scenario which assumes a simplistic implementation ($p-1$ point-to-point operations in the broadcast example) and a *minimal* setting. The minimal approach takes only into account data transfers that *have* to occur logically, from a local per-process perspective. In the broadcast example, the data *has* to appear at each process, hence the receive count is n, the data also *has* to leave the root node, hence the send volume is n at the root node.

The table in Fig. 4 shows which MPI calls are monitored by ompP and which data volumes are reported. dsize corresponds to the data size, i.e., the size of the MPI_Datatype used times the count argument. dsize is used if only one data type and count is used, while rsize and ssize are used when two data types and counts can be specified in the call (one for receiving and one for sending, respectively). The data in the table is meant to be interpreted locally, i.e., a process that execute an MPI call determines the data volume locally according to the table. The "v" variants and MPI_Reduce_scatter allow the specification of varying amounts of data sent/received for peer processes. $dsize[i]$ corresponds to the data volume for process i in this case.

A detailed comparison of the accounting schemes implemented in other tools is beyond the scope of this paper. For simple calls, we have discovered no difference between our *naive* accounting scheme with other MPI tools, while for more complicated calls such as MPI_Scan such differences exist.

3) Result Presentation: OmpP's profiling report output was modified in several places to display the MPI related data. The header of ompP's profiling report now gives overall MPI

```
Start Date      : Wed Apr 23 14:39:57 2008
End Date        : Wed Apr 23 14:40:16 2008
Duration        : 18.29 sec
Application Name: smg2000.ompp
Type of Report  : final
User Time       : 25.91 sec
System Time     : 10.32 sec
Max Threads     : 2
ompP Version    : 0.6.99
ompP Build Date : Apr 23 2008 14:38:30
PAPI Support    : not available
MPI my rank     : 1
MPI num procs   : 2
MPI hostname    : battlecat1
MPI time        : 11.028286
MPI bytes in    : 3381964
MPI bytes out   : 3308644
MPI recv calls  : 20700
MPI send calls  : 20700
MPI collectives : 17
```

Fig. 5: The profiling report header with data related to the MPI execution.

information such as the total number of MPI processes and the rank and host name of the writing process (cf. 5). Currently, each MPI process writes a separate profiling report and a set of utilities are provided that take a group of reports and output MPI related data as described in this section and Sect. III. The header also lists the total time in MPI routines, the number of MPI calls broken down into sends, receives and collectives, and the overall data volume that is sent and received.

In the basic flat and callgraph region profiles, regions having MPI activity now have additional columns detailing how many times MPI calls were executed, how much time was spent in MPI routines, and how much data was received and sent. An example profile of a critical section containing an MPI_Send call is shown in Fig. 6. The parallel region containing the critical section is executed 10 times (cf. the execC column) with a thread team of four threads, each MPI_Send operation transmits a megabyte and the ten total send operations (sendC column) per thread thus generate a send volume of 10 megabytes (outV column), the total send volume for this process from this critical section is 40 megabytes.

As mentioned earlier, our goal was to correlate the MPI behavior to the OpenMP-level activity. Compared to existing MPI tools this gives less detailed MPI data. There is, for example, no differentiation between different MPI calls in the same region and no breakdown more precise than into sends, receives, and collectives. If a user desires this more detailed knowledge, an existing MPI tool can be used. With ompP, a method to achieve a similar effect is to put user-defined instrumentation around MPI calls. For example the code block

```
void foo() {
  MPI_Bcast( )
...
  MPI_Gather()
```

		naive		*minimal*	
Operation	Rank(s)	Send Volume	Receive Volume	Send Volume	Receive Volume
MPI_Send and variants		$dsize$	0	$dsize$	0
MPI_Recv and variants		0	$dsize$	0	$dsize$
MPI_Sendrecv		$ssize$	$rsize$	$ssize$	$rsize$
MPI_Sendrecv_replace		$dsize$	$dsize$	$dsize$	$dsize$
MPI_Bcast	root other	$dsize \times (np-1)$ 0	0 $dsize$	$dsize$ 0	0 $dsize$
MPI_Scatter	root other	$ssize \times np$ 0	$rsize$ $rsize$	$ssize \times np$ 0	$rsize$ $rsize$
MPI_Gather	root other	$ssize$ $ssize$	$rsize \times np$ 0	$ssize$ $ssize$	$rsize \times np$ 0
MPI_Scatterv	root other	$\sum ssize[i]$ 0	$rsize$ $rsize$	$\sum ssize[i]$ 0	$rsize$ $rsize$
MPI_Gatherv	root other	$ssize$ $ssize$	$\sum rsize[i]$ 0	$ssize$ $ssize$	$\sum rsize[i]$ 0
MPI_Allgather	all	$ssize \times np$	$rsize \times np$	$ssize$	$rsize \times np$
MPI_Allgatherv	all	$ssize \times np$	$\sum rsize[i]$	$ssize$	$\sum rsize[i]$
MPI_Reduce	root other	0 $dsize$	$dsize \times (np-1)$ 0	0 $dsize$	$dsize$ 0
MPI_Reduce_scatter	all	$\sum_{j \neq i} dsize[j]$	$dsize[i] \times (np-1)$	$\sum_{j \neq i} dsize[j]$	$dsize[i]$
MPI_Scan	0 $1..last-1$ $last$	$dsize$ $dsize$	0 $dsize$ $1..last-1$ $dsize$ $last$	$dsize$ $dsize$	 $dsize$ $dsize$
MPI_Allreduce	all	$dsize \times (np-1)$	$dsize \times (np-1)$	$dsize \times (np-1)$	$dsize \times (np-1)$
MPI_Alltoall	all	$ssize \times np$	$rsize \times np$	$ssize \times np$	$rsize \times np$
MPI_Alltoallv	all	$\sum ssize[i]$	$\sum rsize[i]$	$\sum ssize[i]$	$\sum rsize[i]$
MPI_Barrier	all	time only			
MPI_Probe	all	time only			
MPI_Pack, MPI_Unpack	all	time only			
MPI_Wait and variants	all	time only			

Fig. 4: The MPI calls monitored by `ompP` and definition of the send and receive volume with two accounting settings naive, and minimal. np is the number of processes in the communicator, $dsize$ is used to designate the data volume (size of the datatype times number of elements). If two datatypes can be specified, the according sizes are called $rsize$ and $ssize$, for receiving and sending, respectively.

```
R00005 main.c (53-58) (unnamed) CRITICAL
 TID  execT  execC  bodyT  enterT  exitT   mpiT   inV       outV  recvC  sendC  collC
   0   0.41     10   0.10    0.31   0.00   0.10     0   10485760      0     10      0
   1   0.35     10   0.09    0.26   0.00   0.09     0   10485760      0     10      0
   2   0.43     10   0.09    0.33   0.00   0.09     0   10485760      0     10      0
   3   0.41     10   0.08    0.33   0.00   0.08     0   10485760      0     10      0
 SUM   1.60     40   0.37    1.23   0.00   0.37     0   41943040      0     40      0
```

Fig. 6: Example (flat) region profile for a critical section with MPI data. `mpiT` is the time spent in MPI routines, `outV` is the data volume sent (in bytes), `inV` is the received data volume. `recvC`, `sendC`, `collC` are, respectively, the number of send, receive, and collective communication operations.

}

can be changed to

```
void foo() {
#pragma pomp inst begin(bcast)
  MPI_Bcast( )
#pragma pomp inst end(bcast)
...
#pragma pomp inst begin(gather)
  MPI_Gather()
#pragma pomp inst end(gather)
}
```

to differentiate between the broadcast and the gather call.

Some MPI tools like mpiP [7] and Scalasca [6] try to determine the MPI callsite by unwinding the stack. This is often technically challenging with respect to portability and reliability and we decided to forgo such an approach for the time being.

Another area where the profiling report was augmented with MPI support is `ompP`'s overhead analysis report. `OmpP` classifies the execution time of an application into useful work and four overhead classes: synchronization, load imbalance, thread management, limited parallelism.

- **Synchronization:** Overheads that arise because threads need to coordinate their activity. An example is the waiting time to enter a critical section or to acquire a lock.
- **Imbalance:** Overhead due to different amounts of work performed by threads and subsequent idle waiting time, for example in work-sharing regions.
- **Limited Parallelism:** This category represents overhead resulting from unparallelized or only partly parallelized regions of code. An example is the idle waiting time threads experience while one thread executes a `single` construct.
- **Thread Management:** Time spent by the runtime system for managing the application's threads. That is, time for creation and destruction of threads in parallel regions and overhead incurred in critical sections and locks for signaling the lock or critical section as available.

Time spent in MPI calls has been added as a new overhead category, **MPI**, in the overhead report. An example overhead report is shown in Fig. 7.

III. Data Analysis

There are multiple ways in which the OpenMP and MPI programming paradigms can be integrated. The complexity as well as the potential synergistic effects vary across those levels as does the applicability of `ompP`'s combined performance analysis model.

Maybe the most common case in existing code is the usage of MPI outside of OpenMP parallel regions. `ompP` will be able to display messaging statistics for the whole program and for user-instrumented regions. It can also be important to discover differences in the OpenMP execution characteristics of the MPI processes. Radar (or spider net) charts can be used to explore the differences between processes. For example, the chart in Fig. 8 shows the total execution time (summed over all threads) of the two most time consuming region in a hybrid execution of the SMG2000 [8] application with 8 processes. Instead of plotting the absolute execution time the plot shows data normalized to the process with the maximum execution time (18.11 seconds and 6.16 seconds, respectively). Evidently the execution time for the residual computation is very uniform accross processes while there are more marked differences in the reduction loop.

The second case is the usage of MPI by a single, dedicated thread inside parallel regions. The OpenMP `master` and `single` constructs can be used to achieve this task.

Consider the following toy example: An application is based on a MPI-level domain decomposition that induces load imbalance among *threads* of a subset of the MPI processes in a parallel work sharing region due to the size or structure of the decomposition. Assume a collective operation such as MPI_Alltoall is necessary after the OpenMP worksharing region. If no load imbalance occurs, processes enter the MPI collective operation earlier, while processes with load imbalance take longer to finish the execution and enter the collective later.

An example of this scenario is shown in the pie charts in Fig. 9. Each pie chart corresponds to one MPI process and shows the breakdown of the overheads. The pie charts show that the overall overheads contribution from this parallel region stays approximately constant and that it is mainly the attribution between time spent in the MPI routine and the OpenMP-level load imbalance that changes.

The third case of combined usage is that any thread may be making MPI calls, but only one at a time. The mutual exclusion can be achieved by using `critical` sections or locks. `ompP` can give detailed information about the number of MPI calls for each thread, the time spent in MPI calls and how those numbers differ from MPI rank to MPI rank.

The fourth case is that any thread can make calls to the MPI library at any time without mutual exclusion. This requires a thread-safe MPI implementation and the provision of some application-level synchronization.

For the future this appears to be a model of increasing importance due to the introduction of the tasking concepts in OpenMP 3.0. A central task pool concept makes the introduction of MPI level parallelism into OpenMP-based code relatively easy [9]. Special tasks can be added that arrange for data being transmitted and received, effectively implementing a model of cross-process task stealing or distribution. A tight integration of the analysis capabilities for OpenMP and MPI is especially important in this case.

IV. Related Work

The range of tools available for monitoring, analyzing, and optimizing MPI applications is large. The most common and powerful tools that originated in an academic environment

```
Overheads wrt. each individual parallel region:
       Total     Ovhds (%) =   Synch (%) +   Imbal (%) +  Limpar (%)    +   Mgmt (%)  +    MPI (%)
R00085 17.79 10.79 ( 9.20)  0.00 ( 0.00)  3.33 ( 3.20)  0.00 ( 0.00)    6.24 ( 6.00)  1.22 ( 0.00)
R00044  5.82  3.34 ( 2.99)  0.00 ( 0.00)  1.09 ( 1.05)  0.00 ( 0.00)    2.02 ( 1.95)  0.23 ( 0.00)
R00047  5.79  4.22 ( 2.99)  0.00 ( 0.00)  1.09 ( 1.04)  0.00 ( 0.00)    2.02 ( 1.94)  1.11 ( 0.00)
...
   SUM 51.67 34.86 (26.69)  0.00 ( 0.00)  9.74 ( 9.37)  0.00 ( 0.00)   18.01 (17.32)  7.11 ( 0.00)
```

Fig. 7: Example overhead analysis report. For each parallel region individually, and for the whole application `ompP` shows a breakdown of the total execution time (summed over threads) into the five overhead classes (Synch, Imbal, Limpar, Mgmt, and MPI).

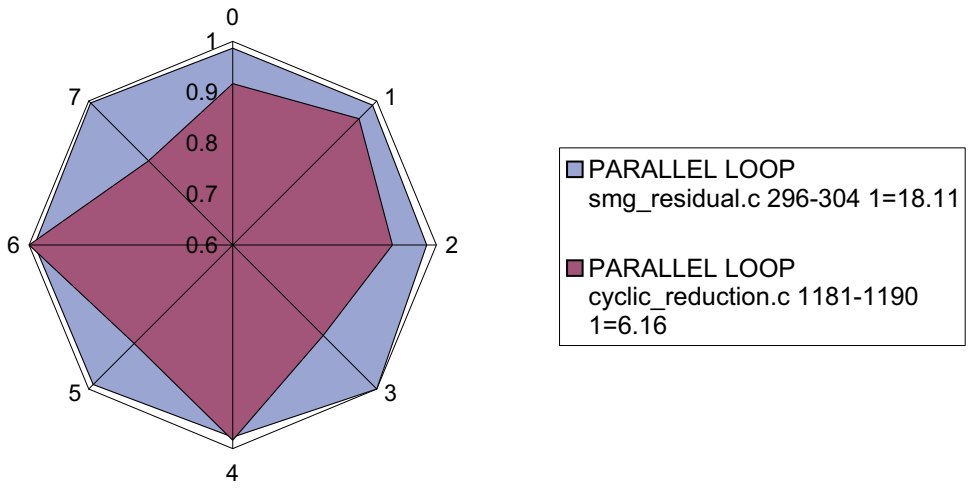

Fig. 8: Example radar chart of the total execution time of two region of the SMG2000 application.

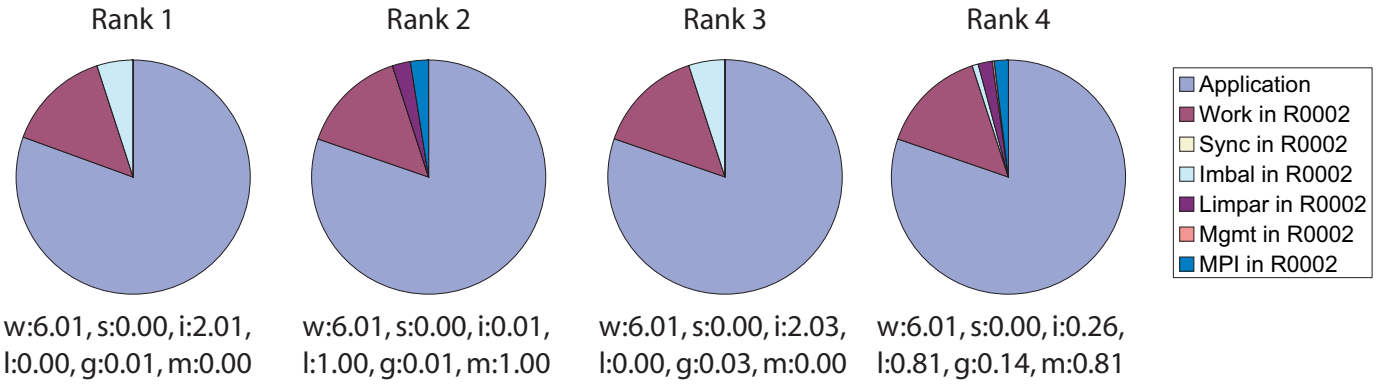

Fig. 9: Example Piechart for the overheads data.

are TAU [4] (tracing and profiling), Vampir [5] (trace visualization), KOJAK [10] (tracing), Scalasca [6] (tracing and profiling), mpiP [11] (profiling), and IPM [12] (profiling). Vendor supplied tools such as Cray PAT, Sun Studio and, Intel trace analyzer are available in addition to the academic offerings.

Some of the above tools also come with support for OpenMP performance measurement (either pure OpenMP or hybrid MPI+OpenMP). KOJAK, Scalasca, TAU, and Vampir, as well as `ompP` rely on source code instrumentation using Opari [13] to accomplish this. As vendor tools are specifically tailored to the vendor's compiler and OpenMP runtime implementation and generally don't have to support other platforms, they do not use source code instrumentation but rely on compiler instrumentation or profiling-enabled runtime libraries instead.

From an MPI perspective, almost all of the above tools give the user more detailed information such as about MPI callsites, sender-receiver pairs, and types of MPI calls. As mentioned earlier, our goal was not to reproduce the capabilities of existing tools, but to provide a mechanism of anchoring basic MPI-level data in the performance space of OpenMP

programs. Our approach is unique in this OpenMP-centric viewpoint, for example by extending ompP's existing overhead classification scheme to a new, messaging related overhead class. We envision it to be most useful for programmers that start with an OpenMP program and wish to add MPI-level parallelism. While this will be generally the more uncommon and challenging way to hybrid parallel programs, new mechanisms such as tasking can make the integration of message passing easier to handle while at the same time necessitating a detailed threads-based analysis of messaging behavior.

V. Conclusion and Future Work

We have presented extensions to an existing OpenMP profiling tool, ompP, for monitoring hybrid (OpenMP +MPI) applications. The focus for our tool remains on the OpenMP side, where we see MPI as a mechanism for an application comprised of several OpenMP parallel processes to communicate. OmpP allows the application developer to explore the MPI activity on a per-thread basis and correlate this with other threads-based data. OmpP provides for the ability to investigate the differences in OpenMP execution characteristics between processes, such as differences in execution time, hardware counters, or runtime overheads. OmpP accounts for the data transmitted by point to point and collective operations and for the time spent in MPI routines, it does not provide more detailed information such as statistics for each pair of senders and receivers.

Future and extended work is planned in multiple directions. First, as mentioned in Sec. II-B2 it is not possible to identify the *actual* amounts of performance data being sent or received at the MPI profiling layer and ompP gives the user the choice of displaying reasonably logic upper or lower bounds.

By using counter logic located on network interface cards such a correlation of the logical MPI behavior to the actual physical implementation can be accomplished. An extension to the PAPI [14], [15] library to handle non-CPU counter data sources is currently being developed under the name component PAPI (PAPI-C) [16]. Support for component PAPI can be readily included in ompP and this would allow for the immediate correlation in the reported profiling data.

Another area for future work will be the integration with existing powerful MPI profiling tools such as IPM and mpiP. The goal here is to correlate and display the data delivered by those tools in an integrated way that allows a developer to make most use of it. Data for this can come either from multiple runs, one for each tool being used, or, preferably, by using an approach such as $P^N MPI$ [17] that allows multiple MPI-level tools to be used simultaneously.

References

[1] S. L. Graham, P. B. Kessler, and M. K. McKusick, "gprof: A call graph execution profiler," *SIGPLAN Not.*, vol. 17, no. 6, pp. 120–126, 1982.

[2] J. Levon, "OProfile, A system-wide profiler for Linux systems," homepage: http://oprofile.sourceforge.net.

[3] M. Itzkowitz, O. Mazurov, N. Copty, and Y. Lin, "An OpenMP runtime API for profiling," accepted by the OpenMP ARB as an official ARB White Paper available online at http://www.compunity.org/futures/omp-api.html.

[4] S. S. Shende and A. D. Malony, "The TAU parallel performance system," *International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.

[5] H. Brunst and B. Mohr, "Performance analysis of large-scale OpenMP and hybrid MPI/OpenMP applications with VampirNG," in *Proceedings of the First International Workshop on OpenMP (IWOMP 2005)*, Eugene, Oregon, USA, May 2005.

[6] M. Geimer, F. Wolf, B. J. N. Wylie, and B. Mohr, "Scalable parallel trace-based performance analysis," in *Proceedings of the 13th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface (EuroPVM/MPI 2006)*, Bonn, Germany, 2006, pp. 303–312.

[7] J. S. Vetter, "Dynamic statistical profiling of communication activity in distributed applications," in *Proceedings of the 2002 ACM International Conference on Measurements and Modeling of Computer Systems (SIGMETRICS 2002)*. Marina Del Rey, California: ACM Press, 2002, pp. 240–250.

[8] P. N. Brown, R. D. Falgout, and J. E. Jones, "Semicoarsening multigrid on distributed memory machines," *SIAM J. Sci. Comput.*, vol. 21, no. 5, pp. 1823–1834, 2000.

[9] K. Fürlinger, O. Schenk, and M. Hagemann, "Task-queue based hybrid parallelism: A case study," pp. 624–632, 2004.

[10] F. Wolf and B. Mohr, "Automatic performance analysis of hybrid MPI/OpenMP applications." in *Proceedings of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2003)*. IEEE Computer Society, Feb. 2003, pp. 13–22.

[11] J. S. Vetter and A. Yoo, "An empirical performance evaluation of scalable scientific applications," in *Proceedings of the 2002 Conference on Supercomputing (SC 2002)*, Baltimore, Maryland, USA, 2002, pp. 1–18.

[12] D. Skinner, "Performance monitoring of parallel scientific applications," Lawrence Berkeley National Laboratory, Tech. Rep. LBNL-PUB-5503, May 1, 2005.

[13] B. Mohr, A. D. Malony, S. S. Shende, and F. Wolf, "Towards a performance tool interface for OpenMP: An approach based on directive rewriting," in *Proceedings of the Third Workshop on OpenMP (EWOMP'01)*, Sep. 2001.

[14] "PAPI web page: http://icl.cs.utk.edu/papi/."

[15] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. J. Mucci, "A portable programming interface for performance evaluation on modern processors," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 3, pp. 189–204, 2000.

[16] "Component papi documentation: http://icl.cs.utk.edu/projects/papi/files/documentation/PAPI-C.html."

[17] M. Schulz and B. R. de Supinski, "A flexible and dynamic infrastructure for MPI tool interoperability," in *Proceedings of the 2006 International Conference on Parallel Processing (ICPP-06)*, Washington, DC, USA, 2006, pp. 193–202.