University of Siena
Department of Engineering

Investigating High Performance Conjugate Gradient Benchmark using COTSon simulations

Presented to Prof. Roberto Giorgi
By AbdAllah Aly Saad

Abstract

“There is, of course, no best method for all problems because the goodness of a method depends to some extent upon the particular system to be solved [3]”

High Performance Conjugate Gradient (HPCG) benchmark is a new benchmark replacing High Performance Linpack (HPL), and it targets real applications performance to help designer making the right choices for designing computer systems, HPCG is using Conjugate gradient method which is CPU hungry, furthermore it adds data dependency distribution and communication through Message Passing Interface (MPI).

Table of Contents

Introduction

Although High Performance Linpack (HPL) benchmark is a very successful metric for high performance computing, it is not relevant to real application performance anymore as a proxy. However, it is not suggested to be eliminated for the historical importance and the high performance community value. But using HPL to design systems would lead to wrong design choices for real applications, or add unnecessary components or complexity to the system. Thus, designing a computer system to run HPL at 1 Exaflop may not be attractive for real applications [1].

HPCG is meant to be a new metric that has a stronger correlation to real application performance unlike HPL, therefore leading system designers and driving them in directions that will improve application performance for broader set of HPC applications [1].

Therefore, HPCG is an attractive option, As it contains a small collection of key computation and communication patterns used in many application. It is also large enough to be mathematically meaningful, and yet small enough to easily understand and use [1].

Real applications are applications governed by differential equations, which requires high bandwidth and low latency, and tend to access data using irregular patterns [1]. There are two program patterns identified in [1], Type 1 and Type 2, while HPL is written for Type 1 patterns, HPCG is written for Type 2 pattern. Bother types are explained in *table 1*.

Type 1	Type 2
• Adequate streaming memory systems • Very high floating-point computation rates • data access is predominantly unit stride and is mostly hidden by concurrently performing computation on previously retrieved data • Accelerators are extremely effective	• High bandwidth memory systems • Low latency • Lower computation-to-data-access ratios • Access memory irregularly • Fine-grain recursive computations

Table 1 does not compare both types but rather focuses on the distinct requirements for both, since both types could normally exist in one application. And so, a system which is designed to efficiently execute both patterns will generally run a broad mix of applications well, Based on that, designing a system that is capable of providing both patterns requirements is a balanced system [1].

HPL is a simple program factoring and solving a large dense system of linear equations using Gaussian

Elimination with partial pivoting, in such systems, the algorithm is dominated by dense matrix-matrix multiplication calculations and related kernels, therefor, HPL stresses only that kind of patterns which is Type 1, and is not capable of measuring type 2 patterns. As a result, HPL shows a distorted picture relative to the real application performance under the tested system [1].

In this project, HPCG is investigated in a performance related manner, this will be more investigated through simulations those providing different cluster configurations for each bench-marking experiment .

Simulations provide the ability to create virtual systems in which hardware components are shaped, for making new functional units or entire microprocessor system, or even clusters. By providing different architectures, Simulations have the aim to show, record and analyze the performance and the behavior of applications on each simulated system, allowing us to select the best architecture out of all the simulated systems [4].

Furthermore, simulators became the first choice for designing and analyzing computing systems due to the increasing complexity of such systems. The main characteristics of a simulator are : Speed, accuracy, full-system capability and ability to extract specific metrics making each simulator distinct from the others, therefore, a good simulator infrastructure can help researchers, system designers and developers to verify their decisions and to possibly find some optimal solutions, one of these simulators is COTSon [4].

COTSon is a simulation framework with the aim of providing an evaluation platform for real systems such as the current multi-core Personal Computers of x86_64 processors with all classical peripherals, and running available operating systems such as Linux or Windows™ . COTSon was originally developed by HP Labs and AMD to target cluster-level systems composed of hundreds or thousands of commodity multi-core nodes and their associated devices connected through a standard communication network, Next to the functional simulation, it provides timing behavior of the architectural component for more accurate evaluation of these systems [4].

COTSon also targets many-core architectures, however, our interest in COTSon is it's key feature, which is the adoption of a functional-directed simulation approach, where fast functional emulators and timing models cooperate to improve the simulation accuracy at a speed suffcient to simulate the full stack of applications, middleware and OS. Functional simulations emulate the behavior of the hardware components (e.g., common devices such as disks, video, and network interfaces) of the target system, without considering latency information. On the other hand, timing simulations are used to assess the performance of the system the operation latency of devices simulated by the functional simulator, assuring that events generated by these devices are simulated in a correct time ordering [4].
A COTSon node cosists of, the functional simulator SimNow, the timing models (timers), the sampler,

the inter-leaver, and the time predictor, but the most important components for our simulation next to SimNow are the network Mediator and the Control allowing to simulate cluster configurations [4].

SimNow is the virtualizer for COTSon, it implements the x86 and x86_64 instruction sets, including system devices, and allows the user to configure a full-system architecture bu modifying the various system components such as the CPU type, CPUs count, main memory size and others. By providing several CPU models, dynamic translation of instructions (the instruction input stream is translated into C-like language and then is compiled for the native machine) and deterministic execution, SimNow is able to simulate the majority of existing hardware uniprocessor and multiprocessor that are available on a modern AMD systems. SimNow also has a very important feature for our project, that is the use of caching techniques and supporting the booting of an unmodified Operating System over which some complex applications can be executed [4].

Background

There are two types of methods identified in, A hand method, and a machine method. Hand methods refers to the ability to use a simple desk calculator to solve the the problem, However, A machine method deploys sequence-controlled machines [3]. Machine method is further identified by the following properties [3] :

1. The method should be simple, composed of a repetition of elementary routines requiring a minimum of storage space.
2. The method should insure rapid convergence if the number of steps required for the solution is infinite.
3. The procedure should be stable with respect to rounding-off errors. If needed, a subroutine should be available to insure this stability. It should be possible to diminish rounding-off errors by a repetition of the same routine, starting with the previous result as the new estimate of the solution.

[3] states that that best methods to fit these criteria are (a) the Gauss elimination method; (b) the conjugate gradient method, However, it also states that the conjugate gradient method is superior to the elimination method as a machine method for the following reasons :

(a) Like the Gauss elimination method, the method of conjugate gradients gives the solution in n steps if no rounding-off error occurs.
(b) The conjugate gradient method is simpler to code and requires less storage space.
(c) The given matrix is unaltered during the process, so that a maximum of the original data is used
(d) At each step an estimate of the solution is given, which is an improvement over the one given in the preceding step.

(e) At any step one can start anew by a very simple device, keeping the estimate last obtained as the initial estimate.

Conjugate Gradient method is an iterative method used to solve a system of linear equations :

$$a_{11}x_1 + a_{12}x_2 + \ldots\ldots + a_{1n}x_n = k_1$$
$$a_{21}x_1 + a_{22}x_2 + \ldots\ldots + a_{2n}x_n = k_2$$
$$\ldots\ldots \quad \ldots\ldots\ldots \quad \ldots\ldots\ldots$$
$$a_{n1}x_1 + a_{n2}x_2 + \ldots\ldots + a_{nn}x_n = k_n$$

which is represented by the vector form :

$$\boldsymbol{Ax=k}$$

Where $\boldsymbol{A}$ is the matrix of coefficients (a_{ij}), $\boldsymbol{x}$ and $\boldsymbol{k}$ are the vectors ($x_1,\ldots,x_n$) and ($k_1,\ldots,k_n$). $\boldsymbol{A}$ is assumed to be symmetric and positive definite [3].

Conjugate Gradient method terminates in at most ***n*** steps if no rounding-off errors are encountered, However, the estimate $\boldsymbol{x_n}$ is in general not the solution ***h*** but a good approximation of ***h***. At the same time it is not advised to continue beyond **n** iterations but to start a new estimate to diminish the effects of rounding- off errors. A principal advantages of the method is the flexibility of the ability to start a new estimate at any step. The conjugate gradient method is defined as a set of steps as illustrated by *table 1* [3].

(x_0 arbitrary, Initial step)	$p_0 = r_0 = k - Ax_0$	Select an estimate x0 of h and com- pute the residual r0 and the direction p0 by formulas
General routine (iterative)	$a_i = \frac{\lvert r_i \rvert^2}{(p_i, Ap_i)}$ $x_{i+1} = x_i + a_i p_i$ $r_{i+1} = r_i + a_i A p_i$ $b_i = \frac{\lvert r_{i+1} \rvert^2}{\lvert r_i \rvert^2}$ $p_{i+1} = r_{i+1} + b_i p_i$	Having determined the estimate xt of h, the residual rt, and the direction pt, compute xi+1, r^+i, and pi+x by formulas (3:1b), . . ., (3:If) successively.
Solution (this is not defined as a step of the routine itself by [2] but rather included since it is the final goal of the whole method)	$x = \sum_{i=0}^{n-1} \frac{(p_i, k')}{(Ap_i, p_i)} p_i$	Once one has obtained the set of n mutually conjugate vectors p0, . . ., -pn-i the solution of $\boldsymbol{Ax=k'}$

Table 1 : The conjugate gradient method

Iterative solvers such as Conjugate Gradient suffer from a widely recognized weakness, that is the lack

of robustness, this is where precondition comes in handy. Preconditioning is not only a key ingredient for the success of Krylov subspace methods such as Conjugate Gradient, but is used also to improve both of the efficiency and and robustness of these iterative techniques [2].

Preconditioning is defined as a way to transform the original linear system into a new one that has the same solution, but is easier to solve with iterative methods [2]. For a linear system in the vector form :

$$\boldsymbol{Ax=b}$$

We find a preconditioner matrix $\boldsymbol{M}$ as a first step. In general, It is required that matrix $\boldsymbol{M}$ makes the new linear system inexpensive to solve, because $\boldsymbol{M}$ is used at each step of the iterative method. The preconditioner matrix $\boldsymbol{M}$ should be Close to $\boldsymbol{A}$, non-singular and non-expensive as it will be used in many iterations [2].

The preconditioner matrix $\boldsymbol{M}$ is a matrix which approximates $\boldsymbol{A}$ in some yet undefined sense, as M is also Symmetric and Positive Definite. There are three known ways for applying a preconditioner as shown in *table 2*, although it is noticed that the first two are no longer symmetric in general [2].

Left conditioner	$\boldsymbol{M^{-1}Ax = M^{-1}b}$	*The precondiotioner can be applied from the left, leading to the preconditioned system*
Right conditioner	$\boldsymbol{AM^{-1}u = b,\ x \equiv M^{-1}u}$	*Note that the above formulation amounts to making the change of variables u = M x, and solving the system with respect to the unknown u*
Factored form (split preconditioner)	$\boldsymbol{M = M_L M_R}$ $M_L^{-1} A M_R^{-1} u = M_L^{-1} b, x \equiv M_R^{-1} u$	Commonly used when the preconditioner is available in factored form. Where, $\boldsymbol{M_L}$ and $\boldsymbol{M_R}$ are triangular matrices, Allowing the split preconditioner form. This is a mandatory choice in case the matrix A is symmetric in order to preserve symmetry, never the less, there are other ways of preserving symmetry or rather taking advantage of this property, even if M is not available in a factored form.

Table 2 : The three known ways for applying a preconditioner.

In order to precondition the conjugate gradient, we have to learn more about the matrix $\boldsymbol{M}$, where we have two situations, 1) $\boldsymbol{M}$ is available in the form of incomplete Cholesky factorization, and 2) M is a Cholesky product :

$$\boldsymbol{M} = \boldsymbol{L}\boldsymbol{L}^{\mathrm{T}} ,$$

One important aspect is to preserve symmetry, a simple way is to use the Split Preconditioner (as shown in *table 1*) yielding the Symmetric Positive Definite matrix, therefor :

$$\boldsymbol{L}^{-1}\boldsymbol{A}\boldsymbol{L}^{-\mathrm{T}}\boldsymbol{u} = \boldsymbol{L}^{-1}\boldsymbol{b} , \qquad \boldsymbol{x} = \boldsymbol{L}^{-\mathrm{T}}\boldsymbol{u}.$$

And because $\boldsymbol{M}^{-1}\boldsymbol{A}$ is self adjoint for the $\boldsymbol{M}$ inner product :

$$(\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{x}, \boldsymbol{y})_{\boldsymbol{M}} = (\boldsymbol{A}\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}, \boldsymbol{A}\boldsymbol{y}) = (\boldsymbol{x}, \boldsymbol{M}(\boldsymbol{M}^{-1}\boldsymbol{A})\boldsymbol{y}) = (\boldsymbol{x}, \boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{y})_{\boldsymbol{M}}$$

Thus, Another way is to replace the usual Euclidean inner product in the Conjugate Gradient algorithm by the $\boldsymbol{M}$ inner product :

$$(\boldsymbol{x}, \boldsymbol{y})_{\boldsymbol{M}} \equiv (\boldsymbol{M}\boldsymbol{x}, \boldsymbol{y}) = (\boldsymbol{x}, \boldsymbol{M}\boldsymbol{y})$$

Using the above observations, the Preconditioned CG Algorithm is concluded in *table 3* :

Before omitting M inner-product	$a_j = \dfrac{(z_j, z_j)_M}{(M^{-1}Ap_j, p_j)}$ $x_{j+1} = x_j + a_j p_j$ $r_{j+1} = r_j - a_j A p_j$ *And* $z_{j+1} = M^{-1} r_{j+1}$ $b_j = \dfrac{(z_{j+1}, z_{j+1})_M}{(z_j, z_j)_M}$ $p_{j+1} = z_{j+1} + b_j p_j$	And since $\boldsymbol{r}_j = \boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}_j$ denotes the original residual, And $\boldsymbol{z}_j = \boldsymbol{M}^{-1}\boldsymbol{r}_j$ denotes the residual for the preconditioned system, Th CG method with out the initial step is re-written using the following :
Concluded Preconditioned CG Algorithm (where M is incomplete Cholesky)	*Compute* $r_0 := b - Ax_0$, $z_0 = M^{-1}r_0$, and $p_0 := z_0$ *For j=0, 1 ….. Until convergence Do :* $a_j = \dfrac{(r_j, z_j)}{(Ap_j, p_j)}$ $x_{j+1} = x_j + a_j p_j$ $r_{j+1} = r_j - a_j A p_j$ $z_{j+1} = M^{-1} r_{j+1}$ $b_j = \dfrac{(r_{j+1}, z_{j+1})}{(r_j, z_j)}$ $p_{j+1} = z_{j+1} + b_j p_j$	Observing that $(\boldsymbol{z}_j , \boldsymbol{z}_j)_M = (\boldsymbol{r}_j , \boldsymbol{z}_j)$ and $(\boldsymbol{M}^{-1}\boldsymbol{A}\boldsymbol{p}_j , \boldsymbol{p}_j)_M = (\boldsymbol{A}\boldsymbol{p}_j , \boldsymbol{p}_j)$, It is concluded that $\boldsymbol{M}$-inner products do not have to be computed explicitly.

Table 3 : Concluded Preconditioned CG Algorithm (where M is incomplete Cholesky)

There are two options when M is a Cholesky product, the split preconditioning option as shown in *table 4* below, or the above algorithm in *table 3*.

Defining variables for the new CG-Algorithm	$\hat{p}_j = L^T p_j$ $u_j = L^T x_j$ $\hat{r}_j = L^T z_j = L^{-1} r_j$ $\hat{A} = L^{-1} A L_{-T}$	Observe that $(r_j, z_j) = (r_j , L^{-T} L^{-1} r_j) =$ $(L^{-1} r_j , L^{-1} r_j) = (\hat{r}_j , \hat{r}_j)$. Similarly, $(Ap_j , p_j) =$ $(AL^{-T} \hat{p}_j , L^{-T} \hat{p}_j) (L^{-1} AL^{-T}$ $\hat{p}_j , \hat{p}_j) =$ $(\hat{A} \hat{p}_j , \hat{p}_j)$
CG-Algorithm using new defined variables	$a_j = \frac{(r_j, r_j)}{(\hat{A}\hat{p}_j, \hat{p}_j)}$ $u_{j+1} = u_j + a_j \hat{p}_j$ $\hat{r}_{j+1} = \hat{r}_j - a_j \hat{A} \hat{p}_j$ $b_j = \frac{(\hat{r}_{j+1}, \hat{r}_{j+1})}{(\hat{r}_j, \hat{r}_j)}$ $\hat{p}_{j+1} = \hat{r}_{j+1} + b_j \hat{p}_j$	the Conjugate Gradient algorithm applied to the preconditioned system $\hat{A}\ \boldsymbol{u} = \boldsymbol{L}^{-1}\boldsymbol{b}$ **.** Where $\boldsymbol{u} = \boldsymbol{L}^T\boldsymbol{x}$
Split Preconditioner Conjugate Gradient	*Compute* $r_0 := b - Ax_0$, $\hat{r}_{0\ 0} = L^{-1} r_0$, and $p_0 = L^{-T}\ \hat{r}_0$ *For j=0, 1 ….. Until convergence Do :* $a_j = \frac{(\hat{r}_j, \hat{r}_j)}{(Ap_j, p_j)}$ $x_{j+1} = x_j + a_j p_j$ $\hat{r}_{j+1} = \hat{r}_j - a_j L^{-1} A p_j$ $b_j = \frac{(\hat{r}_{j+1}, \hat{r}_{j+1})}{(\hat{r}_j, \hat{r}_j)}$ $p_{j+1} = z_{j+1} + b_j p_j$	Iterations are re-written using the $\boldsymbol{x}$ variable instead of the $\boldsymbol{u}$, since it is common in algorithms involving a right split preconditioner.

Table 4 : Split Preconditioner Conjugate Gradient

For the right preconditioned case, since the matrix $\boldsymbol{AM}^{-1}$ is not Hermitian with either the Standard inner product or the $\boldsymbol{M}$ -inner product. However, it is Hermitian with respect to the $\boldsymbol{M}^{-1}$ -inner product, the new CG-Algorithm is shown in *table 5***.**

CG-Algorithm using $\boldsymbol{M}^{-1}$ inner-product	$\alpha_j = (r j , r j) M^{-1} / (AM^{-1} p_j , p_j) M^{-1}$ $u_{j+1} = u_j + \alpha_j p_j$ $r_{j+1} = r_j - \alpha_j AM^{-1} p_j$ $\beta_j = (r_{j+1} , r_{j+1}) M^{-1} / (r_j , r_j) M^{-1}$ $p_{j+1} = r_{j+1} + \beta_j p_j$.	Re-writing CG-Algorithm with respect to the $\boldsymbol{u}$ variable and for the new inner product ($\boldsymbol{M}^{-1}$ inner poduct).

Concluded right-conditioned CG Algorithm	*Compute* $r_0 := b - Ax_0$, $z_0 = M^{-1}r_0$, and $q_0 := z_0$ *For j=0, 1 ….. Until convergence Do :* $a_j = \frac{(z_j, r_j)}{(Aq_j, q_j)}$ $x_{j+1} = x_j + a_j q_j$ $r_{j+1} = r_j - a_j A q_j$ $z_{j+1} = M^{-1} r_{j+1}$ $b_j = \frac{(z_{j+1}, r_{j+1})}{(z_j, r_j)}$ $q_{j+1} = z_{j+1} + b_j q_j$	Since u and x are related by $x = M^{-1} u$, the u vectors are not needed and the u update line is replaced by updating $\boldsymbol{x}_{j+1} = \boldsymbol{x}_j + \boldsymbol{\alpha}_j \boldsymbol{M}^{-1}\boldsymbol{p}_j$. And by using the terms $\boldsymbol{q}_j = \boldsymbol{M}^{-1}\boldsymbol{p}_j$ and $\boldsymbol{z}_j = \boldsymbol{M}^{-1}\boldsymbol{r}_j$

Table 5 : Concluded right-conditioned CG Algorithm

We notice that the CG algorithm discussed here is a basic, non optimized and primitive version of the algorithm which was introduced during early 1950s, However, we can see that the algorithm requires preserving memory for the auxiliary vectors for each step of *k* steps, where $k <= n$ in general , or $k > n$ if the algorithm didn't reach a satisfying solution in n steps.

The HPCG benchmark consists of 5 stages as defined in [1] :

- Problem Setup : In this step a synthetic symmetric positive definite (SPD) matrix ***A*** is generated by using the compressed sparse row format, also a corresponding vector ***b*** for the right hand side and an initial guess for ***x***.
    - building the new data structure is not considered in the benchmark timing but is observed and reported. It is also normalized by the cost of a matrix-vector multiplication operation using the original data structures.
    - It is not permitted to exploit the proprieties of the matrix such as regularity and being value-symmetric by storing it in a sparse diagonal format for reducing the storage requirements, instead, all matrix values will be stored in an unstructured manner.

- Preconditioner setup: setting up data structures for the local symmetric Gauss-Seidel preconditioner by using simple compressed sparse row representation for the lower and upper triangular matrices, each as a separate matrix.
    - This is step is also not considered in the effective benchmark timing, but only reported as part of performance investigation, And is normalized by the cost of one symmetric Gauss-Seidel sweep using the original matrix format.

- Verification and validation setup : after the first two steps of generating the matrix and vector and initializing the data structures, in this step, preconditions, post-conditions and in-variants that will aid in the detection of anomalies during the iteration phases are computed.

- Iteration : performs ***m*** iterations, ***n*** times, by using the same initial guess for each iteration,

where ***m*** and ***n*** are sufficiently large to test system uptime which is configured. By doing this we can compare the numerical results for “correctness” at the end of each m-iteration phase.

  - in case the result is not bit-wise identical across successive ***m***-iteration phases, the deviation is reported. Acceptable deviations (as determined in the V&V setup) will not invalidate the benchmark results. Instead they will alert for the lose of bit-wise reproducibility.
  - In order to report fair timing data for averaging, cache will be flushed between each of the k times the m iterations are performed

- Post-processing and reporting: a single timing result is reported among other metrics such as :

  - Computational verification and validation
  - Timing and execution rate results
  - Nodes count, total storage, processors, accelerators, precision used, compiler version, optimization level, compiler directives used, flop count, power used, cache effects, loads and stores, etc.

Project

We are going to use HPCG on a cluster. A cluster is a set of loosely coupled computers that work together as if they were a single computer [4].

One of the features of COTSon's architecture for our project, is that it has bee developed having in mind the simulation of clusters, that is possible by using a SimNow instance to represent each node of the cluster, and cluster nodes are interconnected through an Ethernet based network card and through a simulated switch (called mediator) by using an individual full-system instance of SimNow for each node. Mediator is a software component of COTSon's architecture which is responsible to manage the network communication among different nodes of the simulated system [4].

COTSon uses configuration files written in the Lua scripting language [4], and we will use one of the provided ready made configurations with a slight modification, follows is the configuration file content for our simulation :

```
runid="twonodes_noscreen"
tmpdir=os.getenv("PWD")

--copy_files_prefix="log."..os.time().."."
copy_files_prefix=runid.."."
debug=true
--to see the simnow windows, uncomment the following line
--display=os.getenv("DISPLAY")

control_script_file='twonodes_noscreen-ctrl'
```

```
--clean_sandbox=false
-- node_config="config.sh"

DBFILE="/tmp/twonodes_noscreen.db"
DESC="HPCG Cluster Silent (no screen) Simulation"
EXP=10

cluster_nodes=2
options = {
      sampler={ type="no_timing", quantum="100k" },
      heartbeat={ type="file_last" },
      heartbeat1={ type="sqlite",dbfile=DBFILE, experiment_id=EXP, experiment_description=DESC
},
      network_cpuid=true, -- send cpuid 'tracer' commands to all nodes
}

-- Poor man's multicast disambiguation (pid-based)
MCAST= (cotson_pid % 200) + 1
mediator = {
      verbose=1,
    multicast_ip="239.200.1."..MCAST,
    quantum_min=10,
    quantum_max=1000,
      heartbeat_interval=10000, -- 10msec
      heartbeat={ type="file_last" },
      heartbeat1={ type="sqlite", dbfile=DBFILE, experiment_id=EXP },
      timer={ type="simple", max_bandwidth=1000, latency=12, cfactor=10 },
      tracefile="/tmp/net-trace.gz", -- dump a trace
      slirp=false, -- don't NAT with the external world
}

simnow.commands=function()
      use_bsd('1p.bsd')
      use_hdd('karmic64.img')
      set_journal()
      set_network()
      execute('hpcg.sh')
end

function build()
      i=0
      while i < disks() do
            disk=get_disk(i)
            disk:timer{ name='disk'..i, type="simple_disk" }
            i=i+1
      end
```

```
    i=0
    while i < nics() do
        nic=get_nic(i)
        nic:timer{ name='nic'..i, type="simple_nic" }
        i=i+1
    end

    x=cpus()
    if x ~= 1 then
        error("This experiment only wants to handle 1 cpu")
    end

    cpu=get_cpu(0)
    cpu:timer{ name='cpu0', type="timer0" }
    mem=Memory{ name="main", latency=1 }
    cpu:instruction_cache(mem)
    cpu:data_cache(mem)
    cpu:instruction_tlb(mem)
    cpu:data_tlb(mem)
end
```

The main section in a COTSon's configuration file is the *simnow.commands*, where we define what BSD configuration to use in the line *use_bsd('1p.bsd')* , a BSD is a Broad-Sword Document, it contains setup parameters of the simulated target machine, such as the architecture, memory size and model, CPU count and model and other system components, and SimNow comes with different ready made BSD configurations [4].

The other important parameter is *cluster_nodes=2* , which tells the mediator how many nodes we have in the cluster, and therefore, assigning each node a unique network address and a hostname. Also *use_hdd('karmic64.img')* declares which disc image to use, in this case it is an Uubntu Karmic 64 biy with OpenMPI packages preinstalled. The script to be executed on each node is indicated by *execute('hpcg.sh')* , it is assumed to be in the same directory along with the COTSon configuration file and it is as follows :

```
if [[ $# < 2 ]]
then
    echo "Exepected 2 arguemtns, Node Number and Total Nodes Count"
    exit
fi

NODE=$1
```

```
TOTAL_NODES=$2

sed 's/PermitEmptyPasswords no/PermitEmptyPasswords yes/g' -i /etc/ssh/sshd_config
sed 's/UsePAM yes/UsePAM no/g' -i /etc/ssh/sshd_config
passwd -d root
/etc/init.d/ssh try-restart
MPIHOSTS=""
sleep 1s

for (( i=1; i <= TOTAL_NODES; i++ ))
do
    checknode="nNUM"
    checknode=${checknode/NUM/$i}
    if [[ $i == 1 ]]
    then
        MPIHOSTS="${MPIHOSTS}${checknode}"
    else
        MPIHOSTS="${MPIHOSTS},${checknode}"
    fi
    if [[ "$(hostname)" == "$checknode" ]]
    then
        continue
     fi
     ssh-keyscan -H $checknode >> ~/.ssh/known_hosts
done

echo 1 > /var/log/hpcgready
echo "node $NODE is READY"

#MPIHOSTS=""
READY_COUNTER=0

for (( i=1; i <= TOTAL_NODES; i++ ))
do
    checknode="nNUM"
    checknode=${checknode/NUM/$i}
    ((READY_COUNTER++))
    if [[ "$(hostname)" == "$checknode" ]]
    then
```

```
            continue
        fi
        ssh $checknode 'cat /var/log/hpcgready'
        if [[ $? == 0 ]]
        then
            echo "$checknode is also ready :D"
        fi


        if [[ $i == $TOTAL_NODES && $READY_COUNTER != $TOTAL_NODES ]]
        then
            echo "not all nodes are ready"
            READY_COUNTER=0
            i=1
        fi
done

echo "HOSTS ARE : $MPIHOSTS"
echo "all nodes are ready"
#ls HPCG-master
cd HPCG-master/My_MPI_OMP_BUILD/bin
export OMP_NUM_THREADS=1
#sed 's/60/3700/g' -i hpcg.dat
sed 's/60/120/g' -i hpcg.dat

if [[ "$1" == "1" ]]
then
        echo "at Node Master"
        echo "Running : mpirun -np $2 -host "$MPIHOSTS" xhpcg &2>1 mpioutput"
        echo "Running : mpirun -np $2 -host "$MPIHOSTS" xhpcg &2>1 mpioutput"  > runningMaster
        xput runningMaster /home/abdallah/hpcgbench2/$2/runningMaster
        mpirun -np $2 -host "$MPIHOSTS" xhpcg
        if [[ $? != 0 ]]
        then
            BROKEN_NODE="$(grep -o "n[1-9]" /home/root/screen.log)"
            ssh $BROKEN_NODE 'cat /var/log/hpcgready'
             if [[ $? == 0 ]]
            then
                    echo "$BROKEN_NODE is DOWN :("
```

```
        fi
    fi
    #mpirun -host n1,n2 xhpcg
    echo "output of mpitoutput is :"
    cat mpioutput
    xput mpioutput /home/abdallah/mpioutput
    echo "Done on Node 1 !"
    #for file in *.txt
    #do
    #    echo "xput $(pwd)/$file /home/abdallah/logs/$TOTAL_NODES"
    #done
    for file in *
    do
        xput $file /home/abdallah/hpcgbench2/$2/$file
    done
    xput /home/root/stdout.log /home/abdallah/hpcgbench2/$2/stdout.log
    echo 1 > /var/log/hpcgdone
fi

HPCHRUNNING=true
while [[ $HPCHRUNNING && $1 != 1 ]]
do
    echo "Node $1 is waiting $(date)" > waiting$1
    xput waiting$1 /home/abdallah/hpcgbench2/$2/waiting$1

    sleep 1s
    ssh n1 'exit'
     if [[ $? != 0 ]]
     then
          HPCHRUNNING=false
     fi

done

     echo "Node $1 is done $(date)" > done$1
     xput done$1 /home/abdallah/hpcgbench2/$2/done$1

xput /home/root/screen.log "/home/abdallah/hpcgbench/$2/screen.log$1"
```

This script is uploaded to the guest machine, which is the simulated machine, from the host machine, which runs the COTSon, through the *execute('hpcg.sh')* command in the COTSon's configuration file, The script is called with two parameters, the total number of nodes on the cluster and the current node number. The guest script will then perform system configuration to enable MPI application execution on the cluster. Then the script makes sure all nodes on the cluster are up and running, and are online.

And the execution phase of the script is to check for the current node number, if node is numbered 1, so it is the Master node and it is the node which starts the HPCG benchmark execution through OpenMPI's ***mpiexec*** command, and passes the number of nodes and their host-names on the network to distributed the binary file and start the communication world. Meanwhile, all other nodes have to keep checking for the master node through a busy loop, otherwise the script will just exist and the simulation would be interrupted, at the same time.

Once the master node has finished the execution of the benchmark, results are obtained at the master node, and then they are copied to the host machine, this is done by using a communcation tool box provided by COTSon , it is called ***Xtools***, which is mainly two programs, ***xput*** which runs on the guest machine to copy a file from the guest machine to the host machine, secondly, the ***xget***, which copies a file from the host machine to the guest machine.

The produced report by the HPCG benchmark looks like the similar :

```
HPCG-Benchmark
2.1
HPCG Benchmark: Version 2.1 January 31, 2014
Machine Summary:
  Distributed Processes: 4
  Threads per processes: 1
Global Problem Dimensions:
  Global nx: 208
  Global ny: 208
  Global nz: 104
Processor Dimensions:
  npx: 2
  npy: 2
  npz: 1
Local Domain Dimensions:
  nx: 104
  ny: 104
  nz: 104
********** Problem Summary  **********:
Linear System Information:
  Number of Equations: 4499456
  Number of Nonzero Terms: 119934040
Multigrid Information:
  Number of coarse grid levels: 3
  Coarse Grids:
```

```
    Grid Level: 1
    Number of Equations: 562432
    Number of Nonzero Terms: 14799400
    Number of Presmoother Steps: 1
    Number of Presmoother Steps: 1
    Grid Level: 2
    Number of Equations: 70304
    Number of Nonzero Terms: 1802416
    Number of Presmoother Steps: 1
    Number of Presmoother Steps: 1
    Grid Level: 3
    Number of Equations: 8788
    Number of Nonzero Terms: 213712
    Number of Presmoother Steps: 1
    Number of Presmoother Steps: 1
********** Validation Testing Summary  **********:
Spectral Convergence Tests:
  Result: PASSED
  Unpreconditioned:
    Maximum iteration count: 11
    Expected iteration count: 12
  Preconditioned:
    Maximum iteration count: 2
    Expected iteration count: 2
Departure from Symmetry |x'Ay-y'Ax|/(||x||*||A||*||y||+||y||*||A||*||x||)/epsilon:
  Result: PASSED
  Departure for SpMV: 1.88472e-08
  Departure for MG: 1.72003e-08
********** Iterations Summary  **********:
Iteration Count Information:
  Result: PASSED
  Number of CG sets: 4
  Average iterations per set: 50
  Total number of iterations: 200
  Reference CG iterations per set: 50
********** Reproducibility Summary  **********:
Reproducibility Information:
  Result: PASSED
  Scaled residual mean: 3.82521e-05
  Scaled residual variance: 0
********** Performance Summary (times in sec) **********:
Benchmark Time Summary:
  Optimization phase: 0
  DDOT: 1.81261
  WAXPBY: 1.51656
  SpMV: 17.4823
```

```
  MG: 96.4276
  Total: 117.244
Floating Point Operations Summary:
  DDOT: 5.39935e+09
  WAXPBY: 5.39935e+09
  SpMV: 4.79736e+10
  MG: 2.73243e+11
  Total: 3.32015e+11
GFLOP/s Summary:
  DDOT: 2.97877
  WAXPBY: 3.56026
  SpMV: 2.74412
  MG: 2.83366
  Total: 2.83182
  Total with Optimization phase overhead: 2.83182
User Optimization Overheads:
  Optimization phase time (sec): 0
  Optimization phase time vs reference SpMV+MG time: 0
DDOT Timing Variations:
  Min DDOT MPI_Allreduce time: 0.273185
  Max DDOT MPI_Allreduce time: 1.68821
  Avg DDOT MPI_Allreduce time: 1.1707
********** Final Summary **********:
  HPCG result is VALID with a GFLOP/s rating of: 2.83182
  Reference version of ComputeDotProduct used: Performance results are most likely suboptimal
  Reference version of ComputeSPMV used: Performance results are most likely suboptimal
  Reference version of ComputeMG used: Performance results are most likely suboptimal
  Reference version of ComputeWAXPBY used: Performance results are most likely suboptimal
  Results are valid but execution time (sec) is: 117.244
  Official results execution time (sec) must be at least: 3600
```

The report contains information on the problem size (grid size) which here is 104x104x104 (defined in the benchmark configuration file ***hpcg,dat***), Distributed Processes, which is the count of the MPI processes executing the benchmark (in our case number of nodes too), Threads per processes is greater than one in case if using OMP, other detailed information such as each major method timing, it is noticed that all reported timing is in wallclock seconds and not cpu seconds, since HPCG is using OpemMPI's ***MPI_Wtime()*** function.

Finally to run the simulation we need to write a bash script to launch different simulations with different cluster size sequentially, and the script is as follows :

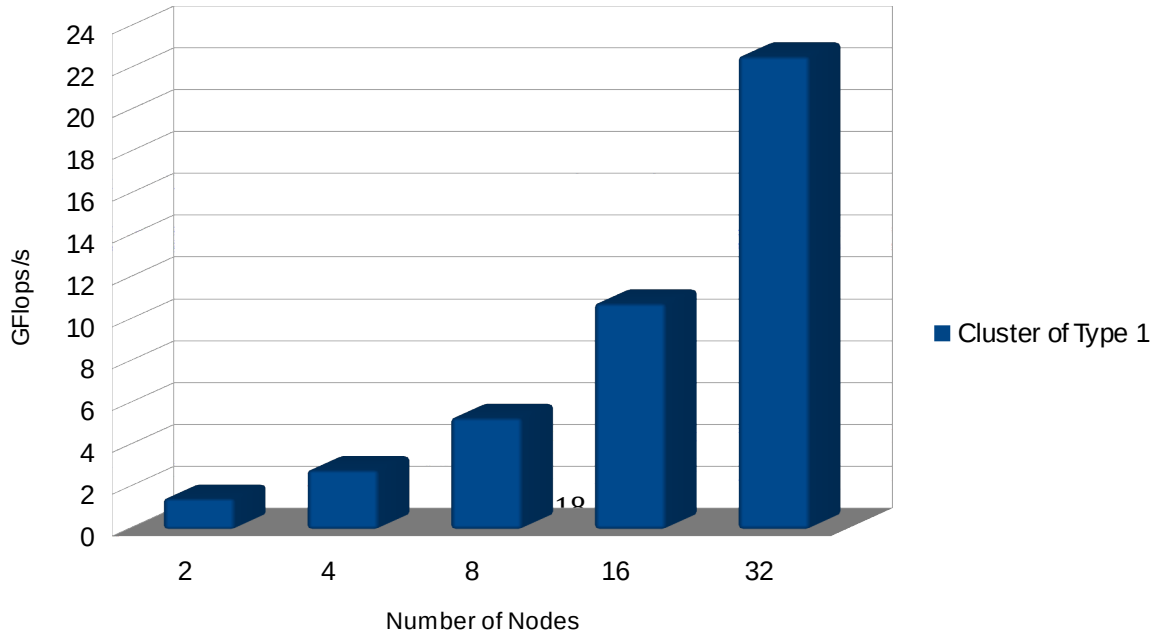
```
rm /home/abdallah/benchstatus
#rm -r /home/abdallah/hpcgbench
for (( i=4; i <= 64; i=i*2 ))
do
      echo "NOW  running hpcg with $i nodes $(date)"  >> /home/abdallah/benchstatus
```

```
    mkdir -p /home/abdallah/hpcgbench/$i/
    cp twonodes_noscreen.in "twonodes_noscreen.in$i"
    sed "s/cluster_nodes=2/cluster_nodes=$i/g" -i "twonodes_noscreen.in$i"

    /bin/rm -f run_twonodes_noscreen.log /tmp/twonodes_noscreen.db; \
    ( (sleep 10; if [ -s ./twonodes_noscreen-ctrl ]; then echo "Firing console
view"; ./twonodes_noscreen-ctrl;  fi) &); \
    ../../bin/cotson "twonodes_noscreen.in$i" > "run_twonodes_noscreen.log$i" 2>&1 ; \
    if [[ $? != 0 ]]
    then
        echo "DONE running hpcg with $i nodes  $(date)" >> /home/abdallah/benchstatus
    else
        echo "DONE With Error running hpcg with $i nodes  $(date)" \
>> /home/abdallah/benchstatus
    fi

    if [ -s /tmp/net-trace.gz ]; then echo "Dumping the network trace"; \
    ../tools/traceplayer /tmp/net-trace.gz 0 1; fi

    echo "DONE running hpcg with $i nodes  $(date)" >> /home/abdallah/benchstatus

done
```

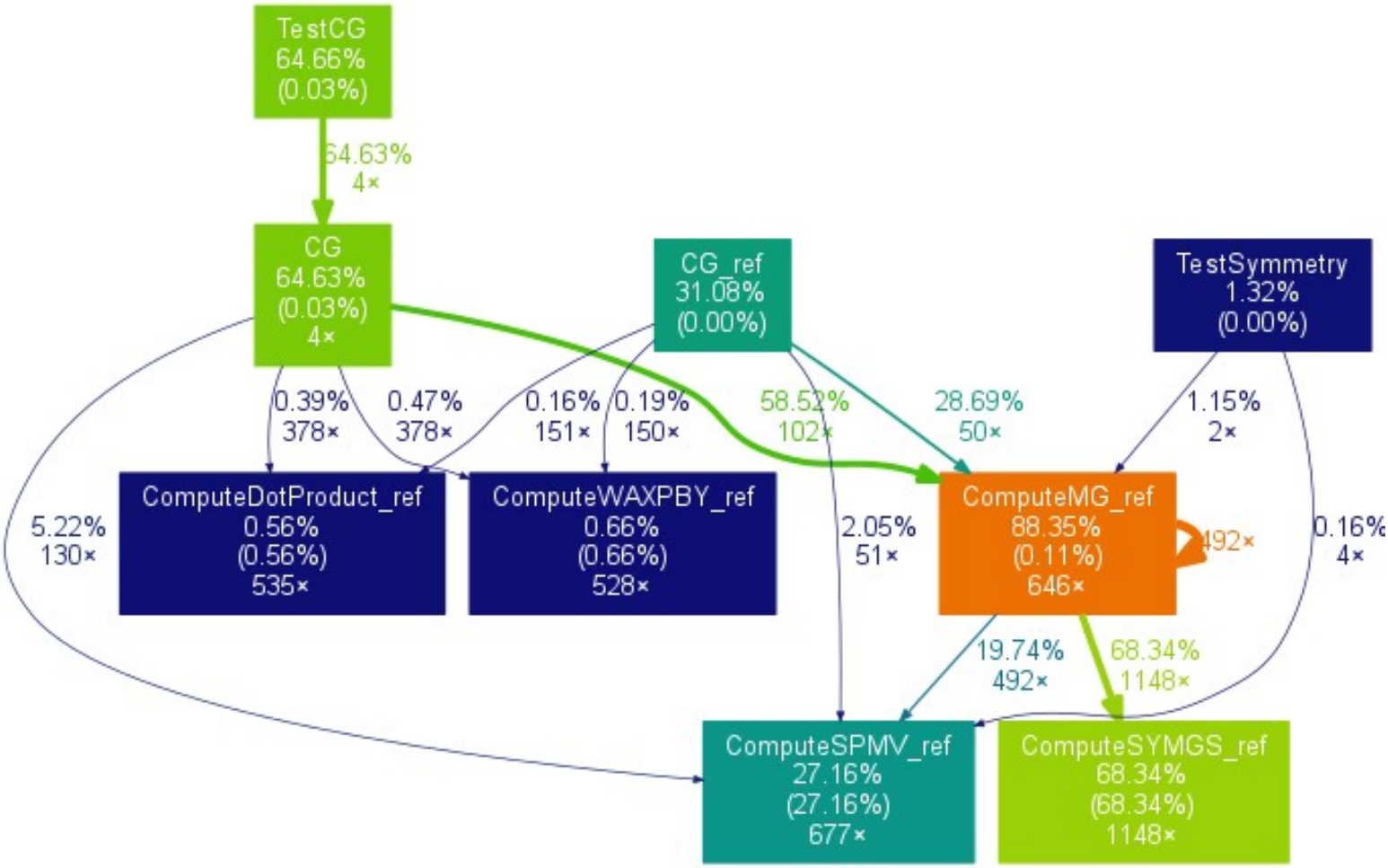
This bash script essentially makes a copy of the COTSon's cluster configuration and changes the number of nodes to match the current simulation cluster size, and then runs the simulation using the new configuration file.

Results

Shown in the graph below, and by using different cluster sizes (2, 4, 8, 16, 32), it is noticed that the performance in Gflops is almost doubled, realtion ship between cluster nodes number and performance increase is consistent, where Cluster of Type 1 is a homogeneous cluster with each node configured with 1 gigabye RAM and a single CPU of AMD uniprocessor.



Benchmark ran for 60 seconds on each node with problem size of 108*108*108, timing is reported using OpnMPI's wallclock function, and the results chart shows that each run performance is almost doubled measured in Gigaflops per second.



The Callgraph above shows the most significant functions and their calling percentage, it is also noticed that both ComputeSPMV (Sparse matrix vector multiplication) and ComputeSYMGS (Symmetric Gauss-Seidel precoditioner) take the greatest share of the whole process time and calls count, therefore focusing and optimizing both functions behavior and performance would result in better performance.

Conclusion

Results report, which is the final stage of the benchmark seems to be enough for computation and execution, however it is not enough if we want to investigate architecture specific bottlenecks or other environment settings and configurations, such detailed information is up to the bench-marker to investigate using other tools.

It is also noticed that performance is doubled as cluster nodes are , and is directly related to the cluster size, but at the same, the problem size is a direct factor of memory allocation and would be a bottleneck between the CPU and memory accesses as a lot of ***P*** vectors (as explained in the background section)

are kept in memory to be used later for solving the system.

References

(1) Jack Dongarra, Michael A. Heroux, “Toward a New Metric for Ranking High Performance Computing Systems ,” Sandia National Laboratories, June 2013.
(2) Yousef Saad, IterativeMethods for Sparse Linear Systems, second edition, Society for Industrial and Applied Mathematics; 2 edition (April 30, 2003)
(3) Magnus R. Hestenes, Eduard Stiefel, “Methods of Conjugate Gradients for Solving Linear Systems ,” Journal of Research of the National Bureau of Standards, Vol. 49, No. 6, December, Research Paper 2379, 1952.
(4) Roberto Giorgi, et al.. , “COTSON USER GUIDE ,” 4th edition, october 2014.