The 41st International Conference on Parallel Processing

Tools for Power and Energy Analysis of Parallel Scientific Applications

Pedro Alonso[1], Rosa M. Badia[2], Jesús Labarta[2], María Barreda[3], Manuel F. Dolz[3], Rafael Mayo[3], Enrique S. Quintana-Ortí[3] and Ruymán Reyes[3]

1 Universitat Politècnica de València 2 Barcelona Supercomputing Center Centro Nacional de Supercomputación 3 Universitat Jaume I

September 10–13, 2012, Pittsburgh, PA

Motivation

- Understanding power usage in parallel workloads is crucial to develop the energy–aware software that will run in future Exascale systems.
- An integrated framework to **profile**, **monitor**, **model** and **analyze** power dissipation in parallel MPI and multi-threaded scientific applications.
- The framework includes
    - an own–designed device to measure internal DC power consumption,
    - a package offering a simple interface to interact with this design (`Extrae`+`Paraver`).
- The result: a useful environment to identify **sources of power inefficiency** directly in the source application code.
- In the case of **task–parallel** codes: statistical software module which inspects the execution trace to calculate the parameters of an accurate model for the global energy consumption.

Outline

1. Introduction: The integrated framework
2. Tools and APIs for Performance–Power Tracing
   - The `Extrae+Paraver` framework
   - The powermeter (`pm`) framework
3. Visualizing the Performance-Power Interaction
   - Example: LAPACK `dpotrf` routine
   - Example: LAPACK LUPP routine
   - Example: ScaLAPACK `pdpotrf` routine
4. Analysis of Task–Parallel Applications
   - Using SMPSs to parallelize the Cholesky factorization
   - An energy/power model for task-parallel applications
5. Conclusions

Introduction: The integrated framework

- An internal DC powermeter microcontroller-based design: samples the nodal power dissipated by the system mainboard. Rates: 25 to 100 Hz.
- A simple API to interact with a number of power measurement devices: commercial external AC meters like WattsUp? Pro .Net and our own internal powermeters.
- The associated `pm` library and drivers which allow to capture the power dissipated during the execution of an application in a separate system.
- Integration with the `Extrae+Paraver` packages which allows interactive analysis of a graphical trace relating the power dissipation per node/core and information per core activity.
- **Task–parallel** applications: a statistical analysis module and an energy model which utilizes the information contained in the power–performance traces to correlate the average power–energy of each task type and the power dissipation per core.

Introduction: The integrated framework

- An internal DC powermeter microcontroller-based design: samples the nodal power dissipated by the system mainboard. Rates: 25 to 100 Hz.
- A simple API to interact with a number of power measurement devices: commercial external AC meters like WattsUp? Pro .Net and our own internal powermeters.
- The associated `pm` library and drivers which allow to capture the power dissipated during the execution of an application in a separate system.
- Integration with the `Extrae+Paraver` packages which allows interactive analysis of a graphical trace relating the power dissipation per node/core and information per core activity.
- **Task–parallel** applications: a statistical analysis module and an energy model which utilizes the information contained in the power–performance traces to correlate the average power–energy of each task type and the power dissipation per core.

Introduction: The integrated framework

- An internal DC powermeter microcontroller-based design: samples the nodal power dissipated by the system mainboard. Rates: 25 to 100 Hz.
- A simple API to interact with a number of power measurement devices: commercial external AC meters like WattsUp? Pro .Net and our own internal powermeters.
- The associated `pm` library and drivers which allow to capture the power dissipated during the execution of an application in a separate system.
- Integration with the `Extrae+Paraver` packages which allows interactive analysis of a graphical trace relating the power dissipation per node/core and information per core activity.
- **Task–parallel** applications: a statistical analysis module and an energy model which utilizes the information contained in the power–performance traces to correlate the average power–energy of each task type and the power dissipation per core.

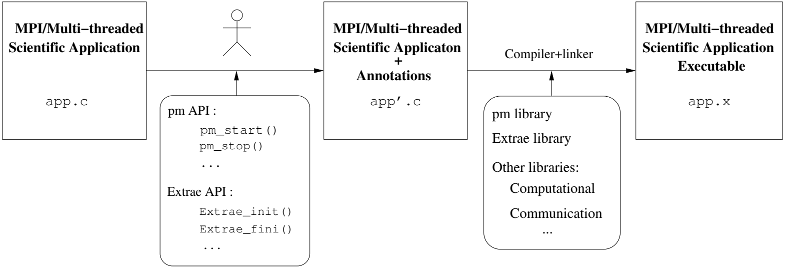
Introduction: The integrated framework

- An internal DC powermeter microcontroller-based design: samples the nodal power dissipated by the system mainboard. Rates: 25 to 100 Hz.
- A simple API to interact with a number of power measurement devices: commercial external AC meters like WattsUp? Pro .Net and our own internal powermeters.
- The associated `pm` library and drivers which allow to capture the power dissipated during the execution of an application in a separate system.
- Integration with the `Extrae+Paraver` packages which allows interactive analysis of a graphical trace relating the power dissipation per node/core and information per core activity.
- **Task–parallel** applications: a statistical analysis module and an energy model which utilizes the information contained in the power–performance traces to correlate the average power–energy of each task type and the power dissipation per core.

Introduction: The integrated framework

- An internal DC powermeter microcontroller-based design: samples the nodal power dissipated by the system mainboard. Rates: 25 to 100 Hz.
- A simple API to interact with a number of power measurement devices: commercial external AC meters like WattsUp? Pro .Net and our own internal powermeters.
- The associated `pm` library and drivers which allow to capture the power dissipated during the execution of an application in a separate system.
- Integration with the `Extrae`+`Paraver` packages which allows interactive analysis of a graphical trace relating the power dissipation per node/core and information per core activity.
- **Task–parallel** applications: a statistical analysis module and an energy model which utilizes the information contained in the power–performance traces to correlate the average power–energy of each task type and the power dissipation per core.

Tools for performance and power tracing



Extrae and Paraver summary

Extrae: instrumentation and measurement package of BSC (Barcelona Supercomputing Center):

- Intercept calls to MPI, OpenMP, PThreads
- Records relevant information: time stamped events, hardware counter values, etc.
- Dumps all information into a single trace file.

Paraver: graphical interface tool from BSC to analyze/visualize trace files:

- Inspection of parallelism and scalability
- High number of metrics to characterize the program and performance application

Instrumented code with Extrae

Guiding example: *Cholesky factorization*. $A = U^T U$, U is upper triangular.

```
#define A_ref(i,j) A[((j)-1)*Alda+((i)-1)]
void dpotrf( int n, int nb, double *A, int Alda, int *info ){
  // Declaration of variables ...
  pm_start_counter(&pm_ctr);
  Extrae_init();
  for (k=1; k<=n; k+=nb) {
    // Factor current diagonal block
    Extrae_event(500000001,1);
    dpotf2( nb, &A_ref(k,k), Alda, info );
    Extrae_event(500000001,0);

    if( k+nb <= n ) {
      // Triangular solve
      Extrae_event(500000001,2);
      dtrsm( "L", "U", "T", "N", nb, n-k-nb+1, &done, &A_ref( k, k ), Alda,
             &A_ref( k, k+nb ), Alda );
      Extrae_event(500000001,0);

    }
    // ... More code ...
  }
  Extrae_fini();
  pm_stop_counter(&pm_ctr);
}
```

Extrae API

`void Extrae_init (void)`

Purpose:	Initializes the tracing library.

`void Extrae_fini (void)`

Purpose:	Finalizes the tracing library and dumps the intermediate tracing buffers onto disk.

`void Extrae_event (unsigned event, unsigned value)`

Purpose:	Adds a single time-stamped event to the tracefile.
`event`:	Identify the event.
`value`:	Identify the event. `value`=0 marks the end of the event.

`void Extrae_user_function (int enter)`

Purpose:	Emits an event into the tracefile which references the source code.
`enter`:	Identify the event. `enter`=0 marks the end of the event.

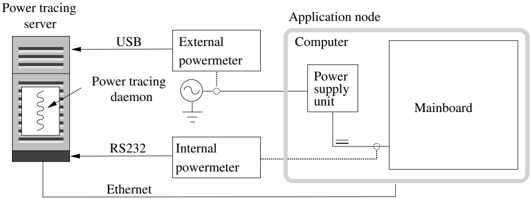
`void Extrae_counters (void)`

Purpose:	Emits the value of the active hardware counter set.

The powermeter (pm) framework

pmlib library

- Power measurement package of Universitat Jaume I (Spain)
- Interface to interact and utilize our own and commercial power meters



- **Server daemon**: collects data from power meters and send to clients
- **Client library**: enables communication with server and synchronizes with `start-stop` primitives

Power meter:

- LEM HXS 20-NP transducers with a microcontroller (own design)
- Sampling rate 25 Hz

The pmAPI

Some client routines:

`int pm_set_server( char *svrip, int port, server_t *svr )`
Initializes IP and port for server connection.

`int pm_create_counter( char *devn, mask_t lin, int aggr, int freq, server_t svr, counter_t *pm_ctr)`
Creates a new counter.

`int pm_start_counter( counter_t *pm_ctr )`
Starts the measurements.

`int pm_continue_counter( counter_t *pm_ctr )`
Continues the measurements.

`int pm_stop_counter( counter_t *pm_ctr )`
Stops the measurements.

`int pm_get_counter_data( counter_t *pm_ctr )`
Dumps power data onto memory.

`int pm_print_data_stdout( counter_t *pm_ctr )`
Imprime los datos por la salida estándar.

`int pm_print_data_paraver( char *file, counter_t *pm_ctr, char *unit )`
Dumps power data into a Paraver compatible file.

`int pm_finalize_counter( counter_t *pm_ctr )`
Finalizes the counter.

Instrumented code with Extrae

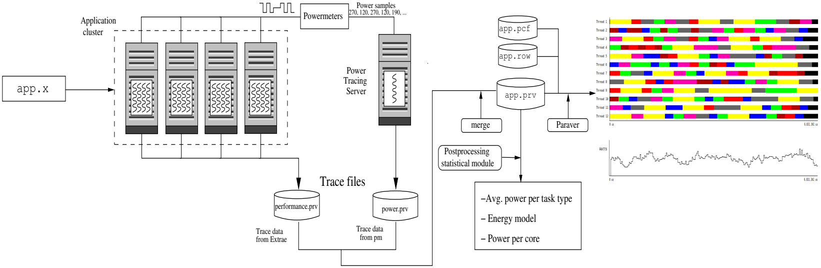
Guiding example: *Cholesky factorization*. $A = U^T U$, U is upper triangular.

```
#define A_ref(i,j) A[((j)-1)*Alda+((i)-1)]
void dpotrf( int n, int nb, double *A, int Alda, int *info ){
  // Declaration of variables ...
  pm_start_counter(&pm_ctr);
  Extrae_init();
  for (k=1; k<=n; k+=nb) {
    // Factor current diagonal block
    Extrae_event(500000001,1);
    dpotf2( nb, &A_ref(k,k), Alda, info );
    Extrae_event(500000001,0);

    if( k+nb <= n ) {
      // Triangular solve
      Extrae_event(500000001,2);
      dtrsm( "L", "U", "T", "N", nb, n-k-nb+1, &done, &A_ref( k, k ), Alda,
             &A_ref( k, k+nb ), Alda );
      Extrae_event(500000001,0);

    }
    // ... More code ...
  }
  Extrae_fini();
  pm_stop_counter(&pm_ctr);
}
```

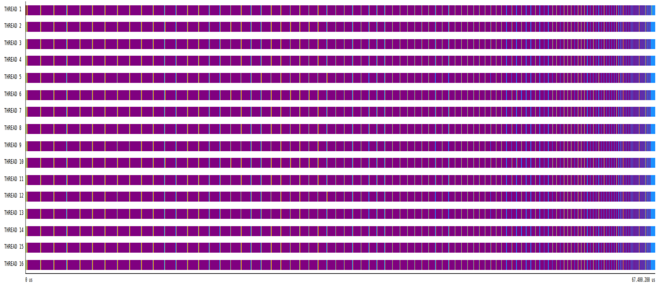
Collecting traces and visualization



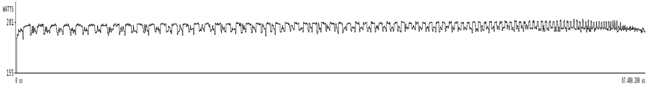
Example: LAPACK `dpotrf` routine

Platform: 16 AMD Opteron 6128 cores (two eight-core sockets, 2.0 GHz) and 48 GB of RAM.
Problem: Cholesky `n=16384`, `nb=256`.

Kernels



Power



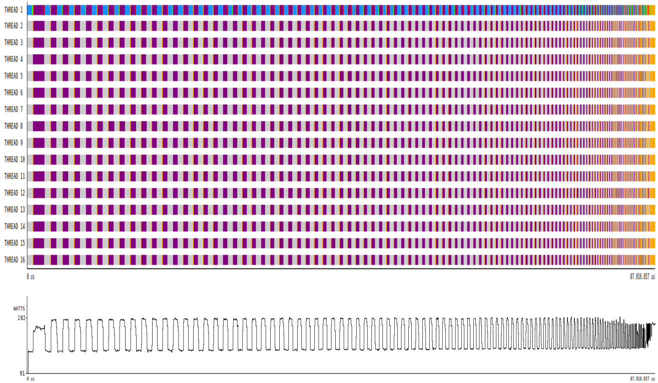
Legend: `idle` `dpotf2` `dtrsm` `dsyrk` `sync`

Example: LAPACK LUPP routine

Problem: LUPP `n=16384`, `nb=256`.

Kernels

Power



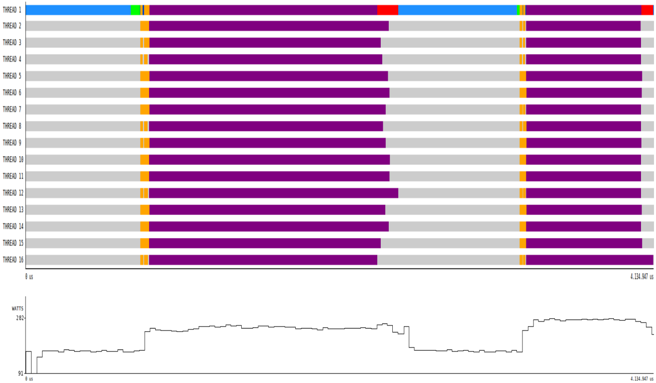
Legend: idle, dgetf2, dlaswp, dtrsm, dgemm, sync

Collecting traces and visualization

Problem: LUPP n=16384, nb=256.

Kernels

Power



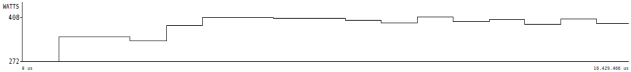
Legend idle dgetf2 dlaswp dtrsm dgemm sync

Example: ScaLAPACK `pdpotrf` routine

Platform: 4 nodes with two Intel Xeon Quad-core E5520 processors (2.27 GHz) and 24 GB.
Problem: ScaLAPACK `pdpotrf n=20000`.

Kernels

Power



Legend

idle dpotrf dtrsm dsyrk dgemm

Introduction

- We develop an energy model for task-parallel scientific applications that can be subsequently leveraged by task schedulers to meet power budget and thermal constraints while simultaneously improving performance.
- We first describe how to obtain a task-parallel implementation/execution of the Cholesky factorization using SMPSs to then formulate and validate our model for parallel applications of this class.

SMPSs

- SMPSs is an instance of the StarSs framework for shared-memory multiprocessors.
- It combines a language with a reduced number of OpenMP-like pragmas, a source-to-source compiler, and a runtime system to leverage task-level parallelism in sequential codes:
    1. The programmer employs pragmas to annotate routines indicating directionality of operands with clauses.
    2. A source-to-source compiler produces a C code.
    3. A runtime decomposes the code into a number tasks at run time, dynamically identifying dependencies, and issuing ready tasks.

SMPSs: Example code for Cholesky factorization

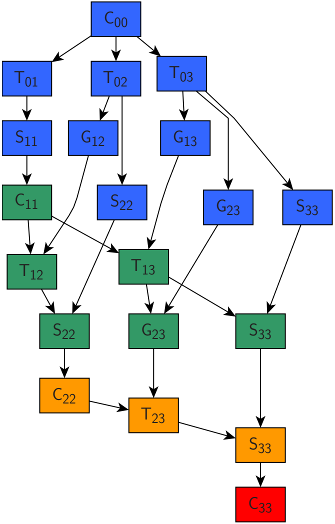
```
#define A_ref(i,j) A[((j)-1)*Alda+((i)-1)]
void dpotrf_smpss( int n, int nb, double *A, int Alda, int *info ){
  // Declaration of variables ...
  for (k=1; k<=n; k+=nb) {
    // Factor current diagonal block
    dpotf2_wrapper( nb, &A_ref(k,k), Alda, info );
    if( k+nb <= n ) {
      // Triangular solve
      for (j=k+nb; k<=n; k+=nb)
        dtrsm_wrapper( nb, &A_ref( k, k ), &A_ref( k, j ), Alda );
      // Update trailing submatrix
      for (i=k+nb; i<=n; i+=nb) {
        dsyrk_wrapper( nb, &A_ref( k, i ), &A_ref( i, i ), Alda );
        for (j=i+nb; j<=n; j+=nb)
          dgemm_wrapper( nb, &A_ref( k, i ), &A_ref( k, j ), &A_ref( i, j ), Alda );
    } }
} }

#pragma css task input( nb, ldm ) inout( A[nb*nb], info )
void dpotf2_wrapper( int nb, double A[], int ldm, int *info ) {
  dpotrf( "U", &nb, A, &ldm, info );
}

#pragma css task input( nb, A[nb*nb], ldm ) inout( B[nb*nb] )
void dtrsm_wrapper( int nb, double A[], double B[], int ldm ) {
  dtrsm( "L", "U", "T", "N", &nb, &nb, &done, A, &ldm, B, &ldm );
}

#pragma css task input( nb, A[nb*nb], ldm ) inout( C[nb*nb] )
. . .
```

SMPSs: Example code for Cholesky factorization



An energy/power model for task-parallel applications

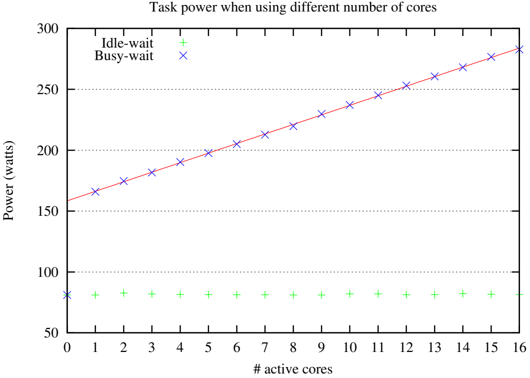
Consider a task-parallel application that can be decomposed into a number of tasks of r different types, where tasks of the j-th type, $j = 1, 2, ..., r$, run during a total time T_j, we propose the following energy consumption model:

$$E_{mod} = P^Y \cdot T_{idle} + (P^Y + P^S) \cdot T_{busy} + \sum_{j=1}^{r} \bar{P}_j^D \cdot T_j,$$

where

- $P^{(S)Y(stem)}$ is the power dissipated by the target platform when idle,
- $P^{S(tatic)}$ is the power required to "energize" the platform,
- T_{tot} is the total execution time of the application,
- T_{idle} is the time during which there are no tasks running,
- $T_{busy} = T_{tot} - T_{idle}$, and
- $\bar{P}_j^{D(ynamic)}$ is the average dynamic power dissipated by one core when running a task of type j.

SMPSs: Example code for Cholesky factorization



- $P^Y = 80.15$ Watts
- $P(c) = \alpha + \beta \cdot c = 158.40 + 7.84 \cdot c$
- $P^S = \alpha - P^Y = 158.40 - 80.15 = 78.25$ Watts

An energy/power model for task-parallel applications

- We evaluate the average power per task type, $\bar{P}_j^D$.
- For this purpose, we propose to employ information from Extrae and apply a postprocessing statistical module to estimate these values.
- A power trace is composed of m samples $s = (s_1, s_2, \ldots, s_m)$, the measured energy consumption derived from these samples is given by

$$E_{mes} = \sum_{i=1}^{m} \frac{s_i}{m} \cdot T_{tot} = \bar{P} \cdot T_{tot},$$

- We also filter the performance trace to obtain a sequence of m tuples of the form $(a_{i,1}, a_{i,2}, \ldots, a_{i,r})$, $i = 1, 2, \ldots, m$, where $a_{i,j}$ is the number of tasks of type j being executed at the instant of time corresponding to the i-th power sample.

An energy/power model for task-parallel applications

- We evaluate the average power per task type, $\bar{P}_j^D$.
- For this purpose, we propose to employ information from Extrae and apply a postprocessing statistical module to estimate these values.
- A power trace is composed of m samples $s = (s_1, s_2, \ldots, s_m)$, the measured energy consumption derived from these samples is given by

$$E_{mes} = \sum_{i=1}^{m} \frac{s_i}{m} \cdot T_{tot} = \bar{P} \cdot T_{tot},$$

- We also filter the performance trace to obtain a sequence of m tuples of the form $(a_{i,1}, a_{i,2}, \ldots, a_{i,r})$, $i = 1, 2, \ldots, m$, where $a_{i,j}$ is the number of tasks of type j being executed at the instant of time corresponding to the i-th power sample.

$$A = \begin{pmatrix} 1 & 2 & 0 & 0 \\ 0 & 2 & 3 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 2 \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

An energy/power model for task-parallel applications

- We evaluate the average power per task type, $\bar{P}_j^D$.
- For this purpose, we propose to employ information from Extrae and apply a postprocessing statistical module to estimate these values.
- A power trace is composed of m samples $s = (s_1, s_2, \ldots, s_m)$, the measured energy consumption derived from these samples is given by

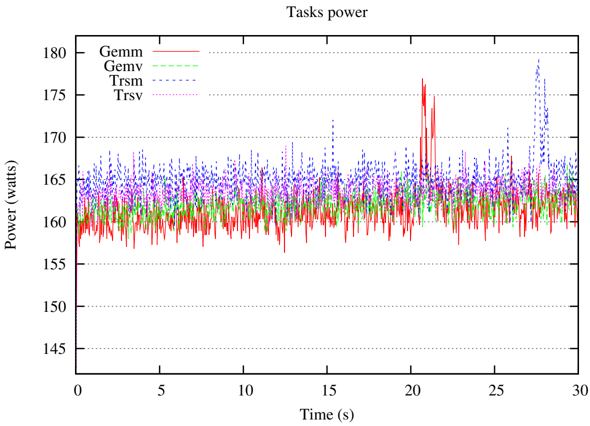
$$E_{mes} = \sum_{i=1}^{m} \frac{s_i}{m} \cdot T_{tot} = \bar{P} \cdot T_{tot},$$

- We also filter the performance trace to obtain a sequence of m tuples of the form $(a_{i,1}, a_{i,2}, \ldots, a_{i,r})$, $i = 1, 2, \ldots, m$, where $a_{i,j}$ is the number of tasks of type j being executed at the instant of time corresponding to the i-th power sample.
- We obtain a linear system $Ax = b$, where the rows of the $m \times r$ matrix A correspond to the m tuples that specify the task activity, and the entries of b satisfy $b_i = s_i - (P^Y + P^S)$.
- The entries of the solution vector, $x = (x_1, x_2, \ldots, x_r)$, for the linear-least squares problem

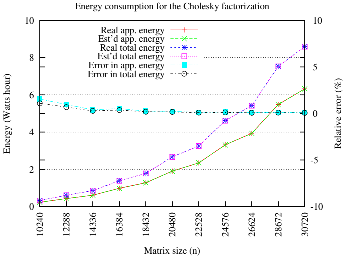
$$\min_x \|Ax - b\|_2,$$

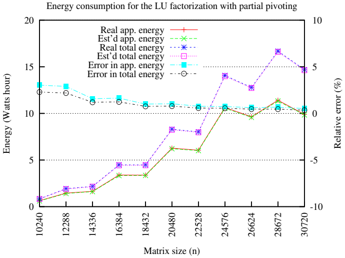
then provide an estimation of the average power per task type; i.e., $x_j \approx \bar{P}_j^D$.

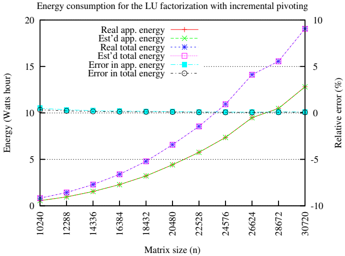
An energy/power model for task-parallel applications

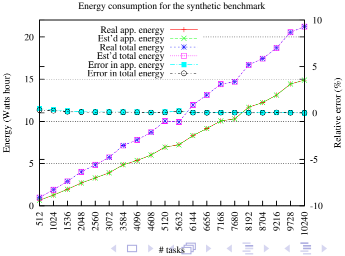


An energy/power model for task-parallel applications









An energy/power model for task-parallel applications

In order to validate the model, we obtain the following data:

- Estimated and real (measured) total energy consumption of the mainboard, as given respectively by

$$E_{mod} = P^Y \cdot T_{idle} + (P^Y + P^S) \cdot T_{busy} + \sum_{j=1}^{r} \bar{P}_j^D \cdot T_j,$$

  and

$$E_{mes} = \sum_{i=1}^{m} \frac{s_i}{m} \cdot T_{tot} = \bar{P} \cdot T_{tot}.$$

- Estimated and real *application* energy consumption, given by $E_{mod}^D = \sum_{j=1}^{r} \bar{P}_j^D \cdot T_j$ and $E_{mes}^D = \sum_{j=1}^{m} \frac{b_i}{m} \cdot T_{tot} = \bar{P}^D \cdot T_{tot}$, respectively.
- Relative errors of the energy estimations, computed as the difference between the estimated and real (total or application) energy consumption divided by the real (total or application) energy consumption: $\frac{|E_{mes} - E_{mod}|}{E_{mes}}$ and $\frac{|E_{mes}^D - E_{mod}^D|}{E_{mes}^D}$.

Conclusions and future work

- We have presented a framework to analyze the power-performance interaction of parallel MPI and/or multi-threaded scientific applications.
- Using a number of representative operations from dense linear algebra, we have illustrated how the hardware and software tools in this framework provide a detailed execution trace of the application that allows the determination of the power cost at the granularity of functions or code fragments.
- For task-parallel applications we have extended these results to derive an energy model which, utilizing the information contained in the power-performance traces, obtains the power cost at the task level as well as the power dissipated per core.
- While we expect that this environment can already be leveraged to write more energy-aware applications, as part of future work, we aim at integrating this information into the SMPSs runtime system, to obtain a power/energy-aware scheduler.

Thanks for your attention!

Questions?