

Use of Cluster OpenMP with the *Gaussian* Quantum Chemistry Code: A Preliminary Performance Analysis

Rui Yang, Jie Cai, Alistair P. Rendell, and V. Ganesh

Department of Computer Science
College of Engineering and Computer Science
The Australian National University,
Canberra, ACT 0200, Australia
{Rui.Yang, Jie.Cai, Alistair.Rendell,
Ganesh.Venkateshwara}@anu.edu.au

Abstract. The Intel Cluster OpenMP (CLOMP) compiler and associated runtime environment offer the potential to run OpenMP applications over a few nodes of a cluster. This paper reports on our efforts to use CLOMP with the *Gaussian* quantum chemistry code. Sample results on a four node quad core Intel cluster show reasonable speedups. In some cases it is found preferable to use multiple nodes compared to using multiple cores within a single node. The performances of the different benchmarks are analyzed in terms of page faults and by using a critical path analysis technique.

Keywords: Cluster OpenMP, performance, quantum chemistry code.

1 Introduction

OpenMP is an attractive shared memory parallel programming model that invites incremental parallelization. Traditionally, however, OpenMP programs have been constrained by the number of processors and available memory on the underlying shared memory hardware, as changing these involves significant expense. By contrast for message passing applications running on a cluster it is usually a relatively easy matter to add a few more nodes to a cluster. In this respect the ability to run an OpenMP code over even a few nodes of a cluster, as promised by the Intel Cluster OpenMP (CLOMP) compiler and runtime environment [1], has an immediate attraction.

CLOMP runs over a cluster by mapping the OpenMP constructs to an underlying page based software Distributed Shared Memory (DSM) system [1]. To transfer an application from native OpenMP to CLOMP is in principle straightforward; requiring shared variables to be identified as ‘sharable’ either automatically by the compiler or manually by the programmer placing sharable directives where the variable is declared. Sharable data are placed on sharable pages which are read/write protected using the `mprotect()` system call. When an OpenMP thread accesses data on one of these pages a segmentation fault occurs that is caught by the CLOMP runtime library. The library is then responsible for retrieving the relevant memory page from the distributed

environment, ensuring that the data is consistent with what is expected given the OpenMP memory consistency model.

Compared to regular OpenMP, the additional overhead of using CLOMP is primarily that associated with the cost of servicing the various segmentation faults. As the latency of any communication between nodes on a cluster is in the order of microseconds, while the clock speed of a typical processor is sub nanosecond, for an application to scale well using CLOMP it will need to perform significant computation between memory consistency points. Ideal CLOMP applications are those that may access large amounts of sharable data but modify only a relatively small amount of this data, and make limited use of thread synchronization. Good examples are applications for doing rendering, data-mining, all kinds of parallel search, speech and visual recognition, and genetic sequencing [2].

In this paper we report on our efforts to adapt parts of the *Gaussian* computation chemistry package [3] for use with CLOMP. In section 2 we give a brief overview of the *Gaussian* code and the parts that we have attempted to use CLOMP with. Section 3 details our benchmark environment and computations, while section 4 discusses the observed performance attempting to rationalize it in terms of the performance model of Cai et al. [4]. Finally section 5 contains our conclusions.

2 The *Gaussian* Program and Its Use with CLOMP

Gaussian is a general purpose computational chemistry package that can perform a variety of electronic structure calculations [3]. It consists of a collection of executable programs or “Links” that are used to perform different tasks. A complete *Gaussian* calculation involves executing a series of Links for different purposes such as data input, electronic integral calculation, self-consistent field (SCF) evaluation etc. Each Link is responsible for continuing the sequence of Links by invoking the `exec()` system call to run the next Link. Not all Links run in parallel; those that run sequentially are mainly responsible for setting up the calculations and assigning symmetry and will usually take only a short time to execute [5].

Two fundamental quantum chemistry methods are Hartree-Fock theory and Density Functional Theory (DFT). In the *Gaussian* code the implementations of both methods are very similar. Specifically in both cases electrons occupy molecular orbitals that are expanded in terms of a set of basis functions (usually atom centered), for which the coefficients are obtained by solving an equation of the form:

$$F C = \epsilon S C \quad (1)$$

where C is a matrix containing the orbital coefficients; S is the overlap matrix with elements that represent the overlap between two basis functions; ϵ is a diagonal matrix of yet to-be-determined molecular orbital energies, and in the case of Hartree-Fock theory, F is the Fock matrix. The Fock matrix is defined as:

$$F_{\mu\nu} = H_{\mu\nu}^{Core} + \sum_{\lambda\sigma}^{N_{basis}} P_{\lambda\sigma} \left[(\mu\nu|\lambda\sigma) - \frac{1}{2}(\mu\sigma|\lambda\nu) \right] \quad (2)$$

where $H_{\mu\nu}^{Core}$ involves integrals representing the kinetic energy of the electrons and their interactions with the nuclei, and the remaining part of the right side of Eqn. (2) comes from the interactions between electrons. In the latter the two-electron repulsion integrals (ERIs) are defined as:

$$(\mu\nu|\lambda\sigma) = \int d\mathbf{r} \int d\mathbf{r}' \frac{\phi_{\mu}(r)\phi_{\nu}(r)\phi_{\lambda}(r')\phi_{\sigma}(r')}{|\mathbf{r}-\mathbf{r}'|} \quad (3)$$

and P is the density matrix which for a closed shell system is given by:

$$P_{\mu\nu} = 2 \sum_m C_{\mu m} C_{\nu m}^* \quad (4)$$

with the summation being over half the number of electrons in the system.

For DFT the equations are essentially the same, except that F is now the Kohn-Sham matrix K , the second “exchange” term in the summation in Eqn. (2) is dropped, and an additional term is added that involves a numerical integration over all space of a quantity that depends on the electron density. Between HF and DFT there lies a number of so called hybrid schemes that include some HF exchange as well as some of the DFT terms requiring numerical integration.

A HF energy calculation performed using *Gaussian* uses a subroutine called PRISM to calculate the relevant ERIs and form their contribution to the Fock matrix. For DFT calculations a subroutine called PRISMC is used to evaluate the ERIs, while the numerical integration is undertaken by a routine called CALDFT. For hybrid DFT methods PRISM is used together with CALDFT. Collectively these three routines will typically consume over 90% of the total execution time for a HF or DFT energy calculation. For this reason these routines have already been parallelized for shared memory hardware using OpenMP; adapting these routines to run with CLOMP and studying their performance is the major objective of this work.

After the formation of the F (or K) matrix, the molecular orbital coefficients C are determined by solving Eqn. (1). This involves diagonalization of F (or K). The new molecular orbital coefficients are then put back into Eqn. (4) to form a new density matrix from which a new F (or K) matrix is computed. This process continues until there is no change in the density matrix between iterations, at which point the interactions between the electrons represents a “self-consistent field”. In *Gaussian*, Link 502 is used to perform the SCF computation and evaluate the total energy for a given configuration of the atomic nuclei.

Most of quantum chemistry revolves around atomic configurations where there are no net forces on the individual nuclei. Finding these locations involves evaluating both the energy of the system and its derivative with respect to a displacement of any of the atomic nuclei. In *Gaussian* such force evaluations are performed for HF and DFT methods by using the same three routines (PRISM, PRISMC and CALDFT) but called from Link 703.

The basic strategy for a HF/DFT energy evaluation is as follows:

1. Start the Link using a single process (the master thread) and allocate a large working memory space using the `malloc()` system call. Data used in the evaluation of the F (or K) matrix is subsequently stored into this working space (e.g. the density matrix and orbital shell information). All density matrix related input data gets arranged as a single block, with the length of this “density matrix block” being dependent on the size of the problem (atoms, theoretical method and basis set size).
2. In the routine responsible for computing the F (or K) matrix, the remaining working memory space is divided into N_{thread} blocks. Each of these “Fock matrix blocks” will be used as private space for a single thread to store its contribution to the F (or K) matrix, and for the intermediate arrays used in F (or K) matrix construction.
3. Create N_{thread} threads using an OpenMP `parallel` directive. Each thread reads in the relevant parts of the shared density matrix block, calculates a subset of the ERIs and does a subset of the numerical integration, computing a contribution to its own local F (or K) matrix.
4. The child threads terminate when all their ERIs have been evaluated and their portion of the numerical integration is complete. The master thread then sums the N_{thread} private copies of the F (or K) matrices. This step is performed sequentially on master thread since the cost of adding a few $O(n^2)$ F (or K) matrices is usually small compared to the $O(n^4)$ cost of forming them.

In the case of a force evaluation the approach is similar, except each thread is now making contributions to the gradient vector rather than the F (or K) matrix.

We note that since the density matrix is read only, while the F (or K) matrices are private to each thread during the parallel section, in the CLOMP implementation once the initial penalty associated with fetching the relevant pages across the network to the right node has occurred, there should be little contention between threads. Thus HF and DFT energy and gradient calculations within *Gaussian* have the potential to achieve high parallel performance when using CLOMP provided that the time required to evaluate the ERIs and/or do the numerical integration is sufficiently large to mask the network overheads.

To use CLOMP in the code we firstly replace the `malloc()` system call by the CLOMP analogue `kmp_sharable_malloc()`; this makes the whole of the *Gaussian* working array shared between all threads in the Link. A large amount of the shared variables were automatically tagged as sharable by the compiler, however, a significant number of other variables had to be identified by hand (around 60 sharable directives were inserted for Link 502). The parallel Links were then compiled and linked by using the CLOMP Intel 10.0 compiler with the `-cluster-openmp`, `-clomp-sharable-commons`, `-clomp-sharable-localsaves` and `-clomp-sharable-argexprs` directives.

For the purpose of this study, a number of other OpenMP directives were deliberately disabled since they correspond to fine grain loop parallelization that is known not

to run well using CLOMP over multiple nodes in a cluster. In due course these directives should be reactivated, but with parallelism limited to just the number of threads within the master node.

3 Experimental Details

All performance experiments were carried out using a Linux cluster containing 4 nodes each with a 2.4GHz Intel Core2 Quad-core Q6600 CPU and 4GB of local DRAM memory. The cluster nodes were connected via Gigabit Ethernet.

The Intel C/Fortran compiler 10.0 was used to compile and build the two ported parallel Links of the *Gaussian* development version (GDV) with the CLOMP flags given above.

Five different HF and DFT benchmark jobs were considered spanning a typical range of molecular system. These are detailed in Table 1.

Table 1. Benchmarks used for performance measurements of the CLOMP implementation of Gaussian program

Case	Method	Basis	Molecule	Links	Routines Used
I	HF	6-311g*	Valinomycin	502	PRISM
II	BLYP	6-311g*	Valinomycin	502	PRISMC, CALDFT
III	BLYP	cc-pvdz	C60	502&703	PRISMC, CALDFT
IV	B3LYP	3-21g*	Valinomycin	502&703	PRISM, CALDFT
V	B3LYP	6-311g**	α -pinene	703	PRISM, CALDFT

Only the first SCF iteration is measured within Link 502 (the time to complete a full SCF calculation will scale almost linearly with the number of iterations required for convergence). The reported times comprises both the parallel formation of the F (or K) matrix and its (sequential) diagonalization.

4 Results and Discussion

Table 2 shows the speedups, defined as the ratio of the serial run time (t_s) divided by the parallel run time (t_p) for the five benchmark tests and Links 502 and 703 as applicable. The effect of using multiple cores and multiple nodes is contrasted. We use the single-thread non-CLOMP OpenMP executing time to calculate the speedup in the present work. We note also that the *Gaussian* code uses cache blocking when evaluating integrals in order to maximize performance [6]. For the quad-core processor used in this work the presence of a shared level 2 cache means that the optimal cache blocking size varies with the number of cores being used. In an attempt to remove this effect, results have been calculated by using the optimal cache blocking size for a given number of threads on a node.

Table 2. Speedups of Link 502(L502) and Link 703(L703) for benchmark systems

$N_{node} \times N_{core}$		Case I	Case II	Case III		Case IV		Case V
		L502	L502	L502	L703	L502	L703	L703
1-thread	1x1	0.93	0.77	0.79	1.02	0.86	0.98	0.93
	1x2	1.73	1.48	1.55	1.99	1.61	1.90	1.75
2-thread	2x1	1.78	1.56	1.56	1.97	1.45	1.92	1.79
	1x4	2.91	2.53	2.83	3.26	2.55	3.07	2.67
4-thread	2x2	3.16	2.83	2.93	3.88	2.30	3.67	3.28
	4x1	3.25	2.94	2.98	3.91	2.21	3.71	3.12
8-thread	2x4	4.83	4.26	4.95	6.41	2.87	5.83	4.49
	4x2	5.55	4.75	5.24	7.68	3.13	7.09	5.30
16-thread	4x4	7.30	5.25	7.71	12.47	2.88	10.76	6.74

The results for the 1-thread case given in Table 2 show that in some cases there exists a significant slowdown associated with using CLOMP over “normal” OpenMP. Similar results have been reported elsewhere [7]. For all multi-thread cases speedups are observed. For two threads this varies from 1.45 to 1.99. The difference between using two cores within one node versus one core each on two nodes is surprisingly small, with the biggest difference being for case IV where using two cores within one node is preferable.

With 4 threads speedups of between 2.21 and 3.91 are observed. In most cases performance is better when using multiple nodes compared to using all cores within one node. Similar results are also found with 8 threads. This suggests that either memory bandwidth and/or contention due to the shared cache is limiting scalability when using 4 cores on a single chip.

For 16 threads the best speedup observed is 12.47 for Link 703 of case III. However we also see a relatively poor speedup of just 2.88 for Link 502 of Case IV. This is due to the fact that for this system Link 502 takes relatively little time (~100s), and the overhead of parallelization with 16 cores is large. Also, for this calculation the sequential time becomes significant.

Overall the results indicate that for large HF and DFT *Gaussian* energy and gradient calculations CLOMP is able to provide reasonable parallel performance over four nodes. To analysis the results further we use the performance model of Cai et al. [4, 8]. In this model execution time is based on the number, types and the associated approximate costs of page faults that occur during the execution of each parallel section. For *Gaussian* an outline of the important features of the two parallel Links and the sources of the page faults is given in Fig. 1.

Start of the Link:

```
Allocate working array in the sharable heap using
kmp_sharable_malloc();
```

Within the Link:

...

```
Obtain new density matrix
```

```
Parallel loop over  $N_{thread}$  (OpenMP Parallel region):
```

```

    Call Prism, PrismC or CalDFT to calculate  $1/N_{\text{thread}}$  of the
    total integrals and save their contributions to each
    thread's private Fock matrix;
End Parallel loop
loop over i=2,  $N_{\text{thread}}$  (sequential region)
    Add Fock matrix created by thread  $i$  to the master
    thread's Fock matrix;
endloop
...
Exit Link;

```

Fig. 1. Basic CLOMP parallel construct in Link 502 of the Gaussian program (Link703 is similar, except contributions are now to the gradient vector)

At the beginning of each Link the master threads obtains a new copy of the density matrix. This will cause the underlying DSM to pass write notices to all other threads at the implicit barrier that occurs at the top of the next parallel region indicating that the corresponding memory pages have been modified. Within the parallel region each thread will read elements of the density matrix and update elements of their F (or K) matrix. This will cause a series of page faults to occur as new memory pages are accessed. Over time each thread is likely to access all density matrix elements and make contributions to all F (or K) matrix elements. For the density matrix this means each thread will end up fetching over its own read-only copy of the entire density matrix from the master thread. While for the F (or K) matrix each thread will end up creating its own writeable copy of this matrix. After the CLOMP parallel region has finished, a moderate amount of fetch page faults will occur on the master thread as it seeks to sum together the partial contributions to the F (or K) matrices computed by the other threads.

In the analysis model of Cai et al [4], the overhead associated with a CLOMP parallel calculation is determined by the thread that encounters the largest number of page faults (the critical path). The total CLOMP execution time on p threads is given by:

$$Tot(p) = \frac{Tot(1)^{par}}{p} + T^{crit}(p) = \frac{Tot(1)^{par}}{p} + Max_{i=0}^{p-1} (N_i^w C^w + N_i^f C^f) \quad (5)$$

where $Tot(1)^{par}$ is the sum of the elapsed times for the parallel regions when the application is run using just one thread, p is the number of threads actually used, $T^{crit}(p)$ is the page fault time cost for the thread that encounters the maximum number of page faults in the p -thread calculations. $T^{crit}(p)$ is further expanded in terms of N_i^w and N_i^f , the number of write and fetch page faults respectively for thread i in the parallel region and their corresponding costs, C^w and C^f . The $Max_{i=0}^{p-1}$ in Eqn. (5) follows the thread in each parallel region that has the maximum page fault cost.

To obtain the page fault number for the five *Gaussian* test cases we have used the SEGVprof profiling tool that is distributed as part of the CLOMP distribution. This tool creates profile files (.gmon) for all CLOMP processes, reporting the segmentation faults occurring for each thread in each parallel region. SEGVprof also provides a script (**segvprof.pl**) that reports aggregated page fault counts. This script was extended

to produce per-thread results. The values of C^w and C^f are system dependent and have been measured using two-nodes with the code given in Ref. [8]. This gave the values of $C^w=10\mu\text{s}$ and $C^f=171\mu\text{s}$. These values are expected to be lower limits on the cost of page faults.

Since the model of Cai et al. [4] assumes there is no sequential time we give in Table 3 execution times and associated critical path page fault counts for just the PRISM, PRISMC and CALDFT portions of Links 502 and 703. This shows that the elapsed times cover a large range from just over 6 seconds to nearly 2900 seconds. Not surprisingly total page fault counts roughly reflect the size of the system, which in return is related to execution time. For any given test case and routine, the number of write and fetch page faults appears to remain roughly constant when going from 2 to 4 threads (1 thread/node). This is in line with the expectation expressed above, i.e. that

Table 3. The elapsed time ($Tot(n)$) and measured page fault numbers of subroutine PRISM, PRISMC and CALDFT in Link 502 (L502) and Link 703(L703) using 2-thread and 4-thread (1 thread/node) for all experimental cases. Also shown are ΔTot and ΔT^{crit} defined in Eqn. (6) (see text for further details).

Experiment (Links)	Routines	N_{thread}	$Tot(n)$ (sec.)	Max page faults per thread		From Eqn. 6 (sec.)	
				Write	Fetch	ΔTot	ΔT^{crit}
Case I (L 502)	PRISM	2	751.8	4931	13148	20.8	2.30
		4	386.3	4934	13150		
Case II (L 502)	PRISMC	2	285.2	5104	6036	7.4	1.17
		4	146.3	5347	6273		
	CALDFT	2	185.1	8043	3933	2.7	0.66
		4	93.9	7540	3680		
Case III (L 502)	PRISMC	2	286.1	5337	5240	8.9	0.96
		4	147.5	5628	5253		
	CALDFT	2	95.0	3852	1493	4.0	0.54
		4	49.5	4146	2187		
Case III (L 703)	PRISMC	2	2894.3	6038	8577	13.7	1.53
		4	1454.0	6043	8580		
	CALDFT	2	272.8	6682	1011	1.2	0.28
		4	137.0	6475	1127		
Case IV (L 502)	PRISM	2	53.9	1497	3576	4.1	0.63
		4	29.0	1500	3579		
	CALDFT	2	86.1	3170	916	0.5	0.22
		4	43.3	3069	1007		
Case IV (L 703)	PRISM	2	353.9	1853	5224	9.9	0.91
		4	181.9	1854	5229		
	CALDFT	2	109.6	8497	1077	-1.2	0.31
		4	54.2	8954	1167		
Case V (L 703)	PRISM	2	21.2	1151	548	1.0	0.11
		4	11.1	1151	551		
	CALDFT	2	12.0	3849	91	0.6	0.05
		4	6.3	3344	97		

each thread will read most density matrix elements and produce contributions to most F (or K) matrix elements. In general the number of fetch faults are more than the number of write faults for PRISM and PRISMC, but much less for CALDFT. This reflects the fact that PRISM or PRISMC is called before CALDFT, so the fetch faults associated with creating a read only copy of the density matrix occur during execution of PRISM/PRISMC not CALDFT. (It is also interesting that in comparison to the page fault counts given by Cai et al. [8] for the NAS parallel benchmark codes the absolute value of the counts for *Gaussian* are much smaller, despite the fact that the execution time is comparable.)

To investigate the applicability of the Cai et al. [4] model we note that there should exist the following equality between 2-thread and 4-thread calculations:

$$\Delta Tot = 2 \times Tot(4) - Tot(2) = 2 \times T^{crit}(4) - T^{crit}(2) = \Delta T^{crit} \quad (6)$$

Thus for the model of Cai et al. [4] to hold the scaled difference between the measured elapsed times reported in Table 2 when using 4 and 2 threads should be equal to the time difference computed using write and fetch page fault numbers also given in Table 2, combined with the cost penalty values of $C^w=10\mu\text{s}$ and $C^f=171\mu\text{s}$. To test this we plot in Fig. 2 these two values. The results clearly show a large difference, although interestingly there does appear to be a linear relationship between the two quantities with a scale factor of around 8.4.

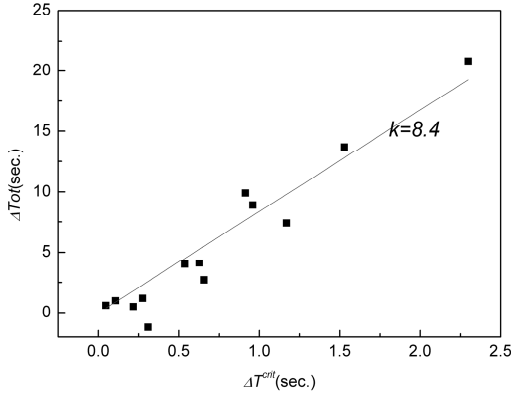


Fig. 2. $\Delta Tot \sim \Delta T^{crit}$ for the data given in Table 2

Possible causes for the inability of the Cai et al. [4] model to describe correctly the performance of *Gaussian* are its failure to account for load imbalance and repeated computation (reflected by the term $\frac{Tot(1)^m}{p}$ in Eqn. (5)), the assumption that page faults occurring on different threads are fully overlapped, and assignment of a fixed cost to servicing a fetch fault regardless the number of threads involved. In this case it is most likely that load imbalance and repeated computation are the main problems, since the total number of page faults appears to be too small compared to the total execution time

for this to be a major cause for error. Specifically for case I and Link 502 ΔT^{cri} as evaluated using the page fault counts in Table 3 with costs of $C^w = 10\mu\text{s}$ and $C^f = 171\mu\text{s}$ is just 2.25s, yet ΔT_{Tot} is 20.8 seconds.

5 Conclusions

We have investigated the performance of the CLOMP implemented quantum chemistry software package *Gaussian* on the 4-node Linux distributed memory system equipped with Intel Quad-core processors. Comparable or better scalability was found for using multi-nodes compared to multi-cores. Page fault measurements reveals relatively low counts within parallel regions, implying that HF and DFT energy and gradient computations within *Gaussian* are well suited to implementation with CLOMP as the effort associated with keeping the sharable memory consistency is low. A critical path model has been used to analyze performance, but the accuracy of this model appears to be limited due to load imbalance. Work is currently underway to extend the model of Cai et al. [4] to include load balancing, and also to study use of CLOMP for other parts of the *Gaussian* code.

Acknowledgments. This work is funded by Australian Research Council Linkage Grants LP0669726 and LP0774896 with support from Intel Corporation, *Gaussian* Inc. and Sun Microsystems.

References

1. Hoeflinger, J.P.: Extending openmp to clusters. White Paper, Intel. Corporation (2006)
2. Hoeflinger, J.P., Meadows, L.: Programming OpenMP on Clusters. HPCwire 15(20) (May 19, 2006), <http://www.hpcwire.com/hpc/658711.html>
3. Frisch, M.J., et al.: Gaussian 03. Gaussian, Inc., Wallingford CT (2004)
4. Cai, J., Rendell, A.P., Strazdins, P.E., Wong, H.J.: Performance model for cluster-enabled OpenMP implementations. In: Proceeding of 13th IEEE Asia-Pacific Computer Systems Architecture Conference, pp. 1–8 (2008)
5. Sosa, C.P., Andersson, S.: Some Practical Suggestions for Performing Gaussian Benchmarks on a pSeries 690 System, IBM Form Number REDP0424, April 24 (2002)
6. Yang, R., Antony, J., Janes, P.P., Rendell, A.P.: Memory and Thread Placement Effects as a Function of Cache Usage: A Study of the Gaussian Chemistry Code on the SunFire X4600 M2. In: The 2008 International Symposium on Parallel Architectures, Algorithms, and Networks, pp. 31–36. IEEE Computer Society, Los Alamitos (2008)
7. Terboven, C., an Mey, D., Schmidl, D., Wagner, M.: First Experiences with Intel. Cluster OpenMP. In: Eigenmann, R., de Supinski, B.R. (eds.) IWOMP 2008. LNCS, vol. 5004, pp. 48–59. Springer, Heidelberg (2008)
8. Cai, J., Rendell, A.P., Strazdins, P.E., Wong, H.J.: Predicting performance of Intel Cluster OpenMP with Code Analysis method, ANU Computer Science Technical Reports, TR-CS-08-03 (2008)