On the Use of a Genetic Algorithm in High Performance Computer Benchmark Tuning

Dominic Dunlop, Sebastien Varrette and Pascal Bouvry
Faculty of Sciences, Technology and Communication,
University of Luxembourg, LUXEMBOURG
<**firstname.name@uni.lu**>

Keywords: HPC, Linpack, Benchmark, Genetic Algorithm, Acovea, Performance evaluation.

Abstract

The High-Performance Linpack (HPL) [14] package is a reference benchmark used worldwide to evaluate high-performance computing platforms. Adjustment of HPL's seventeen tuning parameters to achieve maximum performance is a time-consuming task that must be performed by hand. In this paper, we show how a genetic algorithm may be exploited to automatically determine the best parameters possible to maximize the future results of the benchmark. Indeed we propose a GA based approach, even if we do not really specify a particular GA as our investigation relies on the Acovea framework [11], which managed repeated runs of the benchmark to explore the very large space of parameter combinations on the test-case cluster. This work opens the possibility of creating a fully-automatic benchmark tuning tool.

1. INTRODUCTION

With the configuration of a new computing cluster, generally comes a desire to evaluate its performance against other similar platforms, typically by measuring its floating-point computing power. This desire led the University of Luxembourg to conduct a benchmark evaluation of a recently-acquired Beowulf cluster *i.e.* a scalable-performance cluster based on commodity hardware communicating over a private system network, using open source software. The High-Performance LINPACK (HPL) [14] is a software package that solves a (random) dense linear system using double-precision (64 bit) floating-point arithmetic on distributed-memory computers. It can be regarded as a portable and freely-available implementation of the High Performance Computing LINPACK reference benchmark. The Top500 project [1], which ranks and details the 500 most powerful publicly-known computer systems in the world, continues to rely on HPL, despite the development of alternative packages [6]. The adjustment of HPL's seventeen tuning parameters to achieve maximum performance is a time-consuming task usually performed by hand. In this paper, we show how a genetic algorithm approach can be exploited to automatically determine the best parameters possible in order to maximize the future results of the benchmark. While some independent literature exists on the subject of tuning HPL's parameters – see [4] and [17] for instance, the application of a genetic algorithm to the problem has not, to our knowledge, previously been reported.

This article is organized as follows: §2. presents the HPL benchmark together with the general methodology used throughout the experiments described in this paper. In particular, the Acovea framework is described in §2.1. as an implementation of a genetic algorithm adapted to HPL evaluation (assuming the wrapping proposed in §2.2.). §3. outlines the experiments conducted, first with hand-tuning (§3.1.), then with Acovea (see §3.2.). Further improvements are explored in §3.3. Finally, §4. provides some conclusions and perspectives.

2. BENCHMARK DESCRIPTION AND TUNING METHODOLOGY

Below is a description of the HPL's Main Algorithm [14]

> This software package solves a linear system of order n: $A \times x = b$ by first computing the LU factorization with row partial pivoting of the N-by-$N+1$ coefficient matrix $[Ab] = [[L,U]y]$. Since the lower triangular factor L is applied to b as the factorization progresses, the solution x is obtained by solving the upper triangular system $U \times x = y$. The lower triangular matrix L is left unpivoted and the array of pivots is not returned.
>
> The data is distributed onto a two-dimensional P-by-Q grid of processes according to the block-cyclic scheme to ensure "good" load balance as well as the scalability of the algorithm. The N-by-$N+1$ coefficient matrix is first logically partitioned into NB-by-NB blocks (called *tiles*), that are cyclically "dealt" onto the P-by-Q process grid. This is done in both dimensions of the matrix.
>
> The right-looking variant has been chosen for the main loop of the LU factorization. This means that at each iteration of the loop a panel of NB columns is factorized, and the trailing submatrix is updated. Note that this computation is thus logically partitioned with the same block size NB that was used for the data distribution.

The main factors that directly affect the benchmark performance describe the way the tiles are defined and distributed, *i.e.*, the problem size, N, along with NB, P and Q. The underlying Linpack alternates between computation and communication when solving a system of equations. The work to be done in the computation phases depends on the number and size of the tiles to be solved while P and Q affect how the workload is balanced across the system, and how efficiently each tile can be processed. As problem sizes increase, the amount of computation grows faster than the cost of communication, and the impact of communication performance diminishes accordingly. To really understand how those four parameters are linked, a sample data distribution corresponding to the HPL parameters $N = 128$, $NB = 32$, $P = 3$ and $Q = 1$ is shown in Figure 1. Each tile in the matrix is therefore 32-by-32 in size and there are sixteen tiles to be cyclically distributed across $P \times Q = 3$ processes as illustrated.

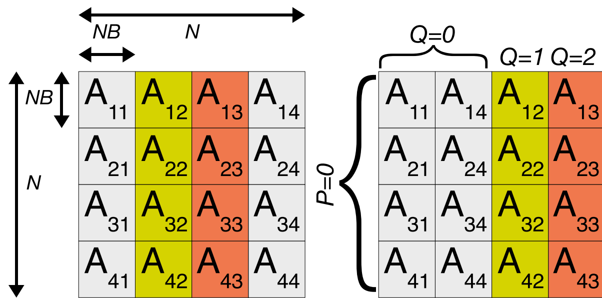

Figure 1. $P \times Q$ grid of processes for good load balancing [14].

In addition to its main parameters, HPL provides thirteen more that could also be significant. They are detailed in Table 5. The parameters for a benchmark run are defined in a configuration file (*HPL.dat* by default). Each run of the benchmark may test several parameter combinations, reporting on the execution time required to solve the problem for each parameter combination and the corresponding processing speed in billions of floating-point operations per second (Gflops). It is possible to determine some constraints on the parameters directly. For instance, $P \times Q$ is generally equal to the number of processing cores. As for N, HPL's authors suggest a simple dimensional analysis based on the cluster's volatile memory:

$$N \simeq 0.8\sqrt{\text{Total Memory Size in bytes} \times \frac{\texttt{sizeof(double)}}{8}}$$

In practice, this factor of 0.8 is hard to achieve, as memory paging strongly degrades performance once the factor value of 0.5 is exceeded. (See §3.2.1.).

Precisely tuning all seventeen parameters requires several HPL benchmark runs with optimization performed by hand between each. For the work described in this paper, hand-tuning was done in the initial experimentation described in §3.1. While good results can often be obtained by adjusting parameters by hand, there is always the suspicion that there is some other combination that would give better results still. The problem of exploring a large parameter space in order to optimize for a particular goal — in this case, maximizing floating point operations per second — is well-suited to the use of a genetic algorithm (GA hereafter). The theory and implementation of evolutionary algorithms are outside the scope of this paper — for comprehensive references, see [8, 9]. In this paper, the approach is validated through the Acovea [11] framework, which implements a GA. It was applied to see if it could discover parameter combinations that produced results that equalled or bettered those achieved by hand. The experiments conducted over this framework are detailed in §3.2. and 3.3. The next section details Acovea, together with the methodology used to adapt it to HPL configuration.

2.1. The Acovea framework

ACOVEA stands for *Analysis of Compiler Options via Evolutionary Algorithm*. The purpose of the open-source package is to investigate the optimum combination of command-line flags for a compiler, given a particular goal for the compiled program. Typical goals are to maximize execution speed or to minimize object code size. In the case of the GNU Compiler Collection[15], the C compiler has more than 60 different flags that affect code optimization and generation for a particular processor architecture. It is clearly impossible to explore all possible combinations exhaustively. Acovea uses a GA, provided by the underlying Evoscosm library[12], to heuristically evolve the set of flags that best optimizes for the chosen goal. In practice, the *runacovea* harness reads possible flags and their allowed values from an XML configuration file. It repeatedly compiles a test program, passing the compiler a different set of command-line flags each time, and then, for successful compiles, runs the resulting program. The program passes back a metric (typically a run-time), which *runacovea* uses to assess the effect of the flags. Figure 2 illustrates this process.

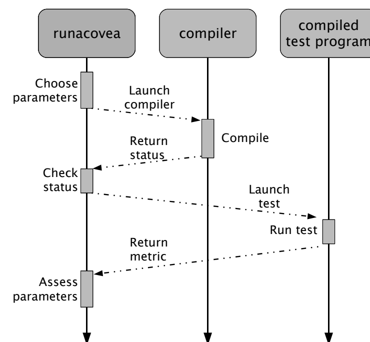

Figure 2. Acovea compiler flag testing process (single test).

Runacovea's parameters may be varied, but default values were used for the work described in this paper. These evaluate twenty generations, each having five populations of twenty individuals (HPL benchmarks in the current application), for a total of 4,000 individuals. These numbers are low compared to those used in many applications of GAs, and were probably chosen because the compilation step needed to evaluate the objective function for each individual is time-consuming. High-performance benchmarking is similarly expensive. Each individual is represented by vectors of bit flags corresponding to values of command-line parameters, and is randomly initialized while making sure that each possibility for multi-valued (as opposed to continuously-variable) parameters is adequately represented. After each generation, the fitness of each individual (in our case, the core HPL algorithm run-time) is examined. The GA used in this paper is configured with a crossover rate of 1 and a mutation rate of 0.01. Additionally, an elitism strategy is applied with 5% of the best individuals passing unchanged to the next generation.

When complete, *runacovea* reports the most successful command line, and summarizes the effects of flags which have statistically-significant good or bad effects. In particular, the *best-of-best* combination is provided (*i.e* the set of parameters resulting in the best execution time).

2.2. Adapting Acovea for use with HPL

Although Acovea is specifically targeted at the manipulation of compiler flags, varying HPL parameters is a similar task. For the sake of simplicity, it was decided to create a wrapper for HPL adapted to the model expected by Acovea instead of building a new application-specific tool based on the Evoscosm library. This wrapper behaves like a compiler by accepting command-line flags. It converts those command-line arguments to the corresponding configuration file *HPL.dat* and produces a second program that Acovea can run in order to obtain a performance metric. This second program launches a single benchmark run using *mpirun* to activate the Message Passing Interface of the execution platform, passing back to *runacovea* the run-time extracted from HPL's output. This process is schematically illustrated in Figure 3.

3. EXPERIMENTS

The experiments described in this article were conducted on a Beowulf cluster located at the University of Luxembourg. This cluster is composed of a front-end server, 18 computing nodes and a NFS file server. The computing elements are of two kinds: *small* nodes having a single dual-core processor and *large* nodes with four dual-core processors. Table 4 gives further details of the hardware configuration, together with the list of the installed software relevant for this work. The nodes of the cluster are connected through a gigabit Ethernet switch. The Maximum Transmission Unit (MTU)

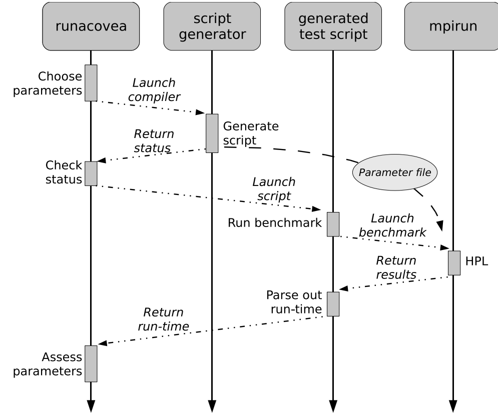

Figure 3. Acovea HPL benchmarking process.

is configured to be 1500 bits. In particular, jumbo frames are not enabled.

The source code for HPL was downloaded and built using one of the supplied sample Makefiles, namely `Make.Linux_PII_CBLAS`, and the system's default C compiler to produce a 32-bit executable, *xhpl*. The executable linked to the dynamic ATLAS and CBLAS libraries. CBLAS is a C interface to BLAS routines, a collection of basic linear algebra subprograms [5]. ATLAS (Automatically Tuned Linearly Algebra Software) [16] is used to automatically obtain the parameters for efficient linear algebra computations. These two libraries take advantage of the SSE2 (Streaming Single instruction, multiple data Extension, version 2) floating-point acceleration available from the cluster's Xeon and Pentium M processors [10]. The *xhpl* executable accessed the MPI middleware interface of the cluster through the OpenMPI library [7]. No attempt was made to improve the performance of the binary code for the application or the libraries, for example by recompiling with non-default compiler options or by creating 64-bit versions rather than the default 32[1]; the benchmarking exercise described in this paper concentrates solely on optimizing performance by varying the parameters passed to the *xhpl* application.

3.1. First experiments based on hand-tuning

The initial hand-tuning on the cluster gave the results shown in Table 1. The best result achieved was 84 Gflops, that is 41.5% of the theoretical maximum, by utilizing all the small nodes. Several observations can be made on the basis of the initial tests:

- System performance metrics captured by Ganglia [13]

[1] Intel's 64-bit processor model and parameter-passing conventions improve on the older 32-bit versions, and can result in generated code that runs faster, despite pointer length being doubled.

Nodes (cores)	N	NB	P	Q	Gflops		Effi-ciency
					Theo.	Prac.	
Small (32)	8000	80	4	8	204.8	33	16.1%
Large (16)	20000	80	4	4	108.8	25	23.0%
Small (32)	20000	80	4	8	204.8	64	31.2%
Small (32)	64000	80	4	8	204.8	**84**	**41.5%**

Table 1. Results of initial "hand" testing.

showed that the most successful runs minimized network traffic and maximized CPU user-mode cycles.

- The final small node test with $N = 64000$ used about half the available memory on each node, suggesting that N could not be increased much beyond this value without incurring performance-degrading paging (see §3.2.1.).
- Results (not shown) of tests that attempted to use both the large and the small nodes were unimpressive. This issue is explored in §3.2.3.

3.2. Experiments based on GAs

The tests were run in two phases. First, Acovea was used to establish optimum HPL parameters for a small problem size, as several runs of the benchmark are required per generation and the total execution time should be reasonable. In the second step, the established parameters were used to establish cluster performance for increasing problem sizes. Note that this strategy assumes that the parameters that give best results with the small problem sizes used while running the genetic algorithm will continue to give optimal results when the problem size is considerably increased. This assumption was later verified in the specific case of the *NB* parameter (see §3.3.), but has not been verified in the general case. This experiment ran on the cluster for three computing nodes configuration: first using only the *small* nodes (32 cores); second, with only the *large* nodes (16 cores); and finally with *all* available nodes *i.e* for a total of 48 cores.

Nodes (cores)	N	NB	P	Q	Gflops		Effi-ciency
					Theo.	Prac.	
Small (32)	84000	80	4	8	204.8	91	44.4%
All (48)	84000	80	2	24	313.6	77	24.6%
Large (16)	40000	80	2	8	108.8	25	23.0%

Table 2. Summary of the best results obtained with Acovea in the three computing nodes configurations.

The peak results are summarized in table 2 (see table 5 for the HPL parameters used in the first phase to attain them). The default values generally correspond to those proposed as reasonable starting points for tuning by the HPL documentation. Experience with each computing node configuration is discussed in the following subsections.

3.2.1. Tuning using small nodes only

Acovea was used to explore HPL's parameter space using a problem size of 12,000. This problem size was chosen as a compromise between the need to keep run-time for each test low, and the desire to obtain results that would extrapolate to large problems. As illustrated in Figure 4, the initial generation of tests had an average run-time of 33.3564 seconds, improving to 23.0882 seconds after twenty generations (4,000 tests of differing parameter combinations), with a *best-of-best* result of 22.59 seconds.

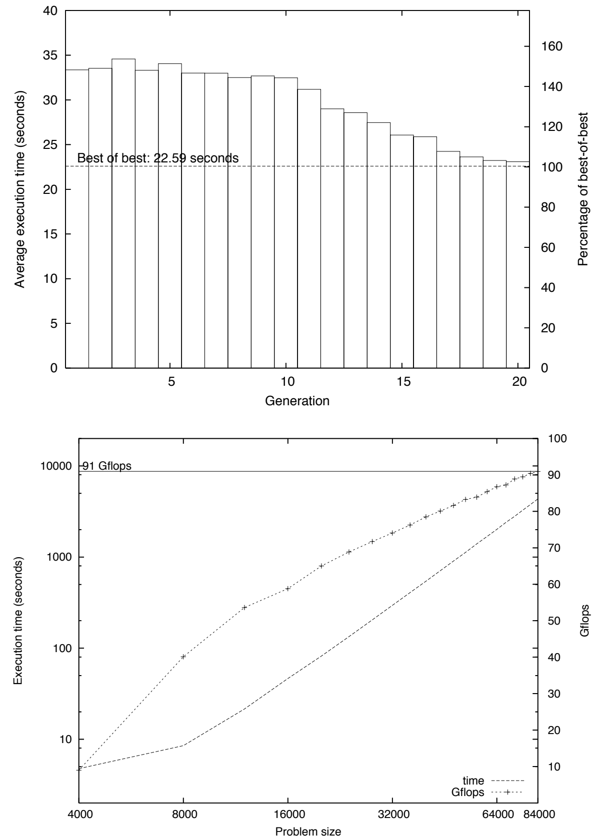

Figure 4. Acovea and HPL evaluation (using small nodes only).

In the second test phase, the parameters previously obtained were used to determine performance for problem sizes ranging from 4,000 to 88,000. As figure 4 shows, a peak of 90.98 Gflops was achieved with a problem size of 84,000. No result was obtained for the final problem size, as it required more memory than each node could offer, and consequently ran very slowly due to heavy paging. The benchmark figure obtained with the parameters suggested by Acovea is 5% better for the $N = 64,000$ case than that initially obtained by hand-tuning. An examination of table 5 shows that Acovea suggested that four parameters – $NBMIN$, $DEPTH$, $SWAPPING$ and U – should be changed from their default values. No parameter was reported as responsible for a

statistically-significant worsening of fitness by Acovea. Monitoring during the benchmark run, shown in figure 5, revealed that user-mode CPU utilization on each node was generally above 90%, dropping to around 80% at the end phase of each benchmark run. Increasing performance with larger problem sizes may be explained by the fact that the end phase accounts for a smaller proportion of run-time as problem size grows. The figure also shows that network traffic is strongly correlated with system-mode CPU time, and is an order of magnitude below that which would saturate the cluster's gigabit Ethernet interconnect. It is possible that performance figures could be slightly improved if network traffic were less expensive in terms of system CPU cycles. This might be achieved by enabling the use of "jumbo frames", so cutting the number of packets required to transfer a given volume of data. Further studies (not shown) suggested that most of the packets on the interconnect were frames of maximum length.

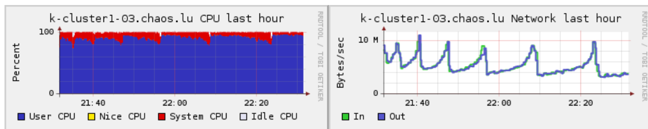

Figure 5. CPU and network use (using small nodes only).

Figure 4 shows that performance is still rising as the problem size reaches 84,000. Testing could not usefully proceed beyond this point because, as figure 6 shows, memory requirements on each node rose from just under 4GB in the $N = 84000$ case to around 7GB for $N = 88000$. As each node has 4GB of physical memory, heavy paging occurred, cutting the amount of time that each CPU spent in user mode working on the problem. It is almost certain that a better benchmark score could be obtained if per-node memory were increased. A rough linear extrapolation suggests that equipping each node with its maximum of 8GB of RAM would allow the $N = 88000$ case to run without paging, taking one hour twenty minutes of run-time and attaining 91.5 Gflops.

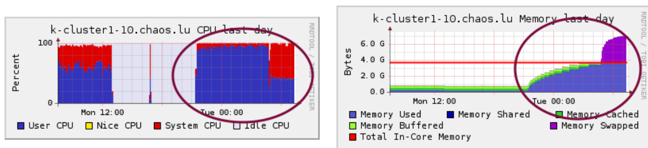

Figure 6. CPU and memory use (small nodes configuration).

3.2.2. Tuning using large nodes only

This time, Acovea was used to explore HPL's parameter space using a reduced problem size of 8,000. As illustrated in Figure 7, the initial generation of tests had an average run-time of 20.8843 seconds, improving to 16.5834 seconds after twenty generations, with 15.98 seconds as the *best-of-best* result. In the second phase, the *best-of-best* parameters previ-

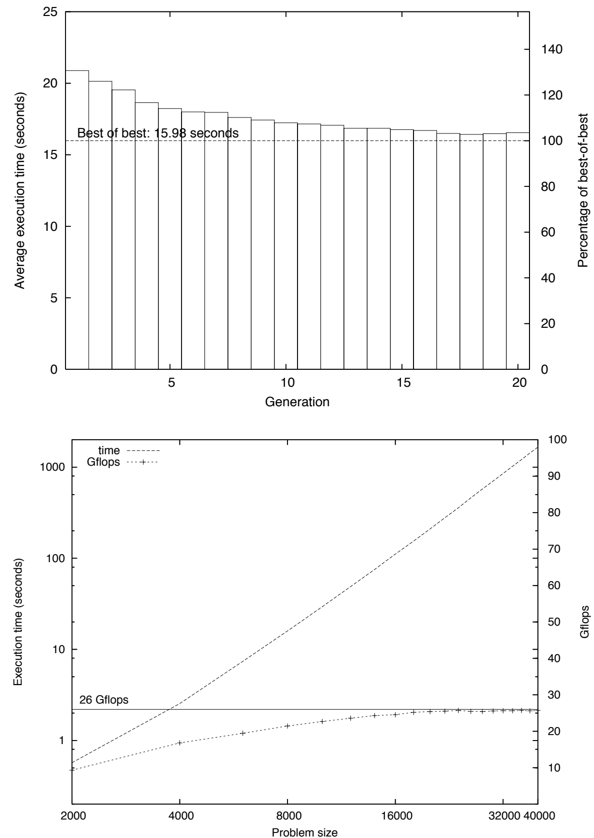

Figure 7. Acovea and HPL evaluation (using large nodes only).

ously obtained were used to determine performance for problem sizes ranging from 2,000 to 40,000. As figure 7 shows, a peak of 25.74 Gflops was achieved with a problem size of 40,000. This problem size resulted in around 6.3GB of memory being used on each node. An examination of table 5 shows that Acovea suggested that seven parameters – P, $NBMIN$, $PFACT$, $DEPTH$, $SWAPPING$, $L1$ and U – should be changed from their default values. Of these, Acovea reported that only three had produced statistically-significant improvements in fitness (P, $DEPTH$ and U). The change of P from four to two (and hence of Q from four to eight) is interesting in that it appears to produce a better match to two nodes, each having eight cores, than the default, although the results were no better than those for the hand-tuned four-by-four configuration. No parameter was reported to produce a statistically-significant worsening of fitness. Observation using Ganglia showed how the genetic algorithm tuned performance by reducing the amount of network traffic generated (and hence system CPU time used) with each succeeding generation — see the circled areas in figure 8.

Network traffic while running benchmarks on the large nodes had a similar profile to that for the small nodes shown in figure 5, but peaked at six MB/sec per node rather than

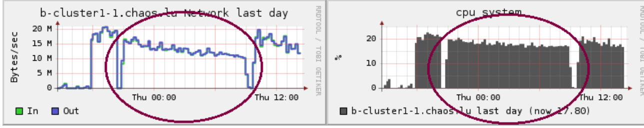

Figure 8. Network traffic and CPU use (large node config.)

ten. Presumably, much network traffic had been replaced by in-memory communication within each node[2]. Unlike the small nodes, the large nodes had clearly reached the limit of their performance before the test was complete, despite having slightly more powerful processors than the small nodes. One can speculate that the reason for this is the considerably slower memory (see table 4) in the large nodes. Equipped with 32GB of memory each, the large nodes could have run problems as large as the $N = 88,000$ case. However, no problem larger than $N = 40,000$ was tried, as run-time would clearly have been excessive and no gain in performance was expected.

3.2.3. Tuning using all nodes

Here also Acovea was used to explore HPL's parameter space using a problem size of 8,000. As illustrated in Figure 9, the initial generation of tests had an average run-time of 30.1619 seconds, improving to 21.5308 seconds after twenty generations, with 9.64 seconds as the *best-of-best* result.

The *best-of-best* parameters were used to determine performance for problem sizes ranging from 4,000 to 84,000. Figure 7 show that a peak of 77.83 Gflops was achieved with a problem size of 80,000. It is important to note that HPL seems to be unable to take advantage of the extra processing power available when all nodes are used: the peak performance is 78 Gflops versus 91 using small nodes only. This may be an indication that the benchmark's algorithm is suited to homogeneous system configurations. In addition, compared to the previous experiments, it can be seen in Figure 9 that convergence is slow, and has not been achieved after twenty generations: the *best-of-best* result is considerably better than the final generation average.

Test run-times were very widely distributed: Figure 10 compares the spread of run-times for the all-node case with those for the small and large nodes only. The latter show well-defined peaks and have almost no members running for more than 40 seconds; in contrast, the all-nodes case shows two small peaks, and has many outliers with very long run-times. Nevertheless, Acovea suggested that five parameters — P, $NBMIN$, $BCAST$, $DEPTH$, and $SWAPPING$ — should be changed from their default values. Of these, P, $BCAST$, and $SWAPPING$ were reported to produce statistically significant improvements in fitness.

[2] Across the cluster, peak traffic was 160MB/sec for small-node benchmarks, and just 12MB/sec for large.

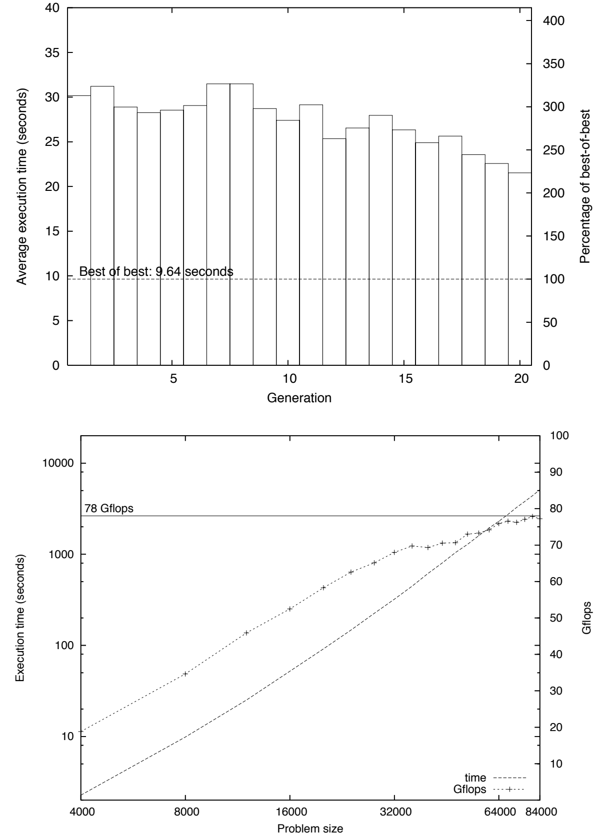

Figure 9. Acovea and HPL evaluation (using all nodes).

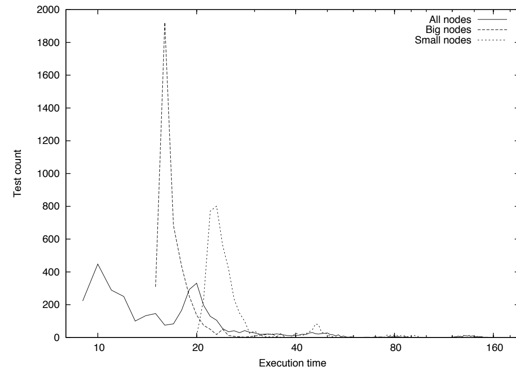

Figure 10. Distribution of genetic algorithm test run-times.

The choice of $P = 2$, and hence $Q = 24$, is interesting. According to [14], "*HPL prefers a* $1 : k$ *ratio, with* k *in* $[1 \ldots 3]$". $P = 2, Q = 24$ is a long way from meeting this criterion, yet an examination of the log files for tests having short run-times shows that it was a frequent choice. It is unlikely that hand-tuning would have discovered this possibility, showing

that a GA can be useful in arriving at unusual yet successful parameter values. Finally, Figure 9 suggests that, unlike the small-nodes case, all-nodes performance has topped out at the largest problem sizes: the figure for $N = 84000$ is slightly below that for $N = 80000$. It could be that the slower large nodes are making the small nodes wait for results, so reducing performance.

3.3. Further Improvements

Examining the results presented in §3.2., it can be seen that the value suggested by Acovea for *NB*, the blocking factor, was 80 in every case. This is the default value set both in the Acovea configuration files and the wrapper for *xhpl*. Two possible explanations are as follows: either $NB = 80$, suggested as a good starting value by the initial tests described in §3.1., is indeed the optimum value; or there is a better value, but, for some reason, Acovea did not find it.

To discover which was the case, a series of trials was run on the small nodes using the *best-of-best* parameters suggested by Acovea in §3.2.1. but with a problem size $N = 8000$ and varying *NB* from 20 to 224. The results, depicted in figure 11, clearly show sharp discontinuities at multiples of 72, all of which giving peaks in performance. Tests for a problem size of 84000 with $NB \in \{72, 144, 216\}$ were then run, giving the results shown in table 3. Of these, the result for $NB = 144$, 111.6 Gflops, was the best achieved so far.

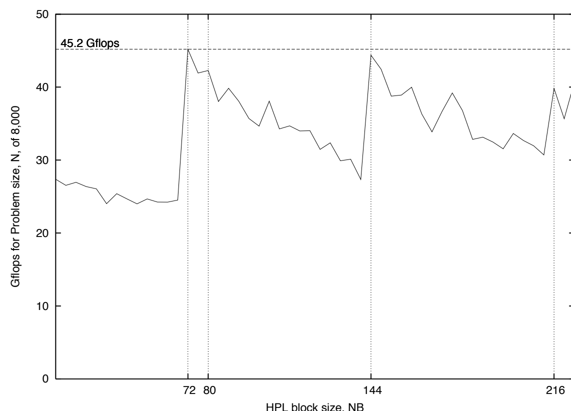

Figure 11. Result of varying *NB* over wide range (small nodes)

The better results presented above raise two questions. Firstly, does the existence of a peak in performance of $NB = 72$ for a problem size of 8,000 imply that there is a peak for the same value of *NB* for the much greater problem size of 84000? Secondly, why did Acovea not find any of the peaks, despite being allowed to vary *NB* from 36 to 256?

To address the first question, a series of tests was run on the small nodes only, varying *NB* from 64 to 96 and *N* from 8,000 to 84,000. The resulting surface appears as in figure 12, and shows clearly that the location of the discontinuity

NB	Performance	Efficiency
72	107.2 Gflops	52.3%
144	**111.6 Gflops**	**54.5%**
216	111.0 Gflops	54.2%

Table 3. Performance for $N = 84000$ and *NB* multiple of 72.

for $NB = 72$ does not vary with *N*. This begins to address the issue raised in §3.2. above concerning whether the parameters that are best for small problems are also best for large. No tests were run to characterize the other discontinuities; their behaviour was presumed to be similar.

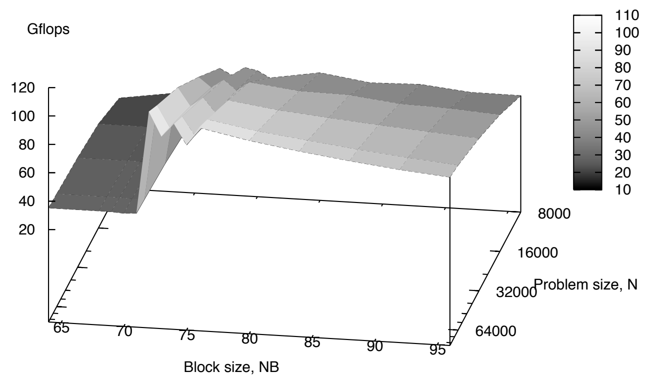

Figure 12. Result of varying *NB* and *N* (over small nodes).

The answer to the second question is simple: Acovea could not find the peak at $NB = 72$ because it was not allowed to: its parameter file allowed *NB* to vary in steps of sixteen away from a default value of 80. The first step below 80 was 64, a value that gave a very poor performance. The GA tried this figure on a few occasions, but the individuals having this gene tended not to reproduce because of their poor fitness. (The first step above 80, 96, produced only slightly worse results on average than the default, and so was explored more often.) To give Acovea a chance to find a peak, another run was tried, allowing *NB* to vary from a default of 36 up to 234 in steps of nine. Results were poor compared to the first run: in most tests, Acovea did not specify a value for *NB*, resulting in the harness's default value of 80 being used; 36 was specified for some tests, giving worse results than the default on average; and a few tests specified 45, giving worse results still. No other values were tried. A first-generation fitness of 39.7891 seconds came down to 28.4151 by the twentieth – not as good as than the first run's final figure of 23.0882 seconds. A third run of Acovea returned the default for *NB* to 80, but allowed variation in steps of eight rather than the sixteen of the original test. The peak for $NB = 72$ was found within a few generations using these parameters, and the final average fitness of 20.3134 seconds improved on the corresponding figure of 23.0882 seconds achieved in the test of §3.2.1. Figure 13 shows these results. The returned "optimistic" op-

tions had also changed, with *NB* replacing *SWAPPING* on the original list.

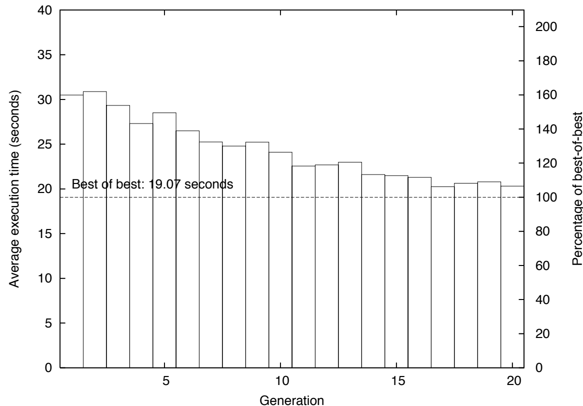

Figure 13. Acovea results with revised parameters.

The *best-of-best* parameters were used to run benchmarks with problem size $N = 84000$ and $NB \in \{72, 144, 216\}$. The results were similar to those shown in table 3 and gave the highest performance to date: 111.7 Gflops.

4. CONCLUSION & PERSPECTIVES

This paper has addressed the issue of extracting the best adapted parameters for the HPL reference benchmark. Adjustment of the seventeen tuning parameters to achieve maximum performance is a time-consuming task that must be performed by hand. The use of a genetic algorithm is proposed here to manage this task with individuals corresponding to an HPL run. Indeed we do not provide here a description of a particular version of a GA. The Acovea framework has been used to validate the approach over a Beowulf cluster composed of heterogeneous resources: a majority of so-called "small" nodes and two "large" nodes. In particular, starting from a hand-tuned performance of 84 Gflops, it was possible to attain the peak performance of 111.6 Gflops on the cluster using a set of parameters determined nearly automatically by Acovea. The main contribution here is not the performance result in itself (obtained using the small nodes only as they seem better to suit the execution of the HPL benchmark), but rather the method used to reach it. This work opens the possibility of creating a fully-automatic benchmark tuning system based on genetic programming. However, an analysis of the impact of the GA configuration (cross-over and mutation rate, number of generations etc.) on parameter determination efficiency remains to be made. Additionally, future research will include evaluation of alternative GA frameworks, such as ParadisEO [3], or MALLBA [2].

REFERENCES

[1] The Top500 project. [Online] see `www.top500.org`.

[2] Geographically Distributed Environments: Combinatorial Optimization Library : the MALLBA project, 2000-2001. MALLBA library tutorial available at `http://neo.lcc.uma.es/mallba/easy-mallba`.

[3] S. Cahon, N. Melab, and E-G. Talbi. "ParadisEO: A Framework for the Reusable Design of Parallel and Distributed Metaheuristics. *Journal of Heuristics*, 10(4):357–380, may 2004.

[4] C-Y. Chou, H-Y. Chang, S-T. Wang, and C-H. Wu. A Semi-Empirical Model for Maximal LINPACK Performance Predictions. In *IEEE International Symposium on Cluster Computing and the Grid (CCGRID'06)*, pages 343–348, Washington, DC, USA, 2006.

[5] J. Dongarra. Basic Linear Algebra Subprograms Technical Forum Standard. *High Performance Applications and Supercomputing*, 16(1–2):1–199, 2002.

[6] R. Eigenmann, G. Gaertner, W. Jones, H. Saito, and B. Whitney. SPEC hpc2002: The next high-performance computer benchmark. In *ISHPC*, pages 7–10, 2002.

[7] E. Gabriel et al. Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary, Sept 2004.

[8] D.E. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley Professional, January 1989.

[9] K.A. De Jong. *Evolutionary Computation*. MIT Press, 2002.

[10] A. Klimovitski. Using SSE and SSE2: Misconceptions and Reality. Technical report, Intel, Mar 2001.

[11] S.R. Ladd. Acovea: Using Natural Selection to Investigate Software Complexities. [Online] see `www.coyotegulch.com/products/acovea/`, 2007.

[12] S.R. Ladd. Evocosm: A C++ Framework for Evolutionary Computing. [Online] see `www.coyotegulch.com/products/libevocosm/`, 2007.

[13] Massie M.L., Chun B.N., and Culler D.E. The Ganglia distributed monitoring system: Design, implementation, and experience. *Parallel Computing*, 30(7):817–840, July 2004.

[14] A. Petitet, R. C. Whaley, J. Dongarra, and A. Cleary. HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers, Jan 2004.

[15] R.M. Stallman et al. *Using GCC: The GNU Compiler Collection Reference Manual*. FSF, 2005.

[16] R. C. Whaley, A. Petitet, and J. Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, Jan 2001.

[17] W. Zhang, J. Fan, and M. Chen. Efficient Determination of Block Size NB for Parallel Linpack Test. In *Proceedings of Parallel and Distributed Computing and Systems (PDCS'2004)*, volume 439. ACTA Press, Nov 2004.

Appendix

Node type	#	Dell model	Processor	cores		Total peak speed	Memory (per-node)
				/node	total		
Frontend	1	SC1425	1 Xeon Dual-Core 3.2GHz	–	–	–	4GB DDR-2 400MHz
NFS	1	PE2450	1 Xeon	–	–	–	4GB DDR-2 400MHz
small	16	PE850	1 Pentium D 3.2GHz	2	32	204.8 Gflops	4GB DDR-2 667MHz
large	2	PE6850	4 Pentium Xeon 3.4GHz	8	16	108.8 Gflops	32GB DDR-2 400MHz
				Total on computing nodes:	48	313,6 Gflops	128 GB

Function		Package	Version	See also
Operating system		Debian	4.0 "*etch*"	`www.debian.org`
Linux Kernel	front-end	–	2.6.18	`www.kernel.org`
	small nodes	–	2.6.22	
	large nodes	–	2.6.23.8	
C/C++ compiler		gcc/g++	4.1.2	`gcc.gnu.org`
Benchmark		HPL	1.0a	`www.netlib.org/benchmark/hpl`
Linear algebra library/scientific library		ATLAS/CBLAS	3.6.0	`math-atlas.sourceforge.net`
Message passing interface		OpenMPI	1.1	`www.open-mpi.org`
GA framework		Acovea	5.1.1	`www.coyotegulch.com`

Table 4. Hardware and software configuration of the cluster for relevant elements at the time of the experiments.

Parameter (in `HPL.dat`)	Default value	Configuration Small	Configuration Large	Configuration All	**Description** (see [14])
procs	32	32	*16*	*48*	Number of processes (nodes)
N	8000	*12000*	8000	8000	Problem size
NB	80	80	80	80	Block size for matrix decomposition
$PMAP$	0	0	0	0	Process mapping (0=row-major, 1=column-major)
P	4	4	**2**	**2**	Process grid rows
Q	8	8	8	**24**	Process grid columns
$THRESHOLD$	16	16	16	16	Error threshold for calculation check
$PFACT$	0	0	**2**	0	Panel factorization (0=left,1=Crout's method, 2=right)
$NBMIN$	1	**4**	**16**	**8**	Recursive stopping criterion in panel factorization
$NDIV$	2	2	2	0	Number of panels created on recursive split
$RFACT$	0	0	0	0	Recursion factor
$BCAST$	0	0	0	**1**	Communication broadcast method
$DEPTH$	0	**1**	**1**	**2**	Look-ahead depth during factorization
$SWAP$	2	2	2	2	Swapping algorithm (0=binary-exchange, 1=spread-roll, 2=mixed)
$SWAPPING$	64	**32**	**32**	64	Switching threshold for mixed swapping
$L1$	0	0	**1**	0	Upper triangle storage form (0=transposed, 1=untransposed)
U	0	**1**	**1**	0	Panel storage form (0=transposed, 1=untransposed)
EQ	0	0	0	0	Use equilibration? (0=no,1=yes)

Table 5. HPL parameters used in the first phase of experiments over the Acovea framework. *italic font* is used to express a non-default value fixed for the benchmark's run, while **bold font** illustrates a non-default value determined by Acovea.