OCCAM

OXFORD CENTRE FOR COLLABORATIVE APPLIED MATHEMATICS

Inner product computation for sparse iterative solvers on distributed supercomputer

by

Sheng-Xin Zhu
Tong-Xiang Gu
Xing-Ping Liu

OCCAM Preprint Number 12/81

Inner product computation for sparse iterative solvers on distributed supercomputer ☆

Sheng-Xin Zhu[a,b,1], Tong-Xiang Gu[c,2], Xing-Ping Liu[c,2]

[a] *Oxford Centre for Collaborative and Applied Mathematics, 24-29 St Giles', Oxford, OX1 3LB, U.K.*
[b] *Numerical Analysis Group, Mathematical Institute, 24-29 St Giles', Oxford, OX1 3LB, U.K.*
[c] *Laboratory of Computational Physics, Institute of Applied Physics and Computational Mathematics, P.O. Box, 8009, Beijing 100088, P.R China.*

Abstract

Recent years have witnessed that iterative Krylov methods without re-designing are not suitable for distribute supercomputers because of intensive global communications. It is well accepted that re-engineering Krylov methods for prescribed computer architecture is necessary and important to achieve higher performance and scalability. The paper focuses on simple and practical ways to re-organize Krylov methods and improve their performance for current heterogeneous distributed supercomputers. In construct with most of current software development of Krylov methods which usually focuses on efficient matrix vector multiplications, the paper focuses on the way to compute inner products on supercomputers and explains why inner product computation on current heterogeneous distributed supercomputers is crucial for scalable Krylov methods. Communication complexity analysis shows that how the inner product computation can be the bottleneck of performance of (inner) product-type iterative solvers on distributed supercomputers due to global communications. Principles of reducing such global communications are discussed. The importance of minimizing communications is demonstrated by experiments using up to 900 processors. The experiments were carried on a Dawning 5000A, one of the fastest and earliest heterogeneous supercomputers in the world. Both the analysis and experiments indicates that inner product computation is very likely to be the most challenging kernel for inner product-based iterative solvers to achieve exascale.

Keywords: Minimizing communication, Krylov methods, Heterogeneous computing
2010 MSC: , 65F10, 68W10, 68W15

1. Introduction

The high resolution numerical simulation of many physical processes, natural phenomenons and engineering problems often result in massive spare linear or nonlinear algebraic systems. Solving these large algebraic systems consumes so much time in the entire simulation that it usually becomes the bottleneck of many simulations. Parallel Krylov iterative solvers are one of the most widely used and increasingly demanded methods in such computationally intensive projects. It is being widely informed and accepted that there is a fundamental difference between the parallel algorithms for distributed computers and the corresponding serial versions. On shared memory computers, reducing computations, exploring parallelism and increasing cache hit rate (or data reuse rate) are always given to priority, while on distributed heterogeneous supercomputers, priority is often given to minimizing all kinds of communications. Recent year has witnessed that minimizing communication in all kinds of algorithms for multi-core computing

Email addresses: shengxin.zhu@maths.ox.ac.uk (Sheng-Xin Zhu), txgu@iapcm.ac.cn (Tong-Xiang Gu), xpl@iapcm.ac.cn (Xing-Ping Liu)
URL: https://www.maths.ox.ac.uk/contact/details/zhus (Sheng-Xin Zhu)

[1] The authors' research is partly supported by Award no KUK-C1-013-04, made by King Abdullah University of Science of Technology.
[2] The authors' research is partly supported by the NSF of China (No. 61170309, 91130024 and 60973151) and the key project of scientific and technical development of China Academy of Engineering Physics (2012A0202008 and 2011A0202012).

platforms has drew increasing attention. In order to achieve the ambitious exascale computing, research involving re-engineering traditional algorithms for the current multi-core computing platforms like GPU based clusters and next generation computer architecture or the so-called emerging (newly emerged) heterogeneous supercomputer have being launched one after another. However, most current research in this area focuses on small GPU cluster computing platforms, few results on heterogeneous supercomputers has been reported. Even though experiments on supercomputers were reported, people usually stress on what a specific and challenging application they implicated and focus on natural phenomena of science or engineering problems, relative less attention focused on the performance of some foundational algorithms on heterogeneous supercomputers. The paper focuses on the performance of Krylov methods themselves on heterogeneous supercomputers rather than other applications. We focus on one of the computation kernels of Krylov iterative methods: inner product computation. Theoretical analysis and experiments will demonstrate why such computation on heterogeneous supercomputers can be the most important kernel of Krylov methods.

The paper is organized as follows. Section § 2 first explains what is communication and why minimizing communication is crucial according to current technologies trends and communication complexity analysis. Section § 3 discuss principles to reduce communications. A detailed case study is presented in section § 4 to show how to reconstruct Krylov methods. Section § 5 analyze the performance of the reconstructed methods and the corresponding original methods. And finally we present some numerical results to verify our analysis and give some conclusions.

2. Communication and Computation Complexity

There are many heterogeneity of current supercomputers all of which should be taken into account when constructing efficient parallel algorithms, while we only stress the aspect that the computer consists of many nodes which themselves consist of multiple processors or every node is like a GPU. In this paper, *communication* refers to the process of exchanging data between different computing nodes or processors via Message Passing Interface (MPI). Other data movements in local shared memory parts without using MPI which often require efficient data structure to increase the cache hit rate go beyond our discussion.

Communication time consists of two parts:one part is called *set up time* and the other part is *message passing time*. The set up time which is necessary to find and prepare the right and available computing resources (computing nodes or processors) mainly depends on *latency* of the underlying systems, while the part of message passing time depends on the ratio of the data size and the bandwidth of the systems. The set up time is usually much bigger than the message passing part. Further more the current technologies shows that the gap between the improvement of latency and bandwidth is still increasing. It has been noted that float points speeds increases at 59%/year, bandwidth increases 26%/year, while only 5.5%/year can be reduced for latency [11], pp.109. And it is generally true for many technologies that latency improves much more slowly than bandwidth [29]. Such a trend suggests that computation in float points is cheap and will be much cheaper than communications, and also indicates that communication complexity analysis will play an more important role for large scale distributed computing. The gap between the bandwidth and latency demonstrates that minimizing the set up time is crucial to minimize the total communication time. Communication complexity analysis should take the two parts into consideration.

As known, there are three basic time-consuming computational kernels of Krylov subspace methods, namely, vector update, matrix-vector multiplications and inner product computation. Table 1 shows their computation and communication complexity. Vector updates is parallel in nature and there is no need for communication. When the data structure of matrix is well organized or the sparse matrix has a special structure, sparse matrix-vector multiplication usually only need *local communication* which is denote as μ in Table 1. μ is often in the order of $O(1)$ against P. While inner product computation needs *global communications* whose complexity is $O(\log_2 P)$. This term is due to the tree reduction algorithm to compute the inner product [12, p.260]. From Table 1, it's easy to formulate the total time for per iterate step of Krylov solvers provided one knows how many number of the three different computational kernels in each loop. The formulae is given by [41]

$$T = \underbrace{\phi(N/P)t_{fl}}_{\text{computation time}} + \underbrace{\psi(t_s, t_w)\log_2 P}_{\text{global communication}} + \underbrace{\mu}_{\text{local communication}} . \tag{1}$$

It is noted that when the number of processors P increases constantly with the problem size N, say, N/P = constant (the so-called weak scale), the computation time almost keep a constant. Or we can say the computational time is in

the order $O(1)$ of P. While the communication time is in the order of $O(\log_2 P)$ when P is large. In this case only one

Table 1: Computation and Communication Complexity of Kernels of Krylov Methods

Name	Operations	Computation time	Communication time
matrix-vector multiplication	$y \leftarrow Ax$	$n_z N t_{fl}/P$	μ
one inner product	$dot \leftarrow x^T y$	$2Nt_{fl}/P$	$2(t_s + t_w)\log_2 P$
k inner products	—	$2kNt_{fl}/P$	$2(t_s + kt_w)\log_2 P$
vector update	$y \leftarrow ax + y$	$2Nt_{fl}/P$	—

N: the number of unknowns; P: the number of processors; n_z: the average number of non-zeros per row of A; t_{fl}: the time per float point operation, t_s: the set up time, t_w: the time of passing per word (unite) of message. Usually, For distributed systems,usally $t_s \gg t_w$.

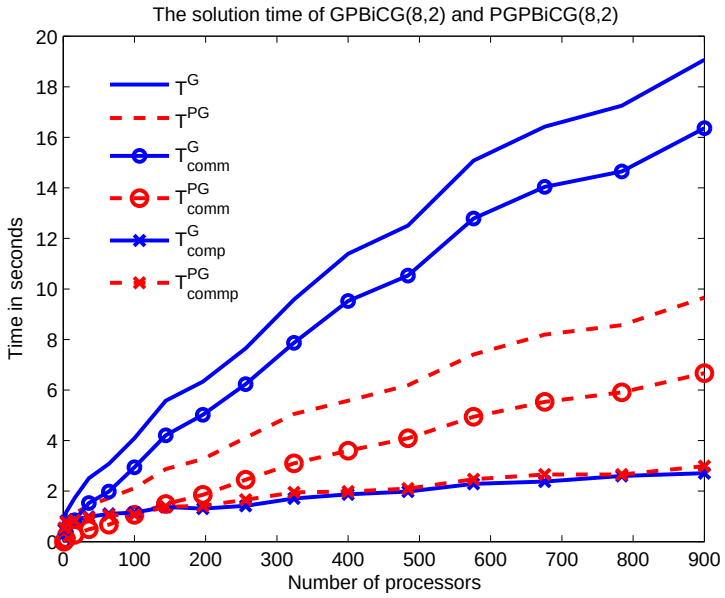


Figure 1: The solution time of two version of GPBiCG(m, ℓ) method (one of the Krylov methods) using up to 900 processors. The blue curves represents the MPI implementation of the original version of GPBiCG(8, 2) and the red curves represent the corresponding MPI implementation of improved or parallel versions PGPBiCG(8, 2). Form up to bottom the curves represent for the total time, communication time and computation time respectively. The two versions of the same algorithm take almost the same computation time. While the parallel version take much less communication time.

or two several such global communications can consume most of time in every iterate, and even dominate the entire time when used processors increasing. Therefore, it is the communication especially the global ones rather than the computation that should be given the priority consideration for distributed heterogeneous supercomputers.

To further motivate our discussion, a numerical experiment is presented in Figure 1. The numerical experiments were designed to investigate which kernel of the Krylov methods is the most important when the number of processors increases. One of the Krylov methods named GPBiCG(m, ℓ) was considered.[13]. The performance of the original version [13] and the corresponding parallel version [41] are reported. As shown in Figure 1, when the number of processors is larger than 100, the total solution time will dominated by the communication time.

The following section will show how tho reduce the communication time for Krylov methods.

3. Strategies for Reducing Communications

As discussed in the last section and seen in Figure 1, communication can dominate the simulation time of parallel Krylov methods when the number of processors increases. Thus finding efficient ways to reduce the communication is crucial to make Krylov iterative methods more scalable for current heterogenous supercomputers.

Three methods to reduce communication has been considered. The first one, or perhaps the most attractive one is to remove inner production computations which need global communications to develop inner product free iterative solvers. For example, the inner product in CG methods can be replaced by solving a small linear system as described in the s-step methods [4, 5] and multiple search direction conjugate gradient method [16, 17]. The second remedy to re-organize the underlying algorithm is to arrange more inner products can be computed at the same *synchronization points*. In this way, the number of inner products usually increases, but all the inner products can be computed and passed concurrently. This is based on the fact that the set up time is usually much bigger than the message passing time (i.e., $t_s \gg t_w$) and reducing the communication time mainly depends on reducing the set uptime. And third, communications should be overlapped with computations as many as possible.

It should be pointed that communication hinders the scalability of iterative Krylov methods has been noticed since 1980s. Pioneer researchers has also proposed some original ideas to reduce communications [6, 7, 8, 23, 24, 25]. However, due to the memory size constraints, the early pioneers work still more focus on maximizing flop point operations per unit memory. With the fast development of hardware, more and more memory and processors are available for general users, and the importance of reducing communications begin to get more public aware, there is a trend to re-engineer all the classical serial algorithms by minimizing the communications and to design communication avoiding Krylov iterative methods [1, 10, 21, 26, 27, 34]. Most of this research are software-development oriented, carefully study and tedious attentions are paid to the underlying computer architectures.

The remaining of the paper will focus on how to reschedule operations in the Krylov methods in order to reduce the number of synchronization points. An example will be given to show how to develop mathematically equivalent iterative methods by *residual split strategy*. The fundamental idea of residual split strategy is using cheap computation to exchange for the expensive communications. Basically, the residual vectors will be represents by a combination of several vectors of last iterative step, thus, the inner product related to the residual vectors can be represented by several inner products which can be computed with vectors of last iterative step or can be computed in the last synchronization points. Thus it reduces the synchronization points. Such a method has been considered in a sequent publications [2, 14, 15, 22, 25, 33, 36, 37, 38, 42]. Because such a strategy focuses on the algorithm rather than one particular hardware, it is suitable for all kinds of distributed machines and is easy to co-operate with all other code optimization methods to improve the overall performance of the iterative solvers. Such a residual split results a mathematically equivalent version of the underlying Krylov methods. While in practice, the convergence of the re-designed methods maybe a little poorer. The reason for such phenomenon will be explained and remedies will also be proposed.

4. Case Study

4.1. Original Algorithm

A hybrid generalized product-type methods based on Bi-CG is considered to show how to redesign a Krylov methods suitable for heterogenous distributed supercomputers. The method is called GPBiCG(m, ℓ) [13] (see Algorithm 1). It includes 3 other well-known Krylov methods when the parameters m and ℓ are chosen differently (see Table 2). The GPBiCG$(1, 0)$ is reduce to the well-known BiCGSTAB method [32]. As known, BiCGSTAB can encounter breakdown. The hybrid BiCGSTAB2 method [19] is supposed to reduce the possibility of breakdown by changing the search direction every other iterate. While the GPBiCG method is supposed to be more robust by increasing the search direction space [40], in which, the residual r_k in line 16 of Algorithm 1 and the approximation solution x_k in line 19 of Algorithm 1 is updated by three vectors other than two vectors in other Krylov methods. The idea of BiCGSTAB2 and GPBiCG suggests that it is very likely to develop breakdown-free Krylov methods by dynamically changing the search direction and increasing the searching space. Therefore the method is a idea representee of Krylov methods and it is an idea candidate to investigate performance of its corresponding parallel version. The parallel version of the methods was first considered in [41].

Table 2: The equivalent methods of GPBiCG(m, ℓ) method.

GPBiCG(m, ℓ)	equivalent method	reference
GPBiCG(1,0)	BiCGSTAB	[32]
GPBiCG(1,1)	BiCGSTAB2	[19]
GPBiCG(0,1)	GPBiCG	[40]

Algorithm 1 GPBiCG(m, ℓ)

1: $r_0 = b - Ax_0$, $t_{-1} = w_{-1} = 0$,
2: **for** $k = 0, 1, ...,$ until $\|r_k\| \leq tol$ **do**
3: $p_k = r_k + \beta_{k-1}\,(p_{k-1} - u_{k-1})\,;\; q_k = Ap_k$
4: $\alpha_k = \frac{(r_0^*, r_k)}{(r_0^*, q_k)}$
5: $t_k = r_k - \alpha_k q_k;\; s_k = At_k$
6: $y_k = t_{k-1} - t_k - \alpha_k w_{k-1}$
7: **if** $(mod(k,\ m+l) < m$ or $k = 0)$ **then**
8: $\zeta_k = \frac{(s_k, t_k)}{(s_k, s_k)}$
9: $u_k = \zeta_k q_k$
10: $z_k = \zeta_k r_k - \alpha_k u_k$
11: $r_{k+1} = t_k - \zeta_k s_k$
12: **else**
13:

$$\eta_k = \frac{(s_k,s_k)(y_k,t_k)-(y_k,s_k)(s_k,t_k)}{(s_k,s_k)(y_k,y_k)-(y_k,s_k)(s_k,y_k)}$$
$$\zeta_k = \frac{(y_k,y_k)(s_k,t_k)-(y_k,t_k)(s_k,y_k)}{(s_k,s_k)(y_k,y_k)-(y_k,s_k)(s_k,y_k)}$$

14: $u_k = \zeta_k q_k + \eta_k\,(t_{k-1} - r_k + \beta_{k-1} u_{k-1})$
15: $z_k = \zeta_k r_k + \eta_k z_{k-1} - \alpha_k u_k$
16: $r_{k+1} = t_k - \eta_k y_k - \zeta_k s_k$
17: **end if**
18: $\beta_k = \frac{\alpha_k}{\zeta_k} \frac{(r_0^*, r_{k+1})}{(r_0^*, r_k)}$
19: $x_{k+1} = x_k + \alpha_k p_k + z_k$
20: $w_k = s_k + \beta_k q_k$
21: **end for**

4.2. Data Dependence Analysis and Algorithm Redesign

The section focuses on how to eliminate the data dependence and reduce synchronization points via *residual split*,*loop shift* and *transpose shift*.

It is noted that there are three synchronization points in Algorithm 1. Global communication are required in line 4, line 8 or line 13, and line 18 in this algorithm. Further the computation of ζ_k depends on the vectors s_k and t_k which themselves depends on α_k. While β_k in line 18 depends on the residual r_{k+1} which depends η_k and ζ_k. Reducing the number of synchronization points need to eliminate the data dependence at first.

4.2.1. Residual Split

One trick to breakdown such data dependence is *residual split*. It is noted that the residual r_{k+1} in line 16 can be replaced by vectors in last iterate. Therefore the inner product (r_0^*, r_{k+1}) can be computed indirectly by

$$\widetilde{rr}_{k+1} := (r_0^*, r_{k+1}) = (r_0^*, t_k) - \eta_k(r_0^*, y_k) - \zeta_k(r_0^*, s_k). \tag{2}$$

In this way, two more inner products have to be computed, but the increasing inner products can be computed in last synchronization point and thus one synchronization point is reduced.

4.2.2. Loop Shift and Transpose Shift

We call another trick to further breakdown data dependence and reduce synchronization points *loop shift*. Suppose every loop has only 3 operation statements say, A, B, C. Then one possible loop pattern is $\{A, B, C\}, \{A, B, C\}, \ldots$. This loop can be shifted as another one, $A, \{B, C, A\}, \{B, C, A\} \ldots$. According to such a loop shift, the information in line 3 to line 20 of Algorithm 1 can be arranged as line 5 to line 20 and then line 3 and line 4. In this way, α_{k+1} is considered instead of α_k for each loop. And the synchronization point to compute α_{k+1} is also expected to be reduced.

It is noted that (r_0^*, q_{k+1}) is necessary for computing α_{k+1} while the update of q_{k+1} depends on a matrix-vector multiplication. If q_{k+1} is updated by the residual split technique directly, it results in another two matrix-vector

multiplications, which is not what we expect. Luckily, the symmetric property of inner product can be employed to avoid such un-necessary increasing the number of matrix vector multiplications.

$$(r_0^*, q_{k+1}) = (r_0^*, Ap_{k+1}) = (A^T r_0^*, p_{k+1}) = (f, p_{k+1}), \tag{3}$$

where $f = A^T r_0^*$. We call such a transform *transpose shift*. It should be pointed out that the transpose of matrix should be avoided to use as possible due to large data movements. Luckily, here the transpose only used once. And then the inner product (f, p_{k+1}) can be computed by cheap vector updates and residual placement techniques. Form $p_{k+1} = r_{k+1} + \beta_k(p_k - u_k)$, we get

$$\widetilde{fp}_{k+1} := (f, p_{k+1}) = (f, r_{k+1}) + \beta_k \left((f, p_k) - (f, u_k)\right). \tag{4}$$

Similarly as (2), the (f, r_{k+1}) can be computed as

$$\widetilde{fr}_{k+1} := (f, r_{k+1}) = (f, t_k) - \eta_k (f, y_k) - \zeta_k (f, s_k). \tag{5}$$

Let $h_k = t_{k-1} - r_k + \beta_{k-1} u_{k-1}$ (see line 14 in Algorithm 1), then (f, u_k) could be formulated as

$$\widetilde{fu}_k := (f, u_k) = \begin{cases} \zeta_k (f, q_k) & \text{if execute line 9,} \\ \zeta_k (f, q_k) + \eta_k (f, h_k) & \text{if execute line 14.} \end{cases} \tag{6}$$

Equation(4)-(6) show that (f, p_{k+1}) can be indirectly computed by (f, t_k), (f, y_k), (f, s_k),(f, q_k) and (f, h_k). These inner products can computed as the same time when computing inner products associated with ζ_k and η_k in line 8 or line 13 in Algorithm 1.

As discussed above, the 3 global synchronization points in Algorithm 1 can be reduced only one by residual split, loop shift and transpose shift. The resulted improved algorithm is presents in Algorithm 2.

Algorithm 2 Parallel GPBiCG(m, ℓ)

1: $r_0 = b - Ax_0$, $t_{-1} = w_{-1} = \vec{0}$, $f = A^T r_0$,$p_0 = r_0$, $q_0 = Aq_0$, $\alpha = (r_0, r_0)/(f, p_0)$
2: **for** $k = 0, 1, ...,$ until convergence **do**
3: $tem = t_{k-1}$
4: $t_k = r_k - \alpha_k q_k$;
5: $s_k = At_k$
6: $y_k = tem - t_k - \alpha_k w_{k-1}$
7: **if** mod$(k,\ m + l) < m$ or $k = 0$ **then**
8: compute (s_k, t_k), (s_k, s_k), (r_0, t_k), (r_0, s_k), (f, s_k), (f, t_k), (f, q_k), (f, p_k)
9: $\zeta_k = \frac{(s_k, t_k)}{(s_k, s_k)}$, $\eta_k = 0$
10: $\widetilde{rr}_{k+1} = (r_0^*, t_k) - \zeta_k(r_0^*, s_k)$
11: $\widetilde{fu}_k = \zeta_k(f, q_k)$
12: $\widetilde{fr}_{k+1} = (f, t_k) - \zeta_k(f, s_k)$
13: $u_k = \zeta_k q_k$
14: $z_k = \zeta_k r_k - \alpha_k u_k$
15: $r_{k+1} = t_k - \zeta_k s_k$
16: **else**
17: $h_k = tem - r_k + \beta_{k-1} u_{k-1}$
18: compute $(s_k, s_k), (s_k, y_k), (s_k, t_k), (y_k, t_k), (y_k, y_k), (r_0^*, t_k), (r_0^*, y_k), (r_0^*, s_k), (r_0^*, r_k), (f, t_k), (f, y_k), (f, s_k), (f, h_k), (f, q_k), (f, p_k),$
19:
$$\eta_k = \frac{(s_k, s_k)(y_k, t_k) - (y_k, s_k)(s_k, t_k)}{(s_k, s_k)(y_k, y_k) - (y_k, s_k)(s_k, y_k)},$$
$$\zeta_k = \frac{(y_k, y_k)(s_k, t_k) - (y_k, t_k)(s_k, y_k)}{(s_k, s_k)(y_k, y_k) - (y_k, s_k)(s_k, y_k)}$$
20: $\widetilde{rr}_{k+1} = (r_0^*, t_k) - \eta_k(r_0^*, y_k) - \zeta_k(r_0^*, s_k)$
21: $\widetilde{fu}_k = \zeta_k(f, q_k) + \eta_k(f, h_k)$
22: $\widetilde{fr}_{k+1} = (f, t_k) - \eta_k(f, y_k) - \zeta_k(f, s_k)$
23: $u_k = \zeta_k q_k + \eta_k h_k$
24: $z_k = \zeta_k r_k + \eta_k z_{k-1} - \alpha_k u_k$
25: $r_{k+1} = t_k - \eta_k y_k - \zeta_k s_k$
26: **end if**
27: $\beta_k = \frac{\alpha_k}{\zeta_k} \frac{\widetilde{rr}_{k+1}}{(r_0^*, r_k)}$
28: $x_{k+1} = x_k + \alpha_k p_k + z_k$
29: $w_k = s_k + \beta_k q_k$
30: $p_{k+1} = r_{k+1} + \beta_k (p_k - u_k)$
31: $q_{k+1} = Ap_{k+1}$
32: $\widetilde{fp}_{k+1} = \widetilde{fr}_{k+1} + \beta_k((f, p_k) - \widetilde{fu}_k)$
33: $\alpha_{k+1} = \frac{\widetilde{rr}_{k+1}}{\widetilde{fp}_{k+1}}$
34: **end for**

5. Performance Analysis

Table 3: The main computations of GPBiCG(m, ℓ) and PGPBiCG(m, ℓ)

Methods	vector update				inner product				matrix-vector	synchronization points
	H	m	ℓ	T	H	m	ℓ	T		
GPBiCG(m, ℓ)	5	3	7	3	1	2	5	2	2	3
PGPBiCG(m, ℓ)	5	3	7	3	0	9	15	0	2	1

The **for** loops in Algorithm 1 and Algorithm 2 are divided into 4 parts: the part before **if** denotes as heard part (H); **if** branch denote as m part, and **else** branch denoted as ℓ part; and the remaining part denotes as tail part (T).

This section focuses on the performance of the original version of Algorithm 1 and the parallel improved version Algorithm 2. Computations in the two algorithms, matrix-vector multiplications, inner products and vector updates are listed in Table 3.

5.1. Time Complexity Analysis

Table 3 compares the number of computation kernels and the number of global synchronization points in Algorithm 1 and Algorithm 2. As shown the redesigned algorithm has 4 to 5 more inner products in average but all the inner products only need one global synchronization point. According to Table 1, k inner products computed at the same synchronization point need

$$t_{inn(k)} = \frac{2kNt_{fl}}{P} + 2\left(t_s + kt_w\right)\omega\left(p\right) = kt_{vec} + 2\left(t_s + kt_w\right)\log_2\left(P\right) \tag{7}$$

computation and communication time. Since the set up time is far bigger than the message passing time, say, $t_s \gg t_w$. Thus reducing the number of global synchronization points can reduce the global communication time. Table 4 details the time needed for each synchronization point in the two Algorithm.

Table 4: The time of inner products computation in GPBiCG(m, ℓ) and PGPBiCG(m, ℓ)

Methods	Position	No.	Time
GPBiCG(m, ℓ)	H	1	$2t_{fl}N/P + 2\log P\left(t_s + t_w\right)$
	M	2	$4t_{fl}N/P + 2\log P\left(t_s + 2t_w\right)$
	L	5	$10t_{fl}N/P + 2\log P\left(t_s + 5t_w\right)$
	T	2	$4t_{fl}N/P + 2\log P\left(t_s + 2t_w\right)$
PGPBiCG(m, ℓ)	M	9	$18t_{fl}N/P + 2\log P\left(t_s + 9t_w\right)$
	L	15	$30t_{fl}N/P + 2\log P\left(t_s + 15t_w\right)$

Different with many other Krylov subspace methods, the search directions in GPBiCG(m, ℓ) dynamically change, it is difficult to evaluate the expect solution time for such an algorithm based on only one iterate step. Because the number of vector updates and the inner products in **if** branch and **else** branch are different. Furthermore, the communication time of inner product doesn't linearly depend on the number of inner products, therefore we can not use the average number in the two branches. We have to view the $m + \ell$ iterates as a "unit loop". Counting the number of kernels of Krylov methods in Algorithm 1, and substituting the time for each kernels into equation 1, we get a concrete formula for the expected time

$$T^{\text{GPBiCG}(m,\ell)} = \{8(m+\ell) + 3m + 7\ell\}t_{vec} + 2(m+\ell)t_{mv} + (m+\ell)(t_{inn(1)} + t_{inn(2)}) + mt_{inn(2)} + \ell t_{inn(5)}, \tag{8}$$

where t_{vec} and t_{mv} is the time for vector update and matrix-vector multiplication given in Table 1, $t_{inn(k)}$ is given in equation (7). The formulae in equation (8) can be simplified as

$$T^{\text{GPBiCG}(m,\ell)} = \frac{\lambda_1}{P} + \lambda_2 \log_2 P + 2\left(m + l\right)\mu, \tag{9}$$

where μ is the local communication time defined in Table 1 and

$$\lambda_1 = \{32m + 46l + 2(m + l)n_z\}Nt_{fl}, \tag{10}$$

$$\lambda_2 = 6(m + l)t_s + (10m + 16l)t_w. \tag{11}$$

Similarly, the expect time of one "unit loop" in PGPBiCG(m, ℓ) is

$$T^{\text{PGPbiCG}(m,\ell)} = \frac{\sigma_1}{P} + \sigma_2 \log_2 P + 2(m + l)\mu, \tag{12}$$

where

$$\sigma_1 = \{40m + 60l + 2(m + l)n_z\}Nt_{fl}, \tag{13}$$

$$\sigma_2 = 2(m + l)t_s + (18m + 30l)t_w. \tag{14}$$

When solving the same problem using the same number of processors, the improvement of solution time of PGPBiCG(m, ℓ) compared with GPBiCG(m, ℓ) methods is

$$\eta = \frac{T^{\text{GPBiCG}(m,\ell)} - T^{\text{PGPBiCG}(m,\ell)}}{T^{\text{GPBiCG}(m,\ell)}} = \frac{(\lambda_2 - \sigma_2)P\log_2 P + \lambda_1 - \sigma_1}{\lambda_2 P\log_2 P + 2(m + l)\mu P + \lambda_1} \tag{15}$$

$$= \frac{\{4(m + l)t_s - (8m + 12l)t_w\}P\log_2 P - (8m + 14l)Nt_{fl}}{\{6(m + l)t_s + (10m + 16l)t_w\}P\log_2 P + 2(m + l)\mu P + \lambda_1}. \tag{16}$$

Because $t_s \gg t_w$, when P is large enough, $\eta \to \frac{2}{3}$.

Conclution 1. *When solving the same problem using the same number of processor, the improvement of solution time using PGPBiCG(m, ℓ) compared with using GPBiCG(m, ℓ) is about two third.*

5.2. Scalability Analysis

Speedup and efficiency are two indicators to evaluate the performance of parallel algorithm. *Speedup* is defined as

$$S = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{T_S}{T_P}, \tag{17}$$

whereT_S and T_P represents for time to solve a problem on a single processor and on P processors, respectively. The *efficiency*is defined as

$$E = \frac{\text{speedup}}{\text{number of processors}} = \frac{S}{P} = \frac{T_S}{PT_P}. \tag{18}$$

Such definition of speed up has its limitation, because the speedup reaches the maxima at certain P^*, and S_P will become smaller with P increasing. In practice, the scaled speedup is often used to access the scalability of the parallel algorithm.

5.2.1. Scaled Speedup

The ideal speedup satisfies that the solution time will be a constant when the number of processors P and the problem size N increasing at the same rate. Let $T\left(\frac{N}{P}, P\right)$ be the solution time£if $\frac{N}{P}$ is a constant, we denote the solution time as $T(\cdot, P)$, and denote $T(\cdot, 1)$ as the solution time on single processor. then the *weakly scaled speed up* is defined as

$$S_P^C = \frac{T_{\text{Serial}}}{T_{\text{Parallel}}} = \frac{T(\cdot, 1)}{T(\cdot, P)} \tag{19}$$

The ideal scaled speedup is 1. However such ideal case seldom appears because the communications and the serial computations in existence. The S_P^C is more approximate to 1, the scalability is much better. Let $S_P^{\text{GPBiCG}(m,\ell)}$ and $S_P^{\text{PGPBiCG}(m,\ell)}$ be the scaled speedup of GPBiCG(m, ℓ) and PGPBiCG(m, ℓ) respectively. Now we derive the formula

of $S_P^{\mathrm{GPBiCG}(m,\ell)}$.

$$T^{\mathrm{GPBiCG}(m,\ell)} = \frac{\lambda_1}{N}\frac{N}{P} + \lambda_2 \log_2 P + 2(m+l)\mu = \tilde{\lambda}_1 C + \lambda_2 \log_2 P + 2(m+l)\mu \tag{20}$$

where $\frac{N}{P} = Constant$ is the problem size on each processor and

$$\tilde{\lambda}_1 = \frac{\lambda_1}{N} = \{32m + 46l + 2(m+l)n_z\}\, t_{fl} \tag{21}$$

then the scaled speedup of GPBiCG(m, ℓ)is

$$S_P^{\mathrm{GPBiCG}(m,\ell)} = \frac{T^{\mathrm{GPBiCG}(m,\ell)}(\cdot, 1)}{T^{\mathrm{GPBiCG}(m,\ell)}(\cdot, P)} = \frac{\tilde{\lambda}_1 C}{\tilde{\lambda}_1 C + \lambda_2 \log_2 P + 2(m+l)\mu} = \frac{1}{1 + \frac{\lambda_2 \log_2 P}{\tilde{\lambda}_1 C} + \frac{2(m+l)\mu}{\tilde{\lambda}_1 C}} \tag{22}$$

Similarly, the scaled speedup of PGPBiCG(m, ℓ)is .

$$S_P^{\mathrm{PGPBiCG}(m,\ell)} = \frac{1}{1 + \frac{\sigma_2 \log_2 P}{\tilde{\sigma}_1 C} + \frac{2(m+l)\mu}{\tilde{\sigma}_1 C}} \tag{23}$$

where

$$\tilde{\sigma}_1 = \frac{\sigma_1}{N} = \{40m + 60l + 2(m+l)n_z\}\, t_{fl}. \tag{24}$$

The ratio of the scaled speedup is

$$\frac{S_P^{\mathrm{PGPBiCG}(m,\ell)}}{S_P^{\mathrm{GPBiCG}(m,\ell)}} = \frac{\frac{\lambda_2 \log_2 P}{\tilde{\lambda}_1 C} + \frac{2(m+l)\mu}{\tilde{\lambda}_1 C} + 1}{\frac{\sigma_2 \log_2 P}{\tilde{\sigma}_1 C} + \frac{2(m+l)\mu}{\tilde{\sigma}_1 C} + 1} \tag{25}$$

Because$t_s \gg t_w$, when P is large enough,we have

$$\frac{S_P^{\mathrm{PGPBiCG}(m,\ell)}}{S_P^{\mathrm{GPBiCG}(m,\ell)}} \approx \frac{\lambda_2}{\sigma_2}\frac{\tilde{\sigma}_1}{\tilde{\lambda}_1} \approx \frac{6(m+l)t_s}{2(m+l)t_s} \cdot \frac{20m + 30l + (m+l)n_z}{16m + 23l + (m+l)n_z} > 3. \tag{26}$$

Conclution 2. *When the processors is large enough, the scaled speedup of PGPBiCG(m, ℓ) is more than 3 times better than that of GPBiCG(m, ℓ).*

5.2.2. Iso-efficiency Analysis

Another way to assess the scalability of the parallel algorithm is *iso-efficiency*. The efficiency defined in (18) in fact depends the problem size N. Keep E as a constant, we can derive a formula which is called the *iso-efficiency function*

$$N = f_E(P) \text{ where } (E \text{ is a constant}). \tag{27}$$

The iso-efficiency function demonstrates that relationship between the problem size N and the number of processors P to keep a constant efficiency. The function value is smaller, the scalability of the algorithm is much better[39, pp. 217].

In the idea case, The efficiency $E = S/P = T_S/(PT_P)$ should be 1 or approximates to 1, where S is speedup given by (17). However, the ideal efficiency and speedup seldom can be achieved because there are communications and serial instructions. The time due to communication time and serial instructions is called *overhead* or *overhead function* given by

$$T_{\mathrm{over}}(N, P) = PT_P(N, P) - T_S(N). \tag{28}$$

Employing such a definition, another formula of efficiency is introduced as

$$E = \frac{T_S(N)}{PT_P(N, P)} = \frac{T_S(N)}{T_S(N) + T_{over}(N, P)} = \frac{1}{1 + \frac{T_{over}(N,P)}{T_S(N)}}. \tag{29}$$

From equation (29), the following relationship holds

$$T_S(N) = \frac{E}{1-E} T_{over}(N,P), \tag{30}$$

where E is constant.

Denote

$$T^{\mathrm{GPBiCG}(m,\ell)} = T_{\mathrm{comp}}^{\mathrm{GPBiCG}(m,\ell)} + T_{\mathrm{comm}}^{\mathrm{GPBiCG}(m,\ell)}, \tag{31}$$

where $T_{\mathrm{comp}}^{\mathrm{GPBiCG}(m,\ell)}$ and $T_{\mathrm{comm}}^{\mathrm{GPBiCG}(m,\ell)}$ represent the computation time and communication time respectively. Since the time for serial instructions (like line 20 to line 22 in Algorithm 2) can be neglect. Thus

$$T_{over}^{G} = PT_{\mathrm{comm}}^{\mathrm{GPBiCG}(m,\ell)}. \tag{32}$$

One the other hand, form (9)

$$T_{\mathrm{comm}}^{\mathrm{GPBiCG}(m,\ell)} = \lambda_2 \log_2 P + 2(m+l)\mu \tag{33}$$

where λ_2 is given in (10). Substituting equation (32) and (33) into equation(30)) gives

$$T_S^{\mathrm{GPBiCG}(m,\ell)}(N) = \frac{EP\{\lambda_2 \log_2 P + 2(m+l)\mu\}}{1-E}. \tag{34}$$

On the other hand, by definition

$$T_S^{\mathrm{GPBiCG}(m,\ell)}(N) = \{32m + 46l + 2(m+l)n_z\} Nt_{fl} = \tilde{\lambda}_1 N \tag{35}$$

where $tilde\lambda_1$ is given in (21). Form above analysis, the iso-efficiency function of GPBiCG(m,ℓ) methods is

$$N^{\mathrm{GPBiCG}(m,\ell)} = \frac{EP(\lambda_2 \log_2 P + 2(m+l)\mu)}{\tilde{\lambda}_1(1-E)} = \frac{\lambda_2}{\tilde{\lambda}_1}\frac{EP\log_2 P}{(1-E)} + \frac{2(m+l)\mu EP}{\tilde{\lambda}_1(1-E)}. \tag{36}$$

where λ_1 is given in (10) and $\tilde{\lambda}_1$ is given in (21).

Similarly, the iso-efficiency function of PGPBiCG(m,ℓ) is

$$N^{\mathrm{PGPBiCG}(m,\ell)} = \frac{\sigma_2}{\tilde{\sigma}_1}\frac{EP\log_2 P}{(1-E)} + \frac{2(m+l)\mu EP}{\tilde{\sigma}_1(1-E)} \tag{37}$$

where $\tilde{\sigma}_1$ is given in (24) and σ_1 is given in (13).

In the formula of $N^{\mathrm{GPBiCG}(m,\ell)}$ and $N^{\mathrm{PGPBiCG}(m,\ell)}$, the second terms include μ which represents for the (local) communication time of matrix vector multiplication. It usually can be overlapped with other useful computations. When number of processor P is large, $N^{\mathrm{GPBiCG}(m,\ell)}$ and $N^{\mathrm{GPBiCG}(m,\ell)}$ are dominated by the leading order of P

$$N^{\mathrm{GPBiCG}(m,\ell)} \approx \frac{\lambda_2}{\tilde{\lambda}_1}\frac{EP\log_2 P}{(1-E)} \quad \text{and} \quad N^{\mathrm{PGPBiCG}(m,\ell)} \approx \frac{\sigma_2}{\tilde{\sigma}_1}\frac{EP\log_2 P}{(1-E)}. \tag{38}$$

The iso-efficiency function is smaller, the scalability of the algorithm is much better. Since $\frac{\lambda_2}{\sigma_2}\frac{\tilde{\sigma}_1}{\tilde{\lambda}_1} > 3$, then we could also get that the scalability of PGPBiCG(m,ℓ) is more than 3 times better than that of GPBiCG(m,ℓ). See Figure 2 for a illustration.

5.2.3. The Optimal Number of Processors

One main goal of scalability analysis is to determine the optimal performance of a given parallel algorithm on prescribed computing platform.When the problem size is given, the optimal number of processors can be determined via performance analysis. One can obtain the optimal number of processors by minimizing the following function.

$$\varphi(x) = \frac{\theta_1}{x} + \theta_2 \log_2 x + C, \quad \theta_1, \theta_2 > 0, \quad C \text{ is a constant} \tag{39}$$

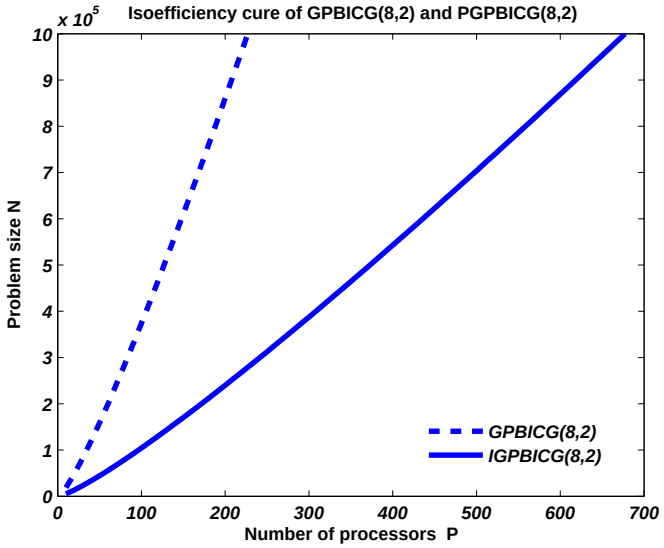


Figure 2: Iso-efficiency curves of GPBiCG(8,2) and PGPBiCG(8,2). where $E = 80\%$, $t_s = 10\mu s$, $t_w = 20ns$, $t_{fl} = 10ns$, $n_z = 5$. The figure shows that to keep the efficiency as 80%, GPBiCG(8, 2) can only use 220 processor for a problem with 100,000 unknowns, while PGPBiCG(8, 2) method can use three times more processors, about 680 in the figure. Since they have the same efficiency, thus the scalability of the redesign version is apparently better than that of GPBiCG(m, ℓ).

Let$\varphi'(x) = 0$, we could find the only stationary point of φ in $(0, \infty)$ is $x = \frac{\theta_1 \ln 2}{\theta_2}$. Further more, $\varphi''\left(\frac{\theta_1 \ln 2}{\theta_2}\right) > 0$, and hence $x = \frac{\theta_1 \ln 2}{\theta_2}$ is the minimal point of the function in $(0, \infty)$. Applying this conclusion to formula (9) and (12), one finds the optimal number of processors for GPBiCG(m, ℓ) and PGPBiCG(m, ℓ) respectively:

$$P_{\text{opt}}^{\text{GPBiCG}(m,\ell)} = \frac{\{32m + 46l + 2(m + l) n_z\} N t_{fl} \ln 2}{6(m + l) t_s + (10m + 16l) t_w} \tag{40}$$

$$P_{\text{opt}}^{\text{PGPBiCG}(m,\ell)} = \frac{\{40m + 60l + 2(m + l) n_z\} N t_{fl} \ln 2}{2(m + l) t_s + (18m + 30l) t_w} \tag{41}$$

Because $t_s \gg t_w$, then $\frac{P_{\text{opt}}^{\text{PGPBiCG}(m,\ell)}}{P_{\text{opt}}^{\text{GPBiCG}(m,\ell)}} > 3$.

Conclution 3. *The optimal number of processors when using PGPBiCG(m, ℓ) is 3 times larger than that when using GPBiCG(m, ℓ).*

5.3. Convergence Analysis

It is noted that the residual split technique can hinder the convergence rate of the recontracted algorithms. Usually, this is because the indirectly computing inner product can accumulate the error due to round off. Consider the following estimate of the error due to round off when computing a inner product.

Lemma 4 ([20, p.75])**.** *If $x, y \in \mathbb{R}^n$, let $fl\left(x^T y\right)$ is the inner product computed by float points computations, $x^T y$ is the true value, $\delta = fl\left(x^T y\right) - x^T y$, then*

$$|\delta| \leq 1.01 N \boldsymbol{u} \sum_{i=1}^{N} |x_i y_i|$$

where $\boldsymbol{u}$ is the machine precision.

Lemma 4 shows that $|\delta|$ can be far larger than the machine precision $\boldsymbol{u}$ when N is extremely large. In most of the case, no mater whether the matrix is sparse or not. the solution x and the residual r_k is usually dense. This brings

another challenging issue: accurate compute the inner product for large dense vectors, which go beyond our discussion here. The reader is directed to [3][9] [20, Chap.3][28][30].

Indirect computing a inner product by several other inner products brings new error sources, thus makes the inner product less accurate. Therefore the convergence performance of the reconstructed algorithm may be poor.

In Algorithm 2, notice that $t_k = r_k - \alpha_k q_k$ in line 4, then the inner product $\widetilde{fr}_{k+1}$ in line 12 and line 22 and inner product $\widetilde{fp}_{k+1}$ in line 32 can be computed by a recursive way:

$$\widetilde{fr}_{k+1} = \widetilde{fr}_k - a_k(f, q_k) - \eta_k(f, y_k) - \zeta_k(f, s_k) \tag{42}$$

$$\widetilde{fp}_{k+1} = \widetilde{fr}_{k+1} + \beta_k\left(\widetilde{fp}_k - (f, u_k)\right) \tag{43}$$

In such a recursive way, if the inner product $(f, r_0) = (f, p_0)$ is provided, then the inner products $\widetilde{fr}_{k+1}$ and $\widetilde{fp}_{k+1}$ can be computed in recursive. By such recursive formula, two less inner products would be computed in line 8 and line 18. This is exactly the same way Yang used in the improved BiCGSTAB [37]. While in Algorithm 2 the inner products (f, r_k) and (r_0^*, r_k) are updated directly for every iterate, which reduces the potential error source. A further analysis the accuracy of these two ways to compute the inner product shows that the recursive way to compute the inner product can accumulate the round off error.

To make the analysis simply, we only consider the GPBiCG$(1, 0)$ method which equivalent to BiCGSTAB method. The inner product (r_0^*, r_{k+1}) are compute in PGPBiCG$(1, 0)$ and IBiCGSTAB method in the following way respectively:

$$\text{PGPBiCG}(1,0): \quad \widetilde{rr}_{k+1} = (r_0^*, t_k) - \zeta_k(f, s_k); \tag{44}$$

$$\text{BiCGSTAB}: \quad \widetilde{rr}_{k+1} = \widetilde{rr}_k - \alpha_k(r_0^*, q_k) - \zeta_k(r, s_k). \tag{45}$$

Denote the error of the inner product (r_0^*, r_{k+1}) according to (44)and (45)) as $\delta^{\text{PGPBiCG}(1,0)}_{\widetilde{rr}_{k+1}}$ and $\delta^{\text{IBiCGSTAB}}_{\widetilde{rr}_{k+1}}$ respectively, then according to Lemma 4

$$\delta^{\text{PGPBiCG}(1,0)}_{\widetilde{rr}_{k+1}} = fl(\widetilde{rr}_{k+1}) - r_0^T r_{k+1} = fl\left(r_0^T, t_k\right) - \zeta_k fl\left(f^T s_k\right) - r_0^T r_{k+1} = \delta_{r^T t_k} - \zeta_k \delta_{f^T s_k} \tag{46}$$

and thus we have

$$\left|\delta^{\text{PGPBiCG}(1,0)}_{\widetilde{rr}_{k+1}}\right| \le \left|\delta_{r_0^T t_k}\right| + |\zeta_k|\left|\delta_{f^T s_k}\right|. \tag{47}$$

While according to (45) in IBiCGSTAB to compute $\widetilde{rr}_{k+1}$,then error

$$\begin{cases} \delta^{\text{IBiCGSTAB}}_{\widetilde{rr}_1} = \delta_{r_0^T r_0} - \alpha_0 \delta_{r_0^T q_0} - \zeta_0 \delta_{r_0^T s_0}; \\ \delta^{\text{IBiCGSTAB}}_{\widetilde{rr}_{k+1}} = \delta^{\text{IBiCGSTAB}}_{\widetilde{rr}_k} - \alpha_k \delta_{r^T q_k} - \zeta_0 \delta_{r_0^T s_k}. \end{cases} \tag{48}$$

Combining the above recursive formulae, one can get

$$\left|\delta^{\text{IBiCGSTAB}}_{rr_{k+1}}\right| \le \left|\delta_{r_0^T r_0} - \sum_{i=0}^{k}\left(\alpha_i \delta_{r_0^T q_i} + \zeta_i \delta_{r_0^T s_i}\right)\right| \le \left|\delta_{r_0^* r_0}\right| + \sum_{i=0}^{k}\left(|\alpha_i|\left|\delta_{r_0^* q_i}\right| + |\zeta_i|\left|\delta_{r_0^T s_i}\right|\right) \tag{49}$$

Lemma 4 indicates that these errors $\delta_{r_0^T r_0}$, $\delta_{r_0^T, q_i}$, $\delta_{r_0^T s_i}$, $\delta_{r_0^T t_k}$, $\delta_{f^T s_k}$ can be much larger than machine precision. It is clear from (49) that the recursive way to compute the inner product can accumulate the round off error.

Conclution 5. *The round error of inner product of large dimension vectors could be much bigger than machine precision, the recursive way to compute the inner product make these relative big errors accumulated quickly and result in large error which hinders the stability of the algorithm. Therefore, The recursive way to compute inner products indirectly should be avoided.*

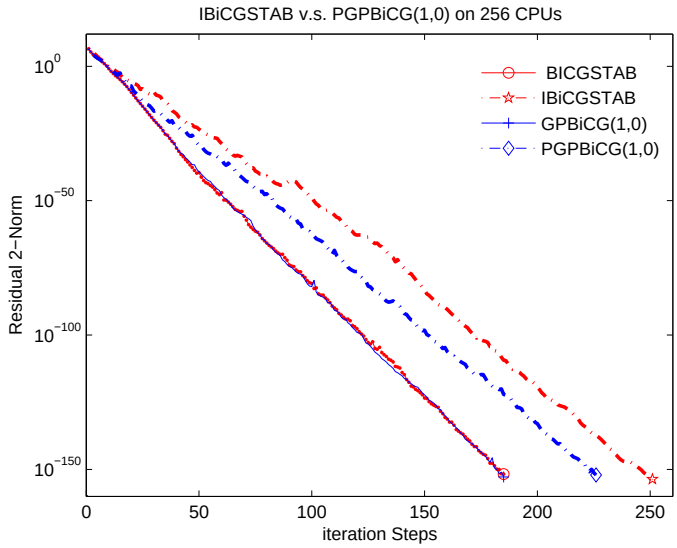


Figure 3: The convergence performance of GPBiCG(1,0), PGPBiCG(1,0), BiCGSTAB and IBiCGSTAB using 256 processors for a problem with 921,600 unknowns. The figure demonstrates the convergence of the original versions of the two equivalent methods is almost the same. While for the parallel versions, the one without recursive inner product computation converges faster.

6. Numerical experiments

The test problem considered in this paper is as follows

$$a\frac{\partial^2 u}{\partial x^2} + b\frac{\partial^2 u}{\partial y^2} + c\frac{\partial u}{\partial x} + d\frac{\partial u}{\partial y} + eu = 0, \quad x, y \in (0,1), \tag{50}$$

with boundary conditions

$$\begin{cases} \frac{\partial u}{\partial x}|_{x=0} = 0, & \frac{\partial u}{\partial x}|_{x=1} = 0 \\ u|_{y=0} = 0, & u|_{y=1} = 10, \end{cases} \tag{51}$$

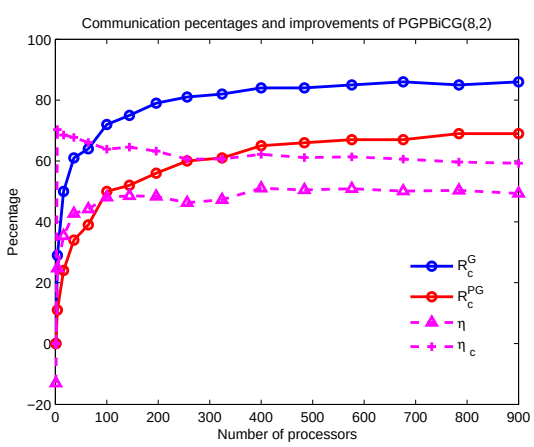
where $a = b = 1512.0$, $c = d = 1.0$, $e = 0.0$. A special nine-point difference scheme was used to approximate the test problem. Therefore the average non-zero elements of the matrix is 9, $n_z = 9$. This is not a challenging problem, but as mentioned before, the paper focuses on the performance of Krylov methods on distributed heterogenous computers rather then other challenging applications. The numerical experiments of GPBiCG(m, ℓ) and PGPBiCG(m, ℓ) run a heterogenous supercomputer: Dawning 5000A. Each result is an average of results from 4 times running.

First,the problem size on each processor is fixed as $3,600$, say $\frac{N}{P} = 3,600$, We record the solution time and the communication time for 3000 iterates for each method. T^G and T^{PG} stand for the wall-time of entire solution of GPBiCG(m, ℓ) and PGPBiCG(m, ℓ) respectively, while T^G_{comm} (or T^G_c) and T^{PG}_{comm} (or T^{PG}_c) denote their communication time. R^G_c and R^{PG}_c denote the ratio of communication time to entire time.

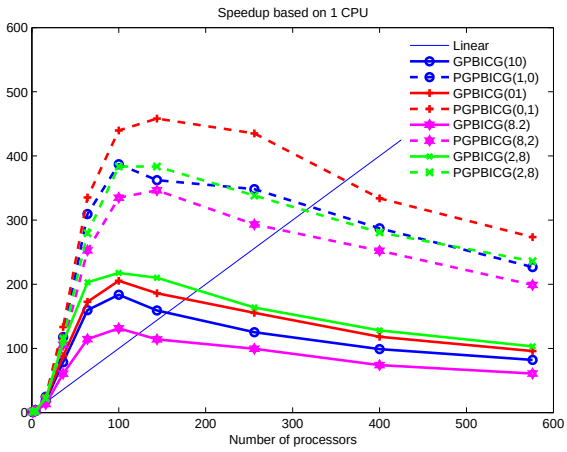
$$\eta = \frac{T_G - T_{PG}}{T_G} \times 100 \quad \text{and} \quad \eta_c = \frac{T^G_{comm} - T^{PG}_{comm}}{T^G_{comm}} \times 100 \tag{52}$$

represent the improvement of solution time and communication time respectively.

Figure 1 illustrates how the solution time of GPBiCG(8, 2) and PGPBiCG(8, 2) increase with the number of processors increasing. The figure illustrates that the computation time of the two methods is almost the same, while the communication time in PGPBiCG(8,2) is about one third of that in GPBiCG(8,2). Figure 4(a) illustrates the improvement of solution time and the ratio of communication time to solution time of GPBiCG(8, 2) and PGPBiCG(8, 2). But the communication can take more than a half when the number of processor number is larger than 200. When using up



(a) The ratio of communication time and improvements



(b) Speedup

Figure 4: (a) The ratio of communication time to the wall-time of entire solution and the improvement for GPBiCG(8,2) and PGPBiCG(8,2). (b) Comparison of the speedup of GPBiCG(m, ℓ) and PGPBiCG(m, ℓ) for different (m, ℓ). The problem size is fixed to 960×960.

Table 5: Performance of GPBiCG$(1, 0)$ and PGPBiCG$(1, 0)$

#	#	GPBiCG(1,0)			PGPBiCG(1,0)			Improvement	
CPU	N	T^G	T_c^G	R_c^G	T^{PG}	T_c^{PG}	R_c^{PG}	η	η_c
1	3600	0.48	0.00	0.00	0.53	0.00	0.00	—	—
4	14400	1.08	0.32	0.30	0.88	0.11	0.12	18.76	66.98
16	57600	1.61	0.76	0.47	1.10	0.28	0.25	31.82	63.60
36	129600	2.39	1.40	0.59	1.29	0.41	0.32	46.01	70.56
64	230400	3.02	2.00	0.66	1.75	0.71	0.41	42.08	64.26
100	360000	4.00	2.96	0.74	2.20	1.07	0.48	45.05	63.97
144	518400	5.62	4.34	0.77	2.92	1.55	0.53	47.95	64.35
196	705600	6.59	5.27	0.80	3.36	1.87	0.56	49.01	64.58
256	921600	7.92	6.38	0.81	4.02	2.43	0.60	49.26	61.99
324	1166400	9.32	7.69	0.82	4.78	3.03	0.63	48.79	60.57
400	1440000	10.54	8.82	0.84	5.36	3.42	0.64	49.17	61.20

to 64 processor with GPBiCG$(8, 2)$ method, R_c^G approximate 60%, while using PGPBiCG$(8, 2)$, the processor could be scaled up to 400 and keep the ratio being about 60%. When the number is 900, $\eta \approx 50\%$ while $\eta_c \approx 60\%$. When parameter (m, l) chosen as other values, we have the similar results.

Table 5 to Table 9 report the results of 4 groups of GPBiCG(m, ℓ) and PGPBiCG(m, ℓ), and the results of BiCGSTAB and IBiCGSTAB [37].

The speedup of GPBiCG(m, ℓ) and PGPBiCG(m, ℓ) is also reported. The problem size is fixed as 960×960. Figure 4(b) reports the speedup based on 1 processors. The speed up based on 4, 16 and 36 processors have the similar trends.

As shown in Figure 1, Figure 4(a) and Table 5 to Table 9, the solution time of Krylov solver is quickly dominated by communication time as the number of processors increases. Therefore, minimizing communication for Krylov methods is crucial for current distributed supercomputers.

Finally, we point out that the experiments were implemented on a modified version of the well-known software package AZTEC for massive linear systems. All the experiments can be redone on different distributed computing platforms.

Table 6: Performance of BICGSTAB and IBICGSTAB

#	#	BiCGSTAB			IBiCGSTAB			Improvement	
CPU	N	T^B	T_c^B	R_c^B	T^{IB}	T_c^{IB}	R_c^{IB}	η	η_c
1	3600	0.38	0.00	0.00	0.41	0.00	0.00	—	—
4	14400	1.18	0.51	0.43	0.85	0.15	0.18	28.22	70.30
16	57600	1.76	1.09	0.62	1.00	0.30	0.30	43.34	72.53
36	129600	2.75	1.96	0.71	1.28	0.50	0.39	53.57	74.76
64	230400	3.45	2.61	0.76	1.55	0.67	0.43	54.96	74.40
100	360000	4.81	3.91	0.81	2.21	1.10	0.50	54.10	71.91
144	518400	6.36	5.24	0.82	2.69	1.50	0.56	57.72	71.33
196	705600	7.69	6.66	0.87	3.12	1.84	0.59	59.41	72.34
256	921600	8.95	7.63	0.85	3.79	2.32	0.61	57.66	69.61
324	1166400	11.10	9.75	0.88	4.69	3.08	0.66	57.72	68.42
400	1440000	12.74	11.24	0.88	5.25	3.51	0.67	58.77	68.81

Table 7: Performance of GPBiCG(1, 1) and PGPBiCG(1, 1)

#	#	GPBiCG(1, 1)			PGPBiCG(1, 1)			Improvement	
CPU	N	T^G	T_c^G	R_c^G	T^{PG}	T_c^{PG}	R_c^{PG}	η	η_c
1	3600	0.51	0.00	0.00	0.59	0.00	0.00	—	—
4	14400	1.27	0.38	0.30	0.99	0.11	0.11	21.67	71.20
16	57600	1.69	0.80	0.48	1.18	0.28	0.24	30.35	65.26
36	129600	2.38	1.41	0.59	1.47	0.49	0.33	38.48	65.46
64	230400	3.05	1.97	0.65	1.79	0.66	0.37	41.15	66.23
100	360000	4.04	2.88	0.71	2.25	1.02	0.45	44.33	64.71
144	518400	5.47	4.17	0.76	3.00	1.56	0.52	45.14	64.49
196	705600	6.27	4.98	0.79	3.51	1.90	0.54	44.04	61.82
256	921600	7.64	6.12	0.80	4.05	2.39	0.59	46.95	60.95
324	1166400	9.76	7.92	0.82	5.04	3.10	0.61	47.87	60.91
400	1440000	10.98	9.15	0.83	5.55	3.56	0.64	**49.50**	61.07

Table 8: Performance of GPBiCG(2, 8) and PGPBiCG(2, 8)

#	#	GPBiCG(2, 8)			PGPBiCG(2, 8)			Improvement	
CPU	N	T^G	T_c^G	R_c^G	T^{PG}	T_c^{PG}	R_c^{PG}	η	η_c
1	3600	0.53	0.00	0.00	0.62	0.00	0.00	—	—
4	14400	1.23	0.37	0.30	1.00	0.10	0.10	18.55	74.06
16	57600	1.69	0.82	0.49	1.20	0.30	0.25	29.07	64.02
36	129600	2.47	1.43	0.58	1.65	0.50	0.30	33.39	64.96
64	230400	3.06	1.89	0.62	1.93	0.71	0.37	37.07	62.64
100	360000	4.16	2.95	0.71	2.42	1.12	0.46	41.79	62.24
144	518400	5.43	4.06	0.75	2.95	1.47	0.50	45.72	63.78
196	705600	6.62	5.21	0.79	3.49	1.94	0.56	47.20	62.71
256	921600	8.15	6.51	0.80	4.09	2.34	0.57	49.79	64.07
324	1166400	10.30	8.53	0.83	5.14	3.16	0.61	**50.09**	62.97
400	1440000	11.47	9.64	0.84	5.82	3.81	0.65	**49.32**	60.54

Table 9: Performance of GPBiCG(0, 1) and PGPBiCG(0, 1)

#	#	GPBiCG(0, 1)			PGPBiCG(0, 1)			Improvement	
CPU	N	T^G	T_c^G	R_c^G	T^{PG}	T_c^{PG}	R_c^{PG}	η	η_c
1	3600	0.48	0.00	0.00	0.53	0.00	0.00	—	—
4	14400	1.08	0.32	0.30	0.88	0.11	0.12	18.76	66.98
16	57600	1.61	0.76	0.47	1.10	0.28	0.25	31.82	63.60
36	129600	2.39	1.40	0.59	1.29	0.41	0.32	46.01	70.56
64	230400	3.02	2.00	0.66	1.75	0.71	0.41	42.08	64.26
100	360000	4.00	2.96	0.74	2.20	1.07	0.48	45.05	63.97
144	518400	5.62	4.34	0.77	2.92	1.55	0.53	47.95	64.35
196	705600	6.59	5. y27	0.80	3.36	1.87	0.56	**49.01**	64.58
256	921600	7.92	6.38	0.81	4.02	2.43	0.60	**49.26**	61.99
324	1166400	9.34	7.69	0.82	4.78	3.03	0.63	48.79	60.57
400	1440000	10.54	8.82	0.84	5.36	3.42	0.64	**49.17**	61.20

7. Conclusions

Unlike most of other software development for Krylov methods, which usually focus on the sparse matrix-vector multiplications. The paper discusses why the way of inner product to be computed is crucial for Krylov iterative solvers on heterogeneous supercomputers according to communication complexity and the trends of current technology discussed before. As shown, the scalability of a underlying Krylov algorithm largely depends on the global communication from the following equation,

$$T = \varphi\left(\frac{N}{P}\right) t_{fl} + \phi\left(t_s, t_w\right) \log_2 P + \mu\left(t_s, t_w\right) \tag{53}$$

where the first item stands for the computation time and the second terms represent the global communication time. When the problems size N and the number of processors P increasing at the same rate, the value of $\varphi\left(\frac{N}{P}\right)$ almost have no change. $\mu\left(t_s, t_w\right)$ represents the local communication time for sparse matrix-vector multiplication, which is usually can be overlapped by compulsory computations. While $\phi\left(t_s, t_w\right) \log_2 P$ represents of the global communication time when computing the inner product, it increasing in nonlinearly way ($O\left(\log_2 P\right)$) with the increasing of the number of processor and thus becomes the bottleneck of the solver. It should be noted that even fine grid parallelism are fully explored on each node with multi-core processors, the term $\log_2(p)$ can only be improved to $\log_m(p)$, which doesn't not essentially change the communication complexity for global communication.

The paper focuses on reconstructed Krylov methods for heterogenous supercomputers by minimizing communications. Examples are presented to show how to use the *residual split*, *loop shift* and transpose sift techniques to reducing 3 synchronization points in GPBiCG(m, ℓ) into only one in *PGPBiCG*(m, l). The performance of the two methods are carefully analyzed and compared. The analysis and numerical experiments shows the reconstructed methods are more scalable on distribute supercomputers. Similar techniques can be applied to other Krylov methods [18]. Because the techniques here are focused on to improve the algorithms themselves, therefore the reconstructed methods are potable for all kinds of distributed supercomputers.

We also analysis why the indirect inner product computation sometimes results in poor convergence. And improved the convergence by eliminating the recursive computation of inner products. It should be mentioned that accurate compute the inner product for large dense vectors itself is a challenging problems. Therefore, if inner product computation can not be avoided in current Krylov solvers, it is very likely to be the most important kernel for distribute heterogenous supercomputers.

Acknowledgments

The first author would like to thank Dr. Andy Wathen for his suggestions, remarks and encouragement.

References

[1] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Anal. Appl.*, 32(3):866–901, 2011.

[2] H.M. Bücker and M. Sauren. A parallel version of the unsymmetric lanczos algorithm and its application to QMR. 1996. www.fz-juelich.de

[3] A.Castaldo and R. Whaley and A. Chronopoulos. Reducing Floating Point Error in DOT product using the Superblock Family of Algorithms. *SIAM. J. Sci. Comput.* 31(2),1156-1174.

[4] A.T. Chronopoulos and C.W. Gear, s-step iterative methods for symmetric linear systems. *J. Comput. Appl. Math*, 25(2):153-168,1989.

[5] A.T. Chronopoulos and C.W. Gear, On the efficient implementation of preconditioned s-step iterative methods on multiprocessors with memory hierachy. *Parallel Computing*, 11(1):37-53,1989.

[6] E. de Sturler. Iterative methods on distributed memory computers. *Ph. D. Thesis, Delft University of Technology, Delft, Netherlands*, 1994.

[7] E. de Sturler and H. van der Vorst. Communication cost reduction for krylov methods on parallel computers. In *High-Performance Computing and Networking*, pages 190–195. Springer, 1994.

[8] E. de Sturler and H.A. van der Vorst. Reducing the effect of global communication in GMRES(m) and CG on parallel distributed memory computers. *Applied Numerical Mathematics*, 18(4): 441–459, 1995.

[9] J. Demmel and Y. Hida. Accurate and Efficient Floating Point Summation *SIAM J. Sci. Comput.* (25)4, 1212-1248.

[10] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[11] S.L. Graham, and M. Snir, and C.A. Patterson, Getting up to speed: The future of supercomputing. *Committee on the Future of Supercomputing, National Research Council*, 2004

[12] A. Grama, and A. Gupta, and G. Karypis, and V. Kumar, Introduction to Parallel Computing Pearson Education Limited, 2003

[13] Seiji Fujino. GPBiCG(m, ℓ): a hybrid of BiCGSTAB and GPBiCG methods with efficiency and robustness. *Appl. Numer. Math.*, 41(1): 107–117, 2002. Developments and trends in iterative methods for large systems of equations—in memoriam Rüdiger Weiss (Lausanne, 2000).

[14] Tong-Xiang Gu, Xian-Yu Zuo, Xing-Ping Liu, and Pei-Lu Li. An improved parallel hybrid bi-conjugate gradient method suitable for distributed parallel computing. *J. Comput. Appl. Math.*, 226(1): 55–65, 2009.

[15] Tong-Xiang Gu, Xian-Yu Zuo, Li-Tao Zhang, Wan-Qin Zhang, and Zhi-Qiang Sheng. An improved bi-conjugate residual algorithm suitable for distributed parallel computing. *Appl. Math. Comput.*, 186(2): 1243–1253, 2007.

[16] Tong-Xiang Gu, Xing-Ping Liu, Ze-Yao Mo, and Xue-Bin Chi. Multiple search direction conjugate gradient method. I. Methods and their propositions. *Int. J. Comput. Math.*, 81(9): 1133–1143, 2004.

[17] Tong-Xiang Gu, Xing-Ping Liu, Ze-Yao Mo, and Xue-Bin Chi. Multiple search direction conjugate gradient method. II. Theory and numerical experiments. *Int. J. Comput. Math.*, 81(10): 1289–1307, 2004.

[18] Tong-xiang Gu, and Xing-Ping Liu, and Sheng-Xin Zhu. A collection of communication reduced Krylov iterative methods suitable for distributed supercomputers *in prepare*

[19] M.H. Gutknecht. Variants of BICGSTAB for matrices with complex spectrum. *SIAM J. Sci. Comput.*, 14(5): 1020–1033, 1993.

[20] N. Higham. Accuracy and Stability of Numerical Algorithms. SIAM,1996

[21] E.J. Im, K. Yelick, and R. Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1): 135–158, 2004.

[22] Xing-Ping Liu, Tong-Xiang Gu, Xu-Deng Hang, and Zhi-Qiang Sheng. A parallel version of QMRCGSTAB method for large linear systems in distributed parallel environments. *Appl. Math. Comput.*, 172(2): 744–752, 2006.

[23] G. Meurant. The block preconditioned conjugate gradient method on vector computers. *BIT*, 24(4): 623–633, 1984.

[24] G. Meurant. Numerical experiments for the preconditioned conjugate gradient method on the cray X-MP 2'. *Lawrence Berkeley Laboratory LBL-18023*, 1984.

[25] G. Meurant. The conjugate gradient method on vector and parallel supercomputers. Technical report, Technical Report CTAC-89, University of Brisbane, 1989.

[26] M. Mohiyuddin, M. Hoemmen, J. Demmel, and K. Yelick. Minimizing communication in sparse matrix solvers. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 36. ACM, 2009.

[27] R. Nishtala and K.A. Yelick. Optimizing collective communication on multicores. In *Proceedings of the First USENIX conference on Hot Topics in Parallelism*, pages 18–18. USENIX Association, 2009.

[28] T. Ogita, and S. Rump, and S. Oishi Accurate Sum and Dot Product. *SIAM J. Sci. Comput.* 26(6), 1955-1988, 2005.

[29] D.A.Patterson. Latency lags bandwith. *Communications of the ACM*, 47(10): 71–75, 2004.

[30] S. Rump, and S. Ogita, and S. Oishi. Accurate Floating-Point Summation Part I: Faithful Rounding *SIAM J. Sci. Comput.* 31(1),189–224, 2008.

[31] G.L.G. Sleijpen and H.A. van der Vorst. Hybrid bi-conjugate gradient methods for CFD problems. *Computational fluid dynamics review*, pages 457–476, 1995.

[32] H.A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM Journal on scientific and Statistical Computing*, 13: 631, 1992.

[33] H.A. van der Vorst. *Iterative Krylov methods for large linear systems*, volume 13 of *Cambridge Monographs on Applied and Computational Mathematics*. Cambridge University Press, Cambridge, 2003.

[34] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick and J. Demmel, Optimization of Sparse Matrix-Vector Multiplication on Emerging Multicore Platforms, *SC07*, Nov. 10-16, 2007, Reno, Nevada, USA.

[35] Shu-Fang Xu, Li Gao, and Ping-Wen Zhang. *Numerical Linear Algebra*. Peking University Press, Beijing, 2000, (in Chinese).

[36] L.T. Yang. The improved CGS method for large and sparse linear systems on bulk synchronous parallel architectures. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 232–237. IEEE, 2002.

[37] L.T. Yang and R.P. Brent. The improved BiCGSTAB method for large and sparse unsymmetric linear systems on parallel distributed memory

architectures. In *Algorithms and Architectures for Parallel Processing, 2002. Proceedings. Fifth International Conference on*, pages 324–328. IEEE, 2002.

[38] T.R. Yang and H.X. Lin. The improved quasi-minimal residual method on massively distributed memory computers. In *High-Performance Computing and Networking*, pages 389–399. Springer, 1997.

[39] Lin-Bo Zhang, Xue-Bin Chi, Ze-Yao Mo, and Ruo Li. *Introduction to Parallel Computing*. Tsinghua University Press, Beijing, 2006, (in Chiness).

[40] S.L. Zhang. GPBi-CG: Generalized product-type methods based on Bi-CG for solving nonsymmetric linear systems. *SIAM Journal on Scientific Computing*, 18:537, 1997.

[41] S. Zhu Parellel GPBiCG(m, ℓ) methods and proconditioning techneques. *Master Thesis, Graduate School, China Academy of Engineering Physics*, 2010

[42] Xian-Yu Zuo, Tong-Xiang Gu, and Ze-Yao Mo. An improved GPBi-CG algorithm suitable for distributed parallel computing. *Applied Mathematics and Computation*, 215(12): 4101–4109, 2010.

RECENT REPORTS

12/58	Multiscale reaction-diffusion algorithms: pde-assisted Brownian dynamics	Franz Flegg Chapman Erban
12/59	Numerical simulation of shear and the Poynting effects by the finite element method: An application of the generalised empirical inequalities in non-linear elasticity	Mihai Goriely
12/60	From Brownian dynamics to Markov chain: an ion channel example	Chen Erban Chapman
12/61	Three-dimensional coating and rimming flow: a ring of fluid on a rotating horizontal cylinder	Leslie Wilson Duffy
12/62	A two-pressure model for slightly compressible single phase flow in bi-structured porous media	Soulaine Davit Quintard
12/63	Mathematical modelling plant signalling networks	Muraro Byrne King Bennett
12/64	A model for one-dimensional morphoelasticity and its application to fibroblast-populated collagen lattices	Menon Hall McCue McElwain
12/65	Effective order strong stability preserving RungeKutta methods	Hadjimichael Macdonald Ketcheson Verner
12/66	Morphoelastic Rods Part I: A Single Growing Elastic Rod	Moulton Lessinnes Goriely
12/67	Wrinkling in the deflation of elastic bubbles	Aumaitre Knoche Cicuta Vella
12/68	Indentation of ellipsoidal and cylindrical elastic shells	Vella Ajdari Vaziri Boudaoud
12/69	Memory of Recessions	Cross McNamara Pokrovskiif
12/70	An estimate of energy dissipation due to soil-moisture hysteresis	McNamara
12/71	The Mathematics Behind Sherlock Holmes: A Game of Shadows	Goriely Moulton

12/74	Static and dynamic stability results for a class of three-dimensional configurations of Kirchhoff elastic rods	Majumdar Goriely
12/75	Error estimation and adaptivity for incompressible, nonlinear (hyper)elasticity	Whiteley Tavener
12/76	A note on heat and mass transfer from a sphere in Stokes flow at low Péclet number	Bell Byrne Whiteley Waters
12/77	Effect of disjoining pressure in a thin film equation with non-uniform forcing	Moulton Lega
12/78	A Review of Mathematical Models for the Formation of Vascular Networks	Scianna Bell Preziosi
12/79	Fast and Accurate Computation of Gauss-Legendre and Gauss-Jacobi Quadrature Nodes and Weights	Hale Townsend
12/80	On the spectral distribution of kernel matrices related to radial basis functions	Wathen Zhu

Copies of these, and any other OCCAM reports can be obtained from:

Oxford Centre for Collaborative Applied Mathematics
Mathematical Institute
24 - 29 St Giles'
Oxford
OX1 3LB
England
www.maths.ox.ac.uk/occam