IBM ExaBounds manual

R. Jongerius
IBM Research, The Netherlands

January 29, 2018

Abstract

This manual describes how to use IBM ExaBounds for performance and power modeling. ExaBounds is implemented in Mathematica and requires a Mathematica license to be used. There is a GUI available to use IBM ExaBounds, but it is also possible to use the Mathematica language to create scripts for automation of design-space exploration experiments. The current GUI can be used for the processor performance models. For the network performance models, IBM ExaBounds provides the user with an API that can be used to create Mathematica scripts.

Contents

1 Introduction

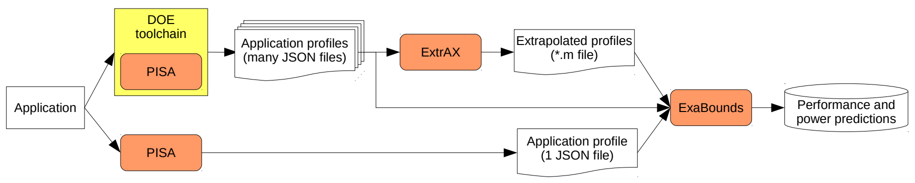
This document describes how to use IBM ExaBounds, which implements the performance model described in [1]. Section 2 describes how to set up the environment correctly. Section 5 discusses how to use the graphical user interface of IBM ExaBounds. Section 6 describes how to use the API of the notebook to create your own automated design-space exploration notebooks.

2 Setup

IBM ExaBounds consists of an executable notebook (ExaBoundsLite.nb) and a large set of supporting packages. It can only be executed with the full version of Wolfram Mathematica (version 11.2 and later).

2.1 McPAT

In order to predict the power consumption of compute nodes, McPAT [2] has to be available on the system. McPAT is used to predict power consumption of the processor cores and caches. ExaBounds expects the McPAT executable named MCPAT (or MCPAT.EXE for Windows) to be available in a directory listed in the $PATH environment variable. ExaBounds is tested with McPAT 1.0, it can be downloaded from `http://www.hpl.hp.com/research/mcpat/`.



Application
DOE toolchain
PISA
PISA
Application profiles (many JSON files)
ExtrAX
Extrapolated profiles (*.m file)
Application profile (1 JSON file)
ExaBounds
Performance and power predictions

Figure 1: Different workflows of our tool flow. IBM Platform-Independent Software Analysis is used to extract profiles of applications. Such profiles can be extrapolated to exascale using IBM Exascale Extrapolator. ExaBounds predicts the performance and power consumption of the application running on different architectures.

2.2 CACTI

The power consumption of the main memory can be predicted using either the build-in power model IBM Memory-Scheduler-Agnostic Power Model [3] or CACTI [4]. In case CACTI is desired, it needs to be available via the $PATH environment variable as well. ExaBounds only works with CACTI 5.3, which can be downloaded from `http://www.hpl.hp.com/research/cacti/`. The executable name has to be CACTI53 (or CACTI53.EXE for Windows).

The memory power model can be selected in the CDF or notebook and is default set to IBM Memory-Scheduler-Agnostic Power Model.

3 Workflows

ExaBounds is part of a larger set of tools that together can be used for performance and power analysis of future (exascale) computing systems. The set of tools encompasses IBM Platform-Independent Software Analysis [5] for hardware-independent application profiling. The generates profiles serve as input for the workload extrapolation models of IBM Exascale Extrapolator [6] or directly to IBM ExaBounds.

Figure 1 shows the potential workflows of our tooling. The lower part of the figure shows the most basic workflow: IBM Platform-Independent Software Analysis is used to profile a single run of an application. The generated profile (in JSON format) is loaded in IBM ExaBounds and predictions are made for an architecture of for a set of architectures (design-space exploration).

The top part of the figure shows a more complex workflow. Applications can behave differently for different workloads (both workload size or data content), or, for parallel applications, threads within an application may behave differently for different amounts of exploited parallelism. To capture this dynamism, a design of experiments (DOE) can be performed to sample the behavior of the application at different operating points. This will involve multiple runs of the profiler and will result in a set of JSON output files. These output files can subsequently be directly used in ExaBounds for performance and power predictions, but it is also possible to first use IBM Exascale Extrapolator to extrapolate the workload profiles to a point that isn't sampled (using machine learning). As an example, the application profiles can be exploited to an exascale-sized workload, a workload size that cannot be analyzed by IBM Platform-Independent Software

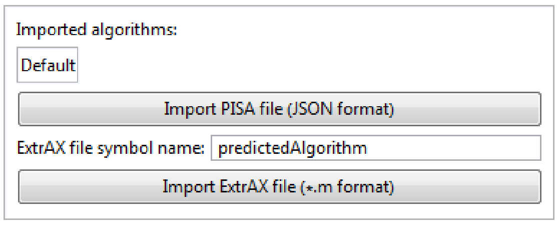


Figure 2: Import algorithms GUI element.

Analysis directly due to constraints on the system which is used for profiling. IBM Exascale Extrapolator generates a Mathematica file containing the profiles of the application at the targed (scaled) operating points. This Mathematica file can be similarly read by IBM ExaBounds for predictions.

4 Default architectures and algorithms

! IBM ExaBounds ships with a set of default loaded architectures and algorithms. The architectural properties aim to reflect their respective architectures as good as possible. Most properties are retrieved from public sources on the Internet. Some of the remaining properties cannot be found and are based on reasonable estimates. It is advised to carefully check default architectures and algorithms for correctness for real experiments.

5 Graphical user inteface

The following section describes the usage of the IBM ExaBounds graphical user interface.

5.1 Loading application profiles

IBM ExaBounds can load application profiles in the JSON format generated by IBM Platform-Independent Software Analysis [5] or in the Mathematica format generated by IBM Exascale Extrapolator [6]. This can be used to use ExaBounds to analyze any application that can be analyzed using IBM Platform-Independent Software Analysis. Figure 2 shows the GUI element that can be used to load algorithms. It shows the algorithms already loaded. The DEFAULT algorithm is loaded by default and contains two profiles of the Graph 500 [7] benchmark set. By clicking IMPORT JSON FILE a dialog box opens to load new IBM Platform-Independent Software Analysis files in JSON format. Similarly, clicking IMPORT MATHEMATICA FILE allows loading an IBM Exascale Extrapolator file in Mathematica format *.m (by default, IBM Exascale Extrapolator stores data in the PREDICTEDALGORITHM symbol).

During loading of an input file, several warning messages can be displayed at the bottom of the CDF file. Two types of warning messages can exist: missing properties and invalid data. In case of invalid data, there is a problem with the

Table 1: Properties and their use in the analytic models.

Property	**Use**	**Always needed?**
instructionMix	Core modeling	Yes
dataReuseDistribution[1]	Core modeling	Yes
instrReuseDistribution	Not used	No
memoryFootprint	Not used	No
branchEntropy	Not used	No
bestMisprectionRate	Core modeling	No
ilp	Core modeling	Yes
ilptype	Core modeling	Yes
inorder ilp[2]	In-order core modeling	No
inorder ilptype[2]	In-order core modeling	No
registerAccesses	Core modeling	Yes
openMPinstructionMix	Not used	No
mpiInstructionMix	Not used	No
mpiDataExchanged	Not used	No
mpiCommunicationVector	Not used	No
externalLibraryCallCount	Not used	No
processId	Not used	No
threadId	Not used	No

[1] The data reuse distribution for two cache line sizes (64 and 128) is being read, only one is needed. However, a warning will be generated when only one is available.

[2] Will be deprecated. In-order modeling will be done using the regular ILP parameters.

data in the input file. For example, the summation of all instruction fractions does not add up to 1.

An example missing properties warning is:

AlgorithmFromFileJSON‘LoadAlgorithmJSON::missingproperties : The properties {externalLibraryCallCount, instrReuseDistribution, memoryFootprint, mpiCommunicationVector, mpiDataExchanged, mpiInstructionMix, openMPinstructionMix} were not read correctly for the file: file.out.

The warning states that several properties are missing in the profile file and therefor not loaded. Care should be taken that all properties that are relevant for a specific analysis are loaded for valid performance and power analysis results. Table 1 list the common properties[1] and for which parts of the models they are used. Note that all properties related to core modeling should always be loaded.

5.2 Loading architectures

Architectures are also described using JSON files. In total, three different types of JSON files can be loaded: processor architecture files, memory architecture files and network architecture files. Each file contains a set of parameters that describe an architecture, the parameters are described in Appendix B. During execution of the models, an processor, memory, and network architecture are combined to describe a complete system.

[1]These properties usually cover several parameters in the profile files.

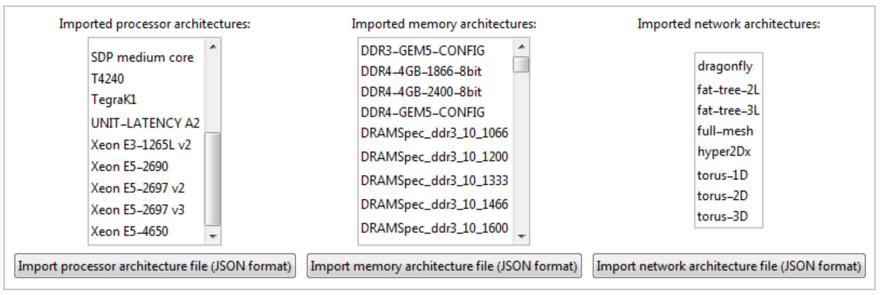
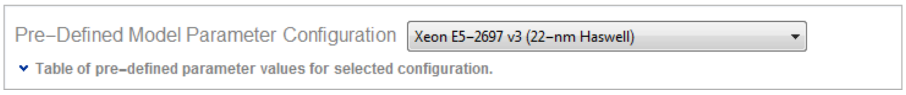


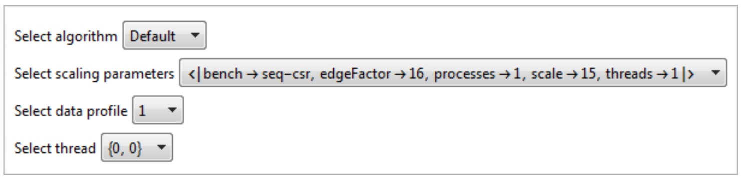
Figure 3: Import architectures GUI element.



Pre-Defined Model Parameter Configuration: Xeon E5-2697 v3 (22-nm Haswell)

Table of pre-defined parameter values for selected configuration.

(a) Architecture picker



Select algorithm: Default

Select scaling parameters: <|bench → seq-csr, edgeFactor → 16, processes → 1, scale → 15, threads → 1|>

Select data profile: 1

Select thread: {0, 0}

(b) Application picker

Figure 4: Selection boxes to select an architecture and algorithm for a basic performance and power analysis.

IBM ExaBounds loads a set of default architectures that can be used. Custom architectures can be loaded using the architecture interface as shown in Figure 3.

5.3 Performing a basic performance and power analysis for a single thread single-node

5.3.1 Choosing the architecture

The first step to do a basic performance and power analysis is to select the architecture to study. Figure 4a shows the architecture selection box. The drop-down menu allows to select any of the predefined machine configurations. By clicking the small downward arrow, a table expands which shows the architectural parameters of the choosen configuration.

5.3.2 Choosing the application

After selection of the architecture, an application can be selected to analyze. Figure 4b shows the four selection boxes needed to select a thread from an application. From top to bottom, the four boxes select: the algorithm, an associated set of scaling parameters (e.g., the data size), a data profile (e.g., different runs with different data), and finally a specific thread.

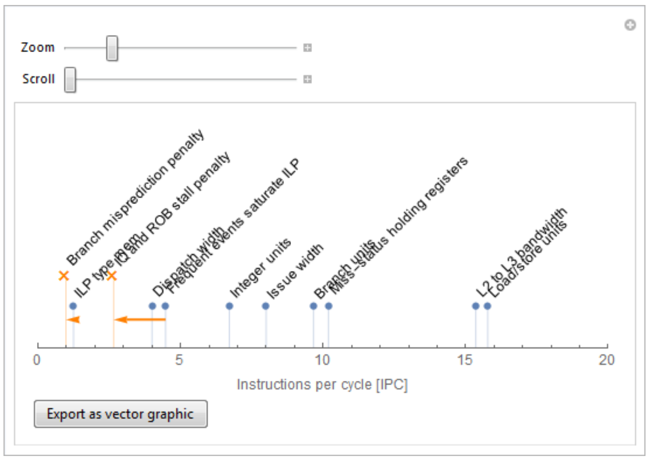
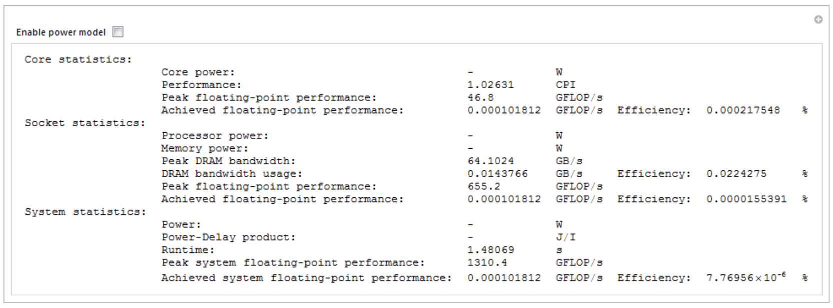


Figure 5: Bottleneck plot for the selected architecture and application pair.



```
Core statistics:
                  Core power:                                  -            W
                  Performance:                                 1.02631      CPI
                  Peak floating-point performance:             46.8         GFLOP/s
                  Achieved floating-point performance:         0.000101812  GFLOP/s  Efficiency:  0.000217548    %
Socket statistics:
                  Processor power:                             -            W
                  Memory power:                                -            W
                  Peak DRAM bandwidth:                         64.1024      GB/s
                  DRAM bandwidth usage:                        0.0143766    GB/s     Efficiency:  0.0224275      %
                  Peak floating-point performance:             655.2        GFLOP/s
                  Achieved floating-point performance:         0.000101812  GFLOP/s  Efficiency:  0.0000155391   %
System statistics:
                  Power:                                       -            W
                  Power-Delay product:                         -            J/I
                  Runtime:                                     1.48069      s
                  Peak system floating-point performance:      1310.4       GFLOP/s
                  Achieved system floating-point performance:  0.000101812  GFLOP/s  Efficiency:  7.76956×10^-6  %
```

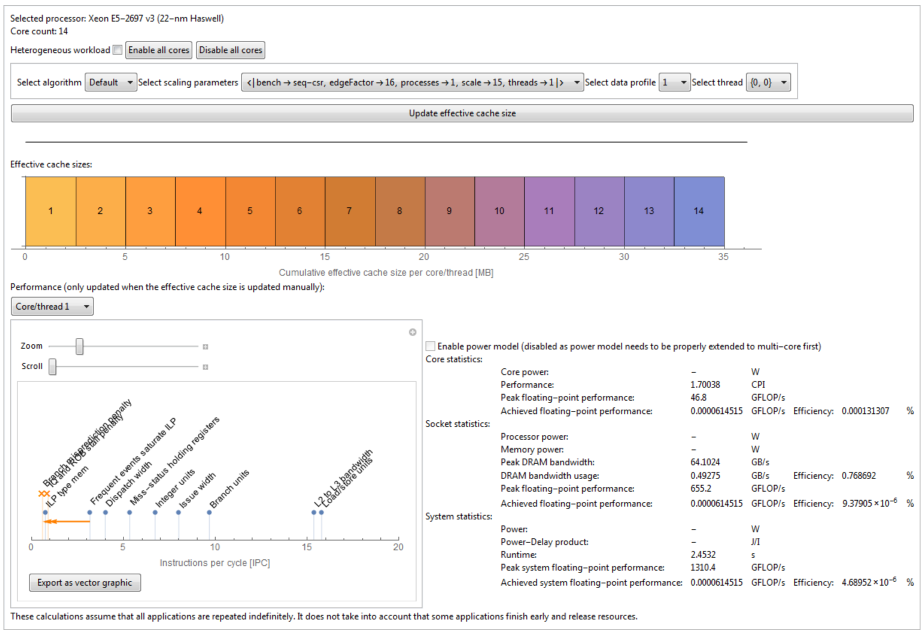
Figure 6: Results table for the selected architecture and application pair.

5.3.3 Results

After selection, the results update automatically and is presented in two forms: the bottleneck plot and a table.

Figure 5 shows the bottleneck plot of the selected combination of architecture and application. The plot gives insight in the architectural and application parameters that limit the performance of the thread. The blue bullets show absolute constraints, while the orange crosses show relative constraint. The relative constraints are always relative of an absolute constraint: shifting the absolute constraints will also shift the relative constraints. The absolute constraints are independent of each other. In the example, the absolute constraint limiting performance is the ILP per type mem, which indicates that the limited parallelism in the memory access or their long latency is expected to limit performance on a real system. The branch misprediction penalty decreases performance further.

The results in table-form are shown in Figure 6. By default, the power analysis using McPAT, and IBM Memory-Scheduler-Agnostic Power Model or CACTI is disabled. By checking the checkbox, the power analysis can be enabled and will run automatically. Please note that this can take some time to finish during which it may appear that the system hangs.



Selected processor: Xeon E5−2697 v3 (22−nm Haswell)
Core count: 14

Heterogeneous workload ☐ Enable all cores Disable all cores

Select algorithm Default Select scaling parameters <|bench → seq−csr, edgeFactor → 16, processes → 1, scale → 15, threads → 1|> Select data profile 1 Select thread {0, 0}

Update effective cache size

Effective cache sizes:

Performance (only updated when the effective cache size is updated manually):

Core/thread 1

☐ Enable power model (disabled as power model needs to be properly extended to multi−core first)

Core statistics:						
	Core power:	–	W			
	Performance:	1.70038	CPI			
	Peak floating−point performance:	46.8	GFLOP/s			
	Achieved floating−point performance:	0.0000614515	GFLOP/s	Efficiency:	0.000131307	%
Socket statistics:						
	Processor power:	–	W			
	Memory power:	–	W			
	Peak DRAM bandwidth:	64.1024	GB/s			
	DRAM bandwidth usage:	0.49275	GB/s	Efficiency:	0.768692	%
	Peak floating−point performance:	655.2	GFLOP/s			
	Achieved floating−point performance:	0.0000614515	GFLOP/s	Efficiency:	9.37905×10^{-6}	%
System statistics:						
	Power:	–	W			
	Power−Delay product:	–	J/I			
	Runtime:	2.4532	s			
	Peak system floating−point performance:	1310.4	GFLOP/s			
	Achieved system floating−point performance:	0.0000614515	GFLOP/s	Efficiency:	4.68952×10^{-6}	%

These calculations assume that all applications are repeated indefinitely. It does not take into account that some applications finish early and release resources.

Figure 7: Multi-core analysis GUI and results.

5.4 Multi-core analysis

A multi-core analysis can be performed using the multi-core GUI interface as is shown in Figure 7. It is a known issue that this interface is sometimes slow to respond.

The first step is to select an architecture with the architecture selection box as shown in Figure 4a. Then, the required threads can be selected to run on the target platform. The GUI gives the choice to run a homogeneous workload (all threads the same) or a heterogeneous workload (all threads different). After specifying the right threads, click the "Update results" button to recalculate the effective cache size for each thread and update the results. The results will be shown with a similar bottleneck plot and table as shown before.

5.5 Design-space exploration

T.b.d.

5.6 Troubleshooting

It can happen that external tools like McPAT fail or that for other reasons IBM ExaBounds returns wrong answers due to some error. Usually, re-evaluating the model does not help as IBM ExaBounds caches intermediate results to speed up evaluation. The three buttons at the bottom of the notebook can be used to clear all caches. Notice that this means that all analysis will be performed again.

6 Programming with the IBM ExaBounds API

IBM ExaBounds can be used with custom notebooks by programming in Mathematica while using the IBM ExaBounds API. This section describes how to the use API.

6.1 Starting point

As the starting point, one should always load the ExaBoundsLite.nb notebook and execute it contents. This will make sure all the packages are loaded and the data structures are initialized.

6.2 Loading algorithms

Algorithms can be loaded automatically without using the GUI. First, create a list of IBM Platform-Independent Software Analysis profile files (full path and filename) to load:

```
In[1]:= dir = "/home/user/";
In[2]:= JSONfileList = {
          dir <> "app1.out",
          dir <> "app2.out",
          dir <> "app3.out"
          };
```

The profile files in the list can now be loaded using AddAlgorithm:

```
In[3]:= algorithmIdentifiers =
          AddAlgorithm[ExaBoundsAlgInfo, #] & /@ JSONfileList;
```

The call to AddAlgorithm[] for each file returns a list of algorithmIdentifiers, each corresponding to the first data profile and first thread in the profile files (in the same order as the files in the file list). Each of the identifiers consists of four elements identifying the application, scaling parameter, data profile, and thread. Essentially, this is in an index into the drop-down boxes in Figure 4b. Note that the GUI elements in the ExaBoundsLite.nb notebook (e.g., Figure 2) are also updated to reflect loading of the files.

IBM Exascale Extrapolator output files can be loaded as well automatically. One can use the Get[] to load the file subsequently followed by a call to AppendAlgorithmAnalysis[] to load the data into the IBM ExaBounds data structures:

```
In[4]:= Get["/home/user/extrax - file.m"];
In[5]:= AppendAlgorithmAnalysis[ExaBoundsAlgInfo, predictedAlgorithm];
```

Here, ExaBoundsAlgInfo is the name of the IBM ExaBounds data structure and predictedAlgorithm is the default name of the symbol defined in the IBM Exascale Extrapolator output file which contains the extrapolated application profiles.

For the network performance models, the user is required to set the communication pattern of the application. The pattern needs to be defined per process profile (if the profile is extracted from a IBM Platform-Independent Software Analysis) or per class (cluster) profile (if the profile is extracted from an extrapolated IBM Exascale Extrapolator file), even though the pattern is the same pattern for all processes or classes (clusters) of processes. If the communication pattern is not known a-priori, the user should manually examine the communication matrix that shows how much data the processes exchange among each other during the execution of the program. Currently IBM ExaBounds supports three types of HPC patterns: *uniform* (uniform all-to-all), *shift* (with shift value *value*) and *nne2D* (2-dimension nearest-neighbor with an application domain size of D1·D2). The user can set the communication pattern as follows:

```
In[6]:= swProfile = ExaBoundsAlgInfo[algName][scalingConfiguration]
            [[dataProfile]][processId, 0];
In[7]:= swProfile = SetKeyValue[swProfile, "CommunicationPattern",
            < | "type" → "nne2D", "D1" → D1, "D2" → D2| >];
In[8]:= swProfile = SetKeyValue[swProfile, "CommunicationPattern",
            < | "type" → "shift", "value" → value| >];
In[9]:= swProfile = SetKeyValue[swProfile, "CommunicationPattern",
            < | "type" → "uniform"| >];
```

6.3 Loading architectures

Architectures can be automatically loaded using three functions that each take a path to the respective processor architecture, memory, or network JSON file:

```
In[10]:= LoadArchJSON[jsonFile];
In[11]:= LoadMemJSON[jsonFile];
In[12]:= LoadNetworkJSON[jsonFile];
```

In order to use LoadMemJSON[] the package PreDefinedMemorySpec‘ has to be manually loaded.

6.4 Selecting and modifying architectures

An architecture can be selected from the predefined configurations by overwriting the ExaBoundsState variable as:

```
In[13]:= architectureString = "T4240";
In[14]:= ExaBoundsState = ExaBoundsMergeState[ExaBoundsState,
             predefinedconfig[architectureString, ""]];
```

The predefinedconfig[] retrieves the architectural properties of the specified architecture. However, it may only specify a subset of the required properties. As a result, it needs to be merged (ExaBoundsMergeState[]) with the existing properties in ExaBoundsState to make sure that all properties are available and have some default value. The second argument of predefinedconfig[] is deprecated.

The architectureString identifies the architecture from the list of predefined architectures. The predefined architectures can be found in the PreDefinedMachineConfigs.m package or by inspection:

```
In[15]:= predefinedConfigID2Name

Out[15]= {"IBM8236 − E8C" → "IBM8236 − E8C(four − socketPower7)",
          "PowerLinux − 7R2" → "PowerLinux7R2(two − socketPOWER7+)", ...
```

It is possible to change architectural parameters using SetKeyValue[];

```
In[16]:= ExaBoundsState = SetKeyValue[ExaBoundsSate, "n1", 16];
```

This example sets the core count to 16. To inspect a parameter:

```
In[17]:= GetKeyValue[ExaBoundsSate, "n1"]
```

```
Out[17]= 16
```

Other examples for changing the architecture are provided below specifically for the network parameters.

To change the name of the network topology to full-mesh:

```
In[18]:= ExaBoundsState = SetKeyValue[ExaBoundsSate,
           "networkConfiguration", "full−mesh"];
```

To change the description of the network topology:

```
In[19]:= ExaBoundsState = SetKeyValue[ExaBoundsSate,
           "topologyDescription", "full−mesh", <| "a" → a, "p" → p| >];
```

To change the end node MPI stack latency:

```
In[20]:= ExaBoundsState = SetKeyValue[ExaBoundsSate,
           "nodeStackLatency", 0.000001];
```

To change the node-to-switch link bandwidth:

```
In[21]:= ExaBoundsState = SetKeyValue[ExaBoundsSate,
           "nodeSwitchBandwidth", 5 * 1024 * 1024 * 1024];
```

To change the mapping of the MPI processes for the *uniform* or *shift* pattern:

```
In[22]:= ExaBoundsState = SetKeyValue[ExaBoundsSate,
           "mappingDescription", <| "type" → "linear"| >];
```

To change the mapping of the MPI processes for the 2-dimension nearest-neighbor *nne2D* pattern for the full-mesh or fat-tree-2L network topology (more information about the mapping is provided in Appendix B):

```
In[23]:= ExaBoundsState = SetKeyValue[ExaBoundsSate, "mappingDescription",
            <| "type" → "linear", "d11" → d11, "d12" → d12| >];
```

6.5 Performing a single-thread single-node experiment

! Before starting an experiment it is key to realize that the processor architecture parameters are for a specific vector width while the loaded algorithm profiles are for an unspecified vector width. Therefor, one should *always* first convert an application profile to match the architected vector width of the target processor architecture. There are no exceptions, this also needs to be done for processor architectures without vector units: in that case, the profiles are converted to have only scalar instructions.

First, we define a wrapper function that converts our profiles for the correct vector width and then calls an appropriate function in IBM ExaBounds to predict the behavior of the application. In this example, we are interested in the compute execution time in seconds:

```
In[24]:= ExecutionTimeWrapper[archProperties_, algProperties_] :=
           Block[{convertedAlgProperties},
             convertedAlgProperties =
               GetAlgorithmVectorProperties[archProperties,
                 algProperties];
             Return[
               ExecutionSeconds[archProperties, convertedAlgProperties]
             ];
           ];
```

In this example, we first create a wrapper function called ExecutionTimeWrapper[] that takes as input a set of architectural and application properties. First, we convert the application properties to match the architecture using GetAlgorithmVectorProperties[] and store this in a temporary variable. Finally, we call ExecutionSeconds[] to predict the execution time and we return the results. A list of potential model calls is listed in Table 2.

Finally, we can call the wrapper function for every algorithm we loaded to get a list of execution times:

```
In[25]:= executionTimeList = ExecutionTimeWrapper[ExaBoundsState,
           GetAlgorithmKeyValueList[ExaBoundsAlgInfo, #]] & /@
           algorithmIdentifiers

Out[25]= {6.43, 12.79, 8.63}
```

6.6 Performing a multi-core single-node experiment

For a multi-core experiment we estimate the individual performance of a single thread using the single-core model [1]. To model congestion on the L3 cache and the memory bandwidth, we calculate the effective L3 cache size of each thread and evaluate the single-core model using the effective L3 cache size as the L3 size. Finally, we scale the results such that the total DRAM bandwidth of all threads on a processor does not exceed the maximum bandwidth.

6.6.1 Using the single-core models

We calculate the effective cache sizes by using SolveCachePressure[]. This function takes as argument the architecture properties as well as a mapping of algorithm profiles. The mapping can either be a list (for a fully custom mapping) or a single profile. In the latter case, it assumes that each core executes the same profile. As an example, consider the case where we wish to run two threads on a processor. This processor has 2 cores, each with a L1 cache of 32 kB, an L2 cache of 256 kB and a L3 cache of 4096 kB:

```
In[26]:= profile1 = GetAlgorithmKeyValueList[ExaBoundsAlgInfo,
           algorithmIdentifiers[[1]]];
In[27]:= profile2 = GetAlgorithmKeyValueList[ExaBoundsAlgInfo,
           algorithmIdentifiers[[2]]];
In[28]:= mapping = {1 → profile1, 2 → profile2};
In[29]:= pressures = SolveCachePressure[ExaBoundsState, mapping]

Out[29]= {1 → {"M0L1" → 32 * 1024, "M0L2" → 256 * 1024, "M1" → 1024 * 1024}, 2 →
           {"M0L1" → 32 * 1024, "M0L2" → 256 * 1024, "M1" → 3072 * 1024}}
```

The returned values are the effective caches sizes for each of the three layers of cache. Each thread has the full size of the private caches per core (M0L1 and M0L2) available, while only part of the size of the shared L3 (M1). Now, we determine the performance of each thread. Note that this does not account for a thread finishing early and freeing resources for other threads to use. We can determine the execution time as:

Table 2: List of model API calls. All functions return values for a single thread.

Function	**Description**
PeakFLOPSi[i_, archProperties_]	Peak performance in FLOPS at layer i[1]
AchievedFLOPSi[i_, archProperties_, algProperties_]	Achieved performance in FLOPS at layer i[2]
d0Optimize[archProperties_, algProperties_]	Achieved CPI
ExecutionCycles[archProperties_, algProperties_]	Total execution cycles
ExecutionSeconds[archProperties_, algProperties_]	Total execution seconds
FiMem[cacheLevel_, archProperties_, algProperties_]	Fraction of total instruction to a certain cache level
Poweri[i_, archProperties_, algProperties_]	Power on layer i
PoweriMemory[i_, archProperties_, algProperties_]	Memory power on layer i
AreaProcessor[archProperties_, algProperties_]	Processor area from McPAT

[1] See Table 3 for helper functions to specify the right cache levels and architecture layers.
[2] Currently only valid at the core layer.

Table 3: List of helper functions to identify levels and layers.

Function	**Description**
CoreLayer[]	Core layer
DieLayer[]	Die layer
SocketLayer[]	Socket layer
CardLayer[]	Card layer
RackUnitLayer[]	Rack unit layer
RackLayer[]	Rack layer
AisleLayer[]	Aisle layer
SystemLayer[]	System layer
MemoryL1Cache[]	Private L1 cache level
MemoryL2Cache[]	Private L2 cache level
MemoryL3Cache[]	Shared L3 cache level
MemoryDRAM[]	DRAM level

```
In[30]:= app = 1;
In[31]:= ExaBoundsState = SetKeyValue[ExaBoundsState, "M1",
           "M1" /. pressures[[app, 2]]];
In[32]:= executionTimeApp1 =
           ExecutionTimeWrapper[ExaBoundsState, profile1]

Out[32]= 4.23

In[33]:= app = 2;
In[34]:= ExaBoundsState = SetKeyValue[ExaBoundsState, "M1",
           "M1" /. pressures[[app, 2]]];
In[35]:= executionTimeApp2 =
           ExecutionTimeWrapper[ExaBoundsState, profile2]

Out[35]= 8.23
```

6.6.2 Using the multi-core models

An easier method is to use the multi-core models available in IBM ExaBounds. These functions automatically calculate the cache pressures and return a list of values, one for each thread. We get the profiles, convert them for the right vector architecture and create a mapping similar as before:

```
In[36]:= profile1 = GetAlgorithmVectorProperties[archProperties,
           GetAlgorithmKeyValueList[ExaBoundsAlgInfo,
           algorithmIdentifiers[[1]]]];
In[37]:= profile2 = GetAlgorithmVectorProperties[archProperties,
           GetAlgorithmKeyValueList[ExaBoundsAlgInfo,
           algorithmIdentifiers[[2]]]];
In[38]:= mapping = {1 → profile1, 2 → profile2};
```

Using the mapping we can simply call the corresponding multi-core model function. Their naming are similar as the single-core functions, but are preceded by MC. Note that not all functions are yet available in a multi-core version.

```
In[39]:= MCExecutionSeconds[archProperties, mapping]

Out[39]= {4.23, 8.23}
```

The multicore models have a shorter method to run a homogeneous workload (all threads have the same properties) an a processor:

```
In[40]:= MCExecutionSeconds[archProperties, profile1, 10]

Out[40]= 7.56
```

This example runs PROFILE1 on 10 cores in parallel. The multi-core functions automatically limit the maximum number of parallel threads by the core count. No warning is given if this happens. The same function can also be used to perform a single-core experiment:

```
In[41]:= MCExecutionSeconds[archProperties, profile1, 1]

Out[41]= 4.23
```

or, as the integer is optional:

```
In[42]:= MCExecutionSeconds[archProperties, profile1]

Out[42]= 4.23
```

6.7 Performing a multi-node experiment

6.8 Processor design-space exploration

The steps discussed in this approach are easily extended to a full design-space exploration. By simply looping over the steps as described for experiments described in this section with different architecture and application design points, an exploration can be performed. A design-space exploration without power models is usually quite rapid (in the order of seconds per design point). However, when enabling the power models the run time can increase dramatically due to McPAT. Due to the large amount of memory McPAT uses, a machine running the design-space exploration can become unusable. It can be recommended to store intermediate results once in a while (store all live elements in the Mathematica kernel).

As an example, we wish to understand the impact of different L2 and L3 caches sizes on the performance of our application and power consumption of the processor. Therefor, we first retrieve the application profile (in the example, a default profile from IBM ExaBounds), set the base architecture and convert the profile to the right vector width:

```
In[43]:= profile1 = GetAlgorithmKeyValueList[ExaBoundsAlgInfo,
             selectedAlgorithm];
In[44]:= ExaBoundsState = MergeKeyValueList[ExaBoundsState,
             predefinedconfig["XeonE5 − 2697v3"]];
In[45]:= profile1 = GetAlgorithmVectorProperties[ExaBoundsState, profile1];
```

We continue to define the design space:

```
In[46]:= L3sizes = {8, 16, 32, 64} ∗ 1024^2;
In[47]:= L2sizes = {128, 256, 512, 1024} ∗ 1024;
```

Now we can run the design-space exploration itself:

```
In[48]:= result = {};
In[49]:= designStr = {};
In[50]:= Do[
             Do[
               tmState = SetKeyValue[ExaBoundsState, "M0L2", M0L2];
```

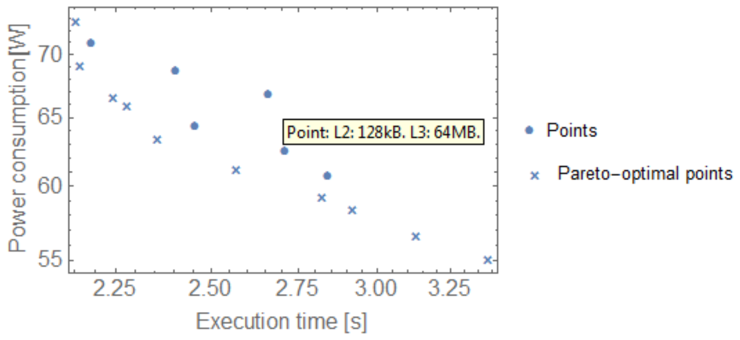


Figure 8: Example DSE Pareto plot.

```
        tmpState = SetKeyValue[tmpState, "M1", M1];
        runtime = MCExecutionSeconds[tmpState, profile1, 14];
        power = Poweri[SocketLayer[], tmpState, profile1, 14];
        AppendTo[result, {runtime, power}];
        AppendTo[designStr, "L2 :  " <> ToString[M0L2/1024]
         <> "kB. L3 :  " <> ToString[M1/1024^2] <> "MB."];
        , {M0L2, L2sizes}
      ]
    , {M1, L3sizes}
    ]
```

We start by initializing the RESULT and DESIGNSTR lists as an empty lists. We iterate over all pairs of L2 and L3 sizes and for each iteration we first create an architecture description in TMPSTATE with the new cache sizes[2]. We the proceed to use the model to predict both the execution time and the power consumption of the socket and add that as a tuple to the RESULT list. Finally, we also append a human-readable string that describes the design point to DESIGNSTR.

After generating the result, we can use the PARETOPLOT[] function from DSEVISUALIZATION.M to visualize the results:

```
In[51]:= ParetoPlot[result, designStr,
        BaseStyle → {FontSize → 16},
        Frame → True,
        FrameLabel → {"Executiontime[s]", "Powerconsumption[W]"}]
```

This generates the plot as shown in Figure 8. Hovering the mouse over one of the bullets will show the human-readable string that we generated in DESIGNSTR to describe the point.

Execution of the design-space exploration can take some time as McPAT is slow to execute. Furthermore, during the design-space exploration Mathematica keeps refreshing the GUI elements in the main notebook regularly, slowing down the process. One way to somewhat speed up the design-space exploration is, after executing the main notebook (ExaBoundsLite.nb) once, to remove the output cells of that notebook. Go to ExaBoundsLite.nb, and select in the *Cell*

[2]Note that we do not need to convert the application profile again as we do not update the vector width. In case of a design-space exploration over multiple vector widths, the profile should be converted using GETALGORITHMVECTORPROPERTIES[] every time a new vector width is configured.

menu *Delete All Output*. This removes all GUI elements (and they subsequently cannot be used anymore), but IBM ExaBounds stays live in the kernel.

6.9 Network and full-system design-space exploration

First we provide an example of a design-space exploration study across different network configurations. The performance metric is the communication time. We assume that the user has an extrapolated profile for an application with one single cluster of MPI processes and a *uniform* communication pattern, such as Graph 500.

The user defines the network architecture properties and the mapping of MPI processes to nodes.

```
In[52]:= ExaBoundsState = SetKeyValue[ExaBoundsState,
            "networkConfiguration", "fat − tree − 3L"];
In[53]:= ExaBoundsState = SetKeyValue[ExaBoundsState,
            "mappingDescription", <| "type" → "linear" |>];
In[54]:= ExaBoundsState = SetKeyValue[ExaBoundsState,
            "nodeStackLatency", 0.0000009];
In[55]:= ExaBoundsState = SetKeyValue[ExaBoundsState,
            "switchLatency", 0.0000007];
In[56]:= ExaBoundsState = SetKeyValue[ExaBoundsState,
            "switchSwitchBandwidth", 5 ∗ 1024 ∗ 1024 ∗ 1024];
In[57]:= ExaBoundsState = SetKeyValue[ExaBoundsState,
            "nodeSwitchBandwidth", 5 ∗ 1024 ∗ 1024 ∗ 1024];
In[58]:= ExaBoundsState = SetKeyValue[ExaBoundsState,
            "nodeSwitchLinkLatency", 0.0000000025];
In[59]:= ExaBoundsState = SetKeyValue[ExaBoundsState,
            "switch1Switch2LinkLatency", 0.000000012];
```

The user also defines the variations of network architectures to be analyzed. Let's assume that the user analyzes multiple network scenarios of fat-tree-3L topologies for an application with 262144 MPI processes. IBM ExaBounds currently supports assigning one MPI process per node (single-threaded process) and the number of nodes needs to be equal to the number of MPI processes, meaning that the system is fully populated. For a fat-tree-3L, the number of nodes is equal to m1·m2·m3 which, in this example, should be 262144.

```
In[60]:= w0 = {1, 1, 1, 1};
In[61]:= w1 = {64, 16, 128, 32};
In[62]:= w2 = {64, 16, 64, 32};
In[63]:= m1 = {64, 64, 128, 256};
In[64]:= m2 = {64, 64, 32, 32};
In[65]:= m3 = {64, 64, 64, 32};
In[66]:= ft3LConfs = Transpose[{w0, w1, w2, m1, m2, m3}];
```

For each of the four network scenarios defined above, the user can retrieve the communication time as follows.

```
In[67]:= Do[
            swProfile = SetKeyValue[swProfile, ”topologyDescription”,
             <| "w0" → conf[[1]], "w1" → conf[[2]], "w2" → conf[[3]],
            "m1" → conf[[4]], "m2" → conf[[5]], "m3" → conf[[6]] |>];
In[68]:= commTime = GetCommunicationTime[ExaBoundsState, swProfile];
  , {conf, ft3LConfs} ];
```

The example above is a simple design-space exploration across the parameters that describe the network topology (in the case of fat-tree-3L, these parameters are w0, w1, w2, m0, m1 and m2). The user can extend this study with more hardware parameters such as latencies or bandwidths.

For the network models, IBM ExaBounds currently assumes that all links between switches have the same latency and the same bandwidth. It can be however easily extended to support multiple latencies and bandwidths. Moreover, if the user gets a *missingModel* warning, the user should double-check the name of the topology and the name of the communication pattern. If the topology name or communication pattern is not recognized, IBM ExaBounds does not exit, but prints a warning (Missing model for *topology*: "average link latency", "average number of links", "effective bandwidth"). Thus, the user should always check the warnings.

Next, to estimate the power consumption of a full system (processors and network), the user must first calculate the processor time (compute, *compTime*) and the network time (communication, *commTime*). In this example, there is only one class of processes, thus we calculate the compute and communication times once (per hardware architecture) and add them to estimate the execution time of the MPI application. This model is reasonable for HPC applications which are typically programmed in a way that minimizes the waiting times and the data dependencies between processes. If there are multiple classes (clusters) of processes, the user should 1. calculate the compute and communication times per cluster, 2. add them to estimate the time of the process representative of the cluster and 3. estimate the execution time of the MPI application by calculating the maximum time across clusters. This is one possible model to predict application time performance. The user is however free to use any other model for estimating the application time, starting from the IBM ExaBounds processor and network estimates per process or cluster.

In the following we show how the user can estimate the power consumption of the full system (an application with one cluster of MPI processes).

```
In[69]:= procs = m1 * m2 * m3;
In[70]:= appTime = compTime + commTime;
In[71]:= compPower = Poweri[SocketLayer[], ExaBoundsState, swProfile, False];
In[72]:= compPowerStatic = powerProcMem[[1]];
In[73]:= compDynamicPower = powerProcMem[[2]];
In[74]:= compStaticEnergy = procs * compPowerStatic * appTime;
In[75]:= compDynamicEnergy = procs * compPowerDynamic * compTime;
In[76]:= commDynamicEnergy =
           procs * GetNetworkDynamicEnergy[ExaBoundsState, swProfile];
In[77]:= commStaticEnergy =
           GetNetworkStaticEnergy[ExaBoundsState, swProfile, appTime];
In[78]:= systemPower = (compStaticEnergy + compDynamicEnergy+
           commStaticEnergy + commDynamicEnergy)/appTime
```

The design-space exploration presented at the beginning of this section could be easily extended with the power analysis above so that the user can perform a complete power-performance trade-off analysis.

7 Contributors

- Rik Jongerius

- Giovanni Mariani
- Andreea Anghel
- Gero Dittmann
- Sandeep Poddar
- Phillip Stanley-Marbell

8 Acknowledgement

This work was conducted in the context of the joint ASTRON and IBM DOME project and was funded by the Dutch Ministry of Economische Zaken, and the Province of Drenthe.

A IBM ExaBounds API

This appendix describes the most commonly used functions in the ExaBounds API for design-space explorations.

Package ExaBoundsGeneric:

GetKeyValue[keyValueList_, key_]:
Get the value of key key from key-value list keyValueList.

SetKeyValue[keyValueList_, key_, value_]:
Update the key key in keyValueList with value value and return the updated key-value list.

MergeKeyValueList[baseList_, UpdateKeyValuePairs_]:
Update the baseList key-value pairs with the key-value pairs in UpdateKeyValuePairs, returns the resulting key-value list. All keys in UpdatekeyValuePairs have to be present already in baseList.

GetAlgorithmName[algInfo_, index_]:
Get the name of algorithm with index Index from the ExaBoundsAlgInfo structure. Pass ExaBoundsAlgInfo as the first argument.

GetAlgorithmKeyValueList[algInfo_, algoIndex_,scalingParameters_, dataProfileId_,threadId_] or GetAlgorithmKeyValueList[algInfo_, sel_List]:
Retrieve algorithm properties as a key-values list from the ExaBoundsAlgInfo structure. The algorithm is identified as a 4-tuple describing the index, scaling parameters, data profile and thread id, either as individual arguments or as a list.

SetAlgorithmKeyValue[algInfo_, sel_List, key_, value_]:
Update a property key of a specific algorithm (identified with a 4-tuple in sel) with value in the ExaBoundsAlgInfo structure. The function is HoldFirst, and does not return anything.

The following functions can be used to determine the 4-tuple of algorithm name, scaling configuration, data profile, and thread for applications loaded in *the* EXABOUNDSALGINFO *structure:*

GetScalingParameters[algDataAssociation_]:
Get the scaling parameters that where used to describe the scale of the algorithm. ALGDATAASSOCIATION is an Association with data for one application, identified as EXABOUNDSALGINFO["ALGORITHM NAME"].

GetScalingConfigurations[algDataAssociation_]:
Get all scaling configurations that where profiled.

GetScalingConfigurationsHaving[algDataAssociation_,partialScalingConfiguration_]:
Get all scaling configurations that have certain scaling parameters. The scaing configuration is given as an Association.

GetDataProfileCount[algInfo_, alg_, scaling_]:
Get the number of data profiles from an application in EXABOUNDSALGINFO with name ALG and scaling properties SCALING.

GetThreads[algInfo_, alg_, scaling_, profile_]:
Get the thread identifiers for an application in EXABOUNDSALGINFO with name ALG, scaling properties SCALING and data profile PROFILE.

Package ALGORITHMPROPERTIES:

AddAlgorithm[algInfoStructure_, jsonfile_]:
Load a IBM Platform-Independent Software Analysis file into EXABOUNDSALGINFO from file JSONFILE.

AppendAlgorithmAnalysis[algInfoStructure_, newAlgoAnalysis_:
Append the application profiles in NEWALGOANALYSIS to EXABOUNDSALGINFO. Used to append algorithms in IBM Exascale Extrapolator *.m-files

Package VECTORSIZECONVERT:

GetAlgorithmVectorProperties[archProperties_, algProperties_]:
Convert the application properties in ALGPROPERTIES to the vector size in the architecture described by ARCHPROPERTIES and return the result. Set the architected vector width to zero for purely scalar instructions.

Package PREDEFINEDMACHINECONFIGS:

LoadArchJSON[file_]:
Load a processor architecture from FILE.

predefinedConfigID2Name:
Replacement list with all available processor architectures and human-readable name.

predefinedconfig[configname_]:
Retrieve an architecture named CONFIGNAME (name matching id in PREDEFINEDCONFIGID2NAME. Always update EXABOUNDSSTATE using MERGEKEYVALUELIST.

Package PreDefinedNetworkConfigs:

LoadNetworkJSON[file_]:
Load a network architecture from file.

predefinedConfigID2NameNetwork:
Replacement list with all available network architectures and human-readable name.

GetNetworkConfig[configname_]:
Retrieve a network architecture named configname (name matching id in predefinedConfigID2NameNetwork. Always update ExaBoundsState using MergeKeyValueList.

Package PreDefinedMemorySpec:

LoadMemJSON[file_]:
Load a memory architecture from file.

predefinedConfigID2NameMem:
Replacement list with all available memory architectures and human-readable name.

predefinedconfigMem[configname_]:
Retrieve a memory architecture named configname (name matching id in predefinedConfigID2NameNetwork. Note that the memory architecture name is stored in ExaBoundsState in key *DRAMType*, these properties are not merged into ExaBoundsState.

Package CPUPerformanceModels:

CacheHitrate[cacheLevel_, archProperties_, algProperties_]:
Get the hit rate on cache level cacheLevel for application algProperties running on architecture archProperties. The cache levels are listed in Table 3 and can be MemoryL1Cache[], MemoryL2Cache[], or MemoryL3Cache[].

CacheMissrate[cacheLevel_, archProperties_, algProperties_]:
Get the miss rate on cache level cacheLevel for application algProperties running on architecture archProperties.

AchievedFLOPSi[i_, archProperties_, algProperties_]:
Get the achieved performance at layer i for application algProperties running on architecture archProperties. The layers are listed in Table 3 and range from CoreLayer[] to SystemLayer[].

BappiDmo[i_, archProperties_, algProperties_]:
Get the achieved memory bandwidth at layer i for application algProperties running on architecture archProperties.

d0[archProperties_, algProperties_]:
Get the performance in cycles per instruction for application algProperties running on architecture archProperties.

ExecutionCycles[archProperties_, algProperties_]:
Get the execution time in clock cycles for application algProperties running

on architecture archProperties.

ExecutionSeconds[archProperties_, algProperties_]:
Get the execution time in seconds for application algProperties running on architecture archProperties.

ClearCPUPerformanceModelsCache[]:
Clear all cached results from the performance models.

Package CommunicationPerformanceModels:

GetAverageNetworkLinkLatency[archProperties_,algProperties_,tag_]:
Returns the average number of links (if tag is 1) or the average link latency (if the tag is 0) that an inter-process communication (MPI) message will incur while traversing the network described in archProperties for the communication pattern in algProperties, when the mapping of MPI processes to end nodes is linear and according to the *mappingDescription* parameter in archProperties and the routing is shortest-path with optimal routing when multiple equal-hop routes are available. Default value is 0 (if the topology or the pattern is not supported).

UniformAverageNetworkLinkLatency[archProperties_,tag_]:
Returns the average number of links (if tag is 1) or the average link latency (if the tag is 0) that an inter-process communication (MPI) message will incur while traversing the network described in archProperties when the communication pattern is uniform, the mapping of MPI processes to end nodes is linear and the routing is shortest-path with optimal routing when multiple equal-hop routes are available. Default value is 0 (if the topology or the mapping is not supported).

NNE2DAverageNetworkLinkLatency[archProperties_,tag_,D1_,D2_]:
Returns the average number of links (if tag is 1) or the average link latency (if the tag is 0) that an inter-process communication (MPI) message will incur while traversing the network described in archProperties when the communication pattern is 2-dimension nearest-neighbor (the 2-dimension application-domain being defined as D1·D2), the mapping of MPI processes to end nodes is linear according to the *mappingDescription* parameter in archProperties and the routing is shortest-path with optimal routing when multiple equal-hop routes are available. Default value is 0 (if the topology or the mapping is not supported).

ShiftAverageNetworkLinkLatency[archProperties_,tag_,shiftValue_]:
Returns the average number of links (if tag is 1) or the average link latency (if the tag is 0) that an inter-process communication message (MPI) will incur while traversing the network described in archProperties when the communication pattern is shift (the shift value being "shiftValue"), the mapping of MPI processes to end nodes is linear and the routing is shortest-path with optimal routing when multiple equal-hop routes are available. Default value is 0 (if the topology or the mapping is not supported).

NodeEffectiveBandwidth[archProperties_,algProperties_]:
Returns the node effective TX bandwidth for the network described in archProperties and the communication pattern in algProperties, for when the mapping of MPI processes to end nodes is linear according to the *mappingDescription* parameter in archProperties and the routing is shortest-path with

optimal routing when multiple equal-hop routes are available. Default value is the value of the *switchSwitchBandwidth* parameter (if the topology is not supported).

UniformNodeEffectiveBandwidth[archProperties_]:
Returns the node effective bandwidth for the network described in archProperties when the communication pattern is uniform, the mapping of MPI processes to end nodes is linear and the routing is shortest-path with optimal routing when multiple equal-hop routes are available. Default value is the value of the *switchSwitchBandwidth* parameter (if the topology is not supported).

NNE2DNodeEffectiveBandwidth[archProperties_,D1_,D2_]:
Returns the node effective bandwidth for the network described in archProperties when the communication pattern is 2-dimension nearest-neighbor (the 2-dimension application-domain being defined as D1·D2), the mapping of MPI processes to end nodes is linear and according to the *mappingDescription* parameter in archProperties and the routing is shortest-path with optimal routing when multiple equal-hop routes are available. Default value is the value of the *switchSwitchBandwidth* parameter (if the topology is not supported).

ShiftNodeEffectiveBandwidth[archProperties_,shiftValue_]:
Returns the node effective bandwidth for the network described in archProperties when the communication pattern is shift (the shift value being *shiftValue*), the mapping of MPI processes to end nodes is linear and the routing is shortest-path with optimal routing when multiple equal-hop routes are available. Default value is the value of the *switchSwitchBandwidth* parameter (if the topology is not supported).

GetCommunicationTime[archProperties_,algProperties_]:
Returns the communication time for the software profile in algProperties and the network hardware description in archProperties. Important note: this is not necessarily the communication time of the application, but the communication time per class/cluster of processes (if algProperties was extracted from an IBM Exascale Extrapolator file) or per MPI process (if algProperties was extracted from a IBM Platform-Independent Software Analysis file with multiple process profiles).

Package CommunicationPowerModels:

GetNumberElectricalLinks[archProperties_,algProperties_,level_]:
Returns the average number of electrical links that an inter-process communication (MPI) message traverses through the network. If level is 1, then only the end nodes connected to their switches use electrical links, thus the function returns 2. If level is 2, then all links in the network will be electrical and the function returns the average number of links traversed by an inter-process communication (MPI) message, which is the return value of averageNetworkLinkLatency[1, algProperties_,archProperties_]. The return value is either 2 (when only the links connecting the end nodes to the switches are electrical), or the return value of the GetAverageNetworkLinkLatency[archProperties_,algProperties_,1] function call (when all links are electrical).

GetNumberOpticalLinks[archProperties_,algProperties_,level_]:

Returns the average number of optical links that an inter-process communication (MPI) message traverses through the network. If level is 1, then all links except the end nodes connected to their switches use optical links, thus the function returns the difference between the average number of traversed links and the average number of traversed electrical links. If level is 2, then all links in the network will be electrical and the function returns 0. The return value is either GetAverageNetworkLinkLatency[archProperties_,algProperties_,1]-2 (when all except the links connecting the end nodes to the switches are optical), or 0 (when no links are optical).

GetNetworkDynamicEnergy[archProperties_,algProperties_]:
Returns the dynamic energy per class/cluster of processes (if algProperties was extracted from an IBM Exascale Extrapolator file) or per process (if algProperties was extracted from a IBM Platform-Independent Software Analysis file with multiple process profiles).

GetTotalNumberSwitches[archProperties_]:
Returns the total number of switches in the network described in archProperties. Default value is 1 (if the topology is not supported).

GetNetworkStaticEnergy[archProperties_,algProperties_,appTime_]:
Returns the static power of the network described in archProperties, when running the application in algProperties for a total duration of time of appTime, comprising both compute and communication time.

Package CPUMultithreadedModels:

All multithreaded model functions have the same function prototype as their respective single-core versions with two optional arguments: the first argument thread and the last argument homogeneousThreads. If thread is specified, the function returns the value for the specified thread. If thread is not specified, a list is returned with a value for each thread. The last optional argument is used to specify the workload. The workload can be specified as:

- a single profile in algProperties, with homogeneousThreads unspecified (or set to 1) for a single-threaded workload (only one core used), e.g. MCdo[architecture, profile1].
- a single profile in algProperties, with homogeneousThreads set to a value N for a workload with N identical threads (when N is larger than the core count, the number of threads will equal the number of cores), e.g. MCdo[architecture, profile1, 14] for 14 threads.
- a replacement list of profiles in algProperties. Specify which profile runs on which core, e.g. MCdo[architecture, {1 $\rightarrow$ profile1, 4 $\rightarrow$ profile2}].

MCd0[archProperties_, algProperties_, homogeneousThreads_]:
Multi-core version of d0, retrieve performance in cycles per instruction for all threads specified by algProperties and homogeneousThreads as specified above.

MCd0[thread_, archProperties_, algProperties_, homogeneousThreads_]:

Multi-core version of D0, retrieve performance in cycles per instruction for THREAD.

MCExecutionSeconds[thread_, archProperties_, algProperties_, homogeneousThreads_]:
Multi-core version of ExecutionSeconds.

MCExecutionCycles[thread_, archProperties_, algProperties_, homogeneousThreads_]:
Multi-core version of ExecutionCycles.

MCCacheMissrate[thread_, level_, archProperties_, algProperties_, homogeneousThreads_]:
Multi-core version of CacheMissrate.

MCBappiDmo[thread_, level_, archProperties_, algProperties_, homogeneousThreads_]:
Multi-core version of BappiDmo.

MCAchievedFLOPSi[thread_, level_, archProperties_, algProperties_, homogeneousThreads_]:
Multi-core version of AchievedFLOPSi.

MCGetKeyValue[algProperties_, key_]:
Multi-core version of GetKeyValue. This function returns per-thread values for a given key.

Package CPUPowerModels:

The workload for the power models can be specified in a similar way as for the multi-threaded performance models in CPUMultithreadedModels.
Poweri[i_, archProperties_, algProperties_, homogeneousThreads_]:
Get the system power consumption at layer I.

PoweriMemory[i_, archProperties_, algProperties_, homogeneousThreads_]:
Get the memory power consumption at layer I.

AreaProcessor[archProperties_, algProperties_, homogeneousThreads_]:
Get the total chip area at layer I (McPAT results).

ClearCPUPowerModelsCache[]:
Clear all cached results from the power models.

DRAMmodel:
Set the DRAM model, set to either "MeSAP" or "CACTI".

Package DSEVisualization:

ParetoPlot[data_List, tooltip_List]:
Displays Pareto plot. Data is a list of tuples, where each tuple is a design point. Tooltip is an optional argument that can be used to pass a list of human-readable strings describing each design point (both input lists should have the same length). This string is displayed in the plot if the mouse pointer is hovered over a design point. ParetoPlot[] accepts similar options as the Mathematica function ListPlot. Besides the default options, additional op-

tions are: PlotType—select the plot type, valid values are ListPlot, ListLogPlot, ListLogLinearPlot, and ListLogLogPlot (default)—, ShowParetoPoints—separately show Pareto points (default: True)—, ParetoPointsMarker—marker used for the Pareto points (default: cross)—, and TooltipPrefix—string prefix used for all tooltips in tooltip (default: Point).

B Architecture parameters

B.1 Processor architecture parameters

year [-]: Year of introduction.
Lnode [nm]: Technology node.
T [K]: Operating temperature.
inorder [True/False]: Inorder core architecture.
n0 [-]: Core issue width.
n0dispatch [-]: Core dispatch width.
n0threads [-]: Number of SMT threads per core.
n0int [-]: Number of integer functional units per core.
n0DPFP [-]: Number of floating-point functional units per core.
n0mem [-]: Number of load/store units per core.
n0control [-]: Number of branch units per core.
n0vectorFU [-]: Number of vector units per core.
n0vectorbits [bit]: Vector width.
T0intop [cycles]: Average integer instruction forwarding delay.
T0intmul [cycles]: Average integer multiply instruction forwarding delay.
T0intdiv [cycles]: Average integer division instruction forwarding delay.
T0fpop [cycles]: Average floating-point instruction forwarding delay.
T0fpmul [cycles]: Average floating-point multiply instruction forwarding delay.
T0fpdiv [cycles]: Average floating-point division instruction forwarding delay.
T0vectorintop [cycles]: Average vector integer instruction forwarding delay.
T0vectorintmul [cycles]: Average vector integer multiply instruction forwarding delay.
T0vectorintdiv [cycles]: Average vector integer division instruction forwarding delay.
T0vectorfpop [cycles]: Average vector floating-point instruction forwarding delay.
T0vectorfpmul [cycles]: Average vector floating-point multiply instruction forwarding delay.
T0vectorfpdiv [cycles]: Average vector floating-point division instruction forwarding delay.
n0Bits [bit]: Core data path width.
n1 [-]: Number of cores per die.
n2 [-]: Number of dies per socket.
n3 [-]: Number of sockets per card.
n4 [-]: Number of cards per rack unit.
n5 [-]: Number of rack units per rack.
n6 [-]: Number of racks per aisle.

n7 [-]: Number of aisles per system.
M0L1 [byte]: Private L1 cache size per core.
L1dagran [byte]: L1 data cache granularity (cache line size).
L1dassoc [-]: L1 data cache associativity.
L1iagran [byte]: L1 instruction cache granularity (cache line size).
L1iassoc [-]: L1 instruction cache associativity.
M0L2 [byte]: Private L2 cache size per core.
L2dagran [byte]: L2 cache granularity (cache line size).
L2dassoc [-]: L2 cache associativity.
M1 [byte]: Shared L3 cache size per die.
L3dagran [byte]: Shared L3 cache granularity (cache line size).
L3dassoc [-]: L3 cache associativity.
M2 [byte]: DRAM memory per socket.
nDIMMs [-]: Number of memory DIMMs per socket.
nRanksPerDIMM [-]: Number of ranks per memory DIMM
DRAMType [-]: Memory architecture (string with name of memory architecture used).
T0L1latency [cycles]: L1 cache access latency.
T0L2latency [cycles]: L2 cache access latency (including L1 latency).
T0L3latency [cycles]: L3 cache access latency (including L1 and L2 latency).
T0DRAMlatency [cycles]: DRAM access latency (including cache latency).
B0Dmo [byte/s]: Core to L1/L2 cache bandwidth.
B1Dmo [byte/s]: L2 cache to L3 cache bandwidth.
B2Dmo [byte/s]: L3 cache to DRAM bandwidth.
f0 [Hz]: Core clock speed.
f1 [Hz]: Processor bus clock speed.
n0pipe [-]: Core pipeline depth.
n0frontpipe [-]: Core front-end pipeline depth.
n0ROB [-]: Core reorder buffer size.
n0IQ [-]: Core issue queue size.
n0MSHR [-]: Core miss-status holding register count.
V0 [V]: Core voltage.

B.2 Memory architecture parameters

MemoryId [-]: Name of memory configuration.
MemoryType [-]: Memory type (DDR3/4).
MemoryRankSize [Gb]: Size of a memory rank.
DevicesPerRank [-]: Number of memory devices (chips) per rank.
DeviceChipSize [MB]: Size of memory device.
DataWidth [bit]: Data bus width of a memory device.
DataRate [1/cycle]: Single or dual data rate.
BurstLength [-]: Maximum supported burst length.
ClkMHz [Hz]: Memory bus clock frequency.
REFI [cycles]: Refresh interval. DRAM is required to go through a refresh cycle once in this refresh interval.
RFC [cycles]: Refresh cycle. The time between the start of a refresh and activation commands.
RAS [cycles]: Row access strobe. Minimum time interval between a row activation and precharge command for data restoration.

RP [cycles]: Row precharge. Time to close an active row.
idd0 [mA]: Operating one bank Active-Precharge current. The command consists of one ACT and one PRE command to one bank in sequence. All other banks are in precharged state.
idd2n [mA]: Precharge Standby Current. All banks are in precharged state.
idd3n [mA]: Active Standby current. One bank is in active state and others are in precharged state.
idd4r [mA]: Burst Read Current. Continuous burst reads using RD commands. All banks open.
idd4w [mA]: Burst Write Current. Continuous burst writes using WR commands. All banks open.
idd5b [mA]: Refresh current. Issued every tRFC cycles.
vdd [V]: Operating voltage.

B.3 Network architecture parameters

In the following list of parameters that describe the network architecture, "X:Y" means that parameter "Y" is defined as part of the "X" JSON object. The same for "X:Y:Z" which means that "Z" is defined as part of the "Y" JSON object which is a part of the "X" JSON object.

id [-]: Network topology ID. Currently it can be fat-tree-2L, fat-tree-3L, 2DhyperX, full-mesh, torus-1D, torus-2D, torus-3D, torus-5D.
name [-]: Network topology name. Currently it can be Fat tree 2L, Fat tree 3L, 2DhyperX, Full-mesh, Torus-1D, Torus-2D, Torus-3D, Torus-5D.

ExaBounds currently provides network models for the following topologies:

- uniform communication pattern: full-mesh, fat-tree-2L, fat-tree-3L, 2D Hyper-X, multi-dimensional tori.
- shift communication pattern: full-mesh, fat-tree-2L, fat-tree-3L.
- 2-dimension nearest-neighbor communication pattern: full-mesh, fat-tree-2L, fat-tree-3L, 2D Hyper-X.

config:topologyDescription [-]: A list of network topology parameters that describe the network configuration. This object includes multiple parameters depending on the topology id (see below).

Topology description parameters per topology id:

- full-mesh: "a" (number of switches in the network) and "p" (number of end nodes connected to a switch).
- fat-tree-2L: "w0" (the number of up-links in level-1 switches), "w1" (the number of up-links in level-2 switches), "m1" (the number of down-links in level-1 switches) and "m2" (the number of down-links in level-2 switches).
- fat-tree-3L: "w0" (the number of up-links in level-1 switches), "w1" (the number of up-links in level-2 switches), "w2" (the number of up-links in level-3 switches),"m1" (the number of down-links in level-1 switches),

"m2" (the number of down-links in level-2 switches) and "m3" (the number of down-links in level-3 switches).

- 2DhyperX: "p" (the number of end nodes connected to a switch), "d1" (the number of switches in the X dimension) and "d2" (the number of switches in the Y dimension).
- torus-Nd: "p" (the number of end nodes connected to a switch), "d1" (the number of switches in the 1st dimension) ... "dN" (the number of switches in the N-th dimension), where N≥1.

config:nodeStackLatency [second]: End node MPI stack latency.
config:switchLatency [second]: Switch internal latency.
config:nodeSwitchLinkLatency [second]: Link latency between the end node and the switch to which the end node is connected.
config:switch1Switch2LinkLatency [second]: Link latency between two directly connected switches. For a fat-tree-2L and fat-tree-3L topologies, this is the link between a level-1 switch and a level-2 switch. For a full-mesh, this is the link between two directly connected switches. For a 2D Hyper-X, this is the link between two directly connected switches. Similar for multi-dimension tori topologies.
config:switch2Switch3LinkLatency [second]: [Valid only for fat-tree 3L topologies] Link latency between a level-2 switch and a level-3 switch.

config:nodeSwitchBandwidth [byte/s]: Link bandwidth between the end node and the switch to which the end node is connected.
config:switchSwitchBandwidth [byte/s]: Link bandwidth between two directly connected switches. Currently supported: all switch-switch links have the same bandwidth.

Currently ExaBounds supports two possible combinations of types (electrical/optical) of links in the network: 1. All links in the network are electrical. 2. The end nodes are connected to their switch via electrical links and all the remainder links connecting switches are all optical.

config:nodeSwitchLinkType [electrical/optical]: Type of link between the end node and the switch to which the end node is connected.
config:switch1Switch2LinkType [electrical/optical]: Type of link between two directly connected switches. For a fat-tree-2L and fat-tree-3L topologies, this is the link between a level-1 switch and a level-2 switch. For a full-mesh, this is the link between two directly connected switches. For a 2D Hyper-X, this is the link between two directly connected switches. Similar for multi-dimension tori topologies.
config:switch2Switch3LinkType [electrical/optical]: [Valid only for fat-tree 3L topologies] Type of link between a level-2 switch and a level-3 switch.
config:electricalLinkEnergyPerBit [pJ]: Energy per bit in electrical links.
config:opticalLinkEnergyPerBit [pJ]: Energy per bit in optical links.
config:switchLogicEnergyPerBit [pJ]: Energy per bit in the switch.
config:switchStaticPower [W]: Static power per switch.

When dealing with applications running on parallel systems, the MPI rank mapping plays an important role for the system performance. For the uniform and shift communication patterns, ExaBounds currently provides network performance models only for linear mappings. For the 2-dimension nearest-neighbor pattern, we cover more than just linear mappings as follows.

ExaBounds models mappings where the application domain (the grid of processes that determines the 2-dimension nearest-neighbor pattern) is partitioned into equal-sized application sub-domains. The domain of hardware nodes is assumed to be also partitioned into same-sized hardware sub-domains. E.g., for the full-mesh topology, the partitioning of the set of hardware nodes into sub-sets each belonging to the same switch is such a partition. The size of an application sub-domain should be equal to the size of a hardware sub-domain. The mappings that ExaBounds covers are those that bijectively map the application sub-domains onto the hardware sub-domains. Thus not only must the total number of end nodes in the system match the total number of application processes, but the processes in an application sub-domain should also fully populate the compute nodes in a hardware sub-domain. Currently, for the 2-dimension nearest-neighbor pattern, ExaBounds provides network performance models only for full-mesh, fat-tree 2L, fat-tree 3L and 2D Hyper-X topologies.

config:mappingDescription:type [linear]: Always set to linear. For uniform and shift application communication patterns, only this parameter is required for describing the mapping of MPI processes to end nodes. For the 2-dimension nearest-neighbor pattern, additional parameters should to be defined as follows (see below).

If the 2-dimension nearest-neighbor application domain is of size D1· D2.

- full-mesh: the mapping should be described by a tuple {d11,d12}, where d11·d12=p, d11 divides D1 and d12 divides D2, meaning that the processes of each application sub-domain of size d11·d12 is linearly mapped to the end nodes of a switch.

- fat-tree-2L: the mapping should similarly be described by a tuple {d11,d12}, where d11·d12=m1, d11 divides D1 and d12 divides D2, meaning that the processes of each application sub-domain of size d11·d12 is linearly mapped to the end nodes of a level-1 switch.

- fat-tree-3L: the mapping should be described by a 4 values {d11,d12,d21,d22}, where d21·d22=m1, d11·d12=m1·m2, d21 divides d11, d22 divides d12, d11 divides D1 and d12 divides D2. In this case, the application domain is divided into sub-domains of size d11·d12 and each sub-domain into sub-sub-domains of size d21·d22. Each application sub-domain is mapped linearly to the end nodes of a level-2-switch-rooted sub-tree and each application sub-sub-domain is mapped linearly to the end nodes of a level-1 switch of the level-2-switch-rooted sub-tree.

- 2DhyperX: the application sub-domain is represented by a line in the application domain. The hardware sub-domain is represented by all the nodes connected to a set of m $\in$ {1,2, ... d1} consecutive switches in the X dimension of the network topology. To describe the mapping in the case of 2D Hyper-X topologies the user only has to define the ”type” parameter as ”linear”.

config:mappingDescription:d11 [-]: [Valid only for the 2-dimension nearest-neighbor pattern] This parameter needs to be defined for full-mesh, fat-tree-2L and fat-tree-3L topologies.
config:mappingDescription:d12 [-]: [Valid only for the 2-dimension nearest-neighbor pattern] This parameter needs to be defined for full-mesh, fat-tree-2L and fat-tree-3L topologies.
config:mappingDescription:d21 [-]: [Valid only for the 2-dimension nearest-neighbor pattern] This parameter needs to be defined fat-tree-3L topologies.
config:mappingDescription:d22 [-]: [Valid only for the 2-dimension nearest-neighbor pattern] This parameter needs to be defined fat-tree-3L topologies.

References

[1] R. Jongerius, *Exascale Computer System Design: The Square Kilometre Array.* PhD thesis, Eindhoven University of Technology, September 2016.

[2] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, “McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures,” in *42nd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO-42, pp. 469–480, IEEE, December 2009.

[3] S. Poddar, R. Jongerius, L. Fiorin, G. Mariani, G. Dittmann, A. Anghel, and H. Corporaal, “MeSAP: A fast analytic power model for DRAM memories,” in *Design, Automation & Test in Europe Conference*, DATE ’17, IEEE, 2017.

[4] S. Thoziyoor, N. Muralimanohar, J. H. Ahn, and N. P. Jouppi, “CACTI 5.1,” Tech. Rep. HPL-2008-20, HP Laboratories, April 2008.

[5] A. Anghel, L. M. Vasilescu, G. Mariani, R. Jongerius, and G. Dittmann, “An instrumentation approach for hardware-agnostic software characterization,” *International Journal of Parallel Programming*, pp. 1–25, March 2016.

[6] G. Mariani, A. Anghel, R. Jongerius, and G. Dittmann, “Scaling properties of parallel applications to exascale,” *International Journal of Parallel Programming*, pp. 1–28, April 2016.

[7] Graph 500, “Graph 500 benchmark.” `http://www.graph500.org/`.