Экспертная методология анализа производительности взаимодействия процессов в MPI приложениях

А.В. Дергунов[1]

Нижегородский государственный университет им. Н.И. Лобачевского[1]

Для анализа MPI программ часто используют программные системы, которые осуществляют сбор и визуализацию трассы выполнения программы на кластере. Но при использовании таких инструментов пользователь сталкивается с проблемой анализа больших объемов информации. Поэтому возникает потребность в средствах для автоматизации анализа трассы, которые подсказали бы пользователю, как повысить производительность его программы. Задача повышения производительности является достаточно сложной и вряд ли может быть решена формальными методами. В данной работе предлагается экспертная методология решения поставленной задачи.

1. Введение

Для анализа MPI программ наиболее часто используют программные системы, которые осуществляют сбор и визуализацию трассы выполнения программы. Примером такого средства является Jumpshot [1]. На рис. 1 показано окно временной диаграммы системы Jumpshot, показывающее трассу работы MPI программы, реализующей сеточные вычисления при работе на 32 процессорах в течение 9 секунд (см. описание этой программы далее в разделе «Эксперимент по повышению производительности одной MPI программы»).

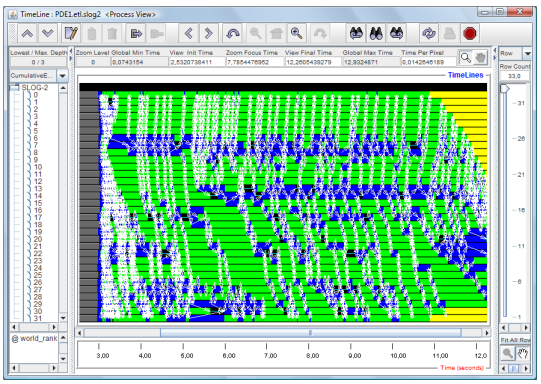


Рис. 1. Окно временной диаграммы системы Jumpshot для трассы MPI программы, реализующей сеточные вычисления при работе на 32 процессорах в течение 9 секунд

Но при использовании таких инструментов пользователь сталкивается с проблемой анализа больших объемов информации. Пользователь должен использовать инструменты зуммирования и фильтрации, чтобы исследовать события трассы. Эта проблема особенно актуальная при анализе трассы с большим количеством процессов, собранных на больших суперкомпьютерах.

Другой проблемой при использовании средств визуализации трассы является то, что часто встречающиеся ситуации, приводящие к потерям производительности MPI программ, явно не визуализируются. Таким образом, пользователь должен опираться на свой опыт работы с MPI и детально исследовать трассу программы для выявления причин недостаточной производительности.

Одной из причин недостаточной производительности является плохая синхронизация приемов и передач данных в MPI программе. В результате процедура приема сообщения может простаивать, дожидаясь посылки сообщения (см. рис. 2). Для решения этой проблемы нужно обеспечить, чтобы сообщения посылались как можно раньше, и т.о. повысить вероятность того, что сообщение прибудет до момента, когда оно потребуется другому процессу.

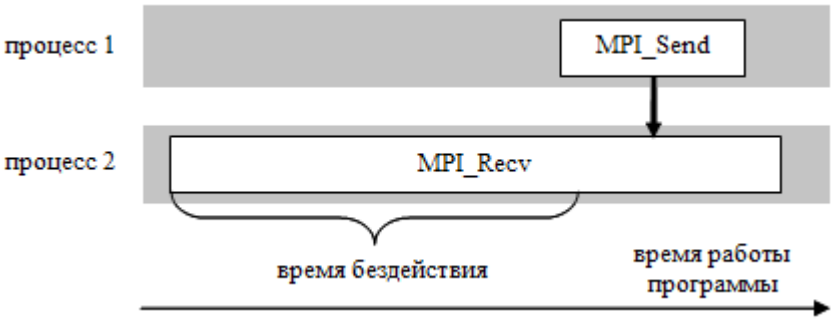


Рис. 2. Поздняя посылка сообщения

Но даже если пользователь обнаружил причину падения производительности, то оказывается невозможным оценить, насколько она влияет на общее время работы программы. Например, чтобы оценить, в какой степени поздняя посылка данных замедлила работу программу, не достаточно просто обратиться к суммарному времени, затраченному на вызов процедур MPI_Recv, т. к. простой при ожидании данных составляет лишь долю времени работы этой процедуры. Поэтому нет возможности оценить, какой выигрыш производительности получит пользователь, если соответствующим образом изменит свою программу.

В этой работе рассматривается созданная автором система Performance Expert, основанная на экспертной методологии, которая позволяет автоматизировать анализ трасс MPI программ.

2. Модели представления знаний для анализа производительности взаимодействия процессов в MPI приложениях

2.1 Модель MPI приложения

MPI приложение рассматривается как множество взаимодействующих процессов $PR = \{PR_1, PR_2, ...PR_N\}$. Взаимодействие осуществляется с помощью действий, инициируемых процессами в определенной последовательности:

$$PR_i \Rightarrow A_i = \langle a_{ij} \rangle; i = 1,2,...,N; j = 1,2,...,M_i$$

Каждое действие процесса состоит в вызове функций библиотеки MPI $F = \{f_k\}, k = 1,2,...,K$. Функция характеризуется вектором входных и выходных аргументов:

$$f_k \Rightarrow FA_k^{IN} = (fa_{k1}^{IN},...,fa_{kL_k^{IN}}^{IN}); k = 1,...,K$$

$$f_k \Rightarrow FA_k^{OUT} = (fa_{k1}^{OUT},...,fa_{kL_k^{OUT}}^{OUT}); k = 1,...,K$$

Таким образом, каждое *действие процессов* характеризуется функцией библиотеки MPI и значениями ее входных и выходных аргументов:

$$a_{ij} = \langle f_k, FAval_k^{IN}, FAval_k^{OUT} \rangle; i = 1,2,...,N; j = 1,2,...,M_i; f_k \in F$$

$$FAval_k^{IN} = \left(av_{k1}^{IN},...,av_{kL_k^{IN}}^{IN}\right); FAval_k^{OUT} = \left(av_{k1}^{OUT},...,av_{kL_k^{OUT}}^{OUT}\right);$$

Существует несколько моделей обмена сообщениями, которые определяются синтаксисом и семантикой функций F.

2.2 Повышение производительности MPI приложения

Каждому действию a_{ij} процесса соответствует время начала его выполнения t_{ij}^a и длительность d_{ij}^a. *Потеря производительности взаимодействия процессов* определяется как сумма длительности выполнения действий всех процессов:

$$D^a = \sum_{i=1}^{N} \sum_{j=1}^{M_i} d_{ij}^a$$

Длительность выполнения действий состоит из следующих компонентов:

$$d_{ij}^{a} = (d_{ij}^{a})_{prep} + (d_{ij}^{a})_{wait} + (d_{ij}^{a})_{comm}$$

где:

$(d_{ij}^{a})_{prep}$ – длительность подготовки к передаче/приему сообщений;

$(d_{ij}^{a})_{wait}$ – длительность ожидания готовности других процессов к передаче/приему сообщений;

$(d_{ij}^{a})_{comm}$ – длительность передачи/приема сообщений по сети.

Способами сокращения длительности выполнений действий являются:

- Сокращение длительности $(d_{ij}^{a})_{prep}$: группировка отдельных сообщений в одно большое для сокращения подготовки данных для отправки.
- Сокращение времени $(d_{ij}^{a})_{wait}$: улучшение синхронизации приемов/передач сообщений, балансировка нагрузки процессов, совмещение обменов сообщениями и вычислений.
- Сокращение времени $(d_{ij}^{a})_{comm}$: сокращение количества пересылаемых сообщений, учет топологии сети при передаче сообщений.

Приведенные способы сокращения времен выполнения действий можно представить в виде *рекомендаций* $REC = \{rec_i\}, i = 1,2,...,R$. Повышение производительности взаимодействия процессов определяется как:

$$D^{a} \Rightarrow \min_{REC}$$

Сформулированная задача повышения производительности является достаточно сложной и вряд ли может быть решена формальными методами. Потому в работе предлагается экспертная методология решения поставленной задачи, включающая выполнение следующих основных этапов:

1. Выявление и систематизация проблем производительности – типовых (повторяющихся) фактов потери производительности.
2. Установление причин потери производительности для выявленных проблем.
3. Разработка рекомендаций по устранению этих причин.
4. Описание выявленных проблем производительности, правил установления причин их возникновения и рекомендаций по их устранению.

Первые три этапа выполняются экспертом на основе своего опыта повышения производительности и знаний о принципах работы библиотеки MPI. Четвертый – с помощью разработанной системы.

2.3 Модель проблемы производительности

В работе применяется следующая модель *проблемы производительности* MPI приложения:

$$pb = \langle pd, dur, TRRULES, ANRULES, REC(A^{INFO}) \rangle$$

где:

pd – вербальное описание проблемы;

dur – длительность проявления данной проблемы;

$TRRULES$ – правила трассировки действий проблемы;

$ANRULES$ – правила распознавания проблемы;

REC – рекомендации по ее устранению;

$AINFO = \{\langle f_i, t_i, d_i, pr_i, cs_i \rangle\}$ – описание действий, приведших к проблеме (где f_i – функция действия, t_i – время вызова функции, d_i – длительность работы функции, pr_i – процесс, который инициировал вызов функции, cs_i – место в исходном коде, из которого осуществлялся вызов).

2.4 Схема работы системы

Схема работы системы Performance Expert представлена на рис. 3 и состоит из двух этапов: подготовительного (этапа обучения) и этапа применения.

Подготовительный этап выполняется экспертом после выявления очередной проблемы производительности. На этом этапе:

1. Эксперт описывает правила трассировки действий процессов, которые (или сочетания которых) могут привести к возникновению выявленной проблемы.
2. Эксперт описывает правила распознавания выявленной проблемы.

На этапе применения:

1. Трассировщик запускается вместе с MPI приложением и формирует трассы, в которой в виде последовательности событий регистрируются все необходимые для последующего анализа действия процессов приложения.
2. После выполнения приложения анализатор считывает события трассы и выполняет правила распознавания проблемы. Результатом анализа является список проблем и рекомендаций.
3. При необходимости пользователь вносит изменения в MPI приложение и повторяет этап применения.

Рис. 3. Схема работы системы

2.5 Трассировка выполнения MPI приложения

Обучение трассировщика выполняется на основе правил трассировки действий проблемы производительности, выполнение которых может привести к ее возникновению. Результатом работы трассировщика должна быть трасса работы процесса приложения, в которой зарегистрированы простые события, вызванные действиями процессов.

Простое событие представляется как:

$$e = \langle f, et, EPval, t, d, pr, cs \rangle$$

где:

f – функция действия;

et – тип события, который задает набор параметров события $EP = (ep_1,...,ep_K)$;

$EPval = (v_1,...,v_K)$ – значения параметров события;

t – время наступления события;

d – длительность события;

pr – процесс, на котором произошло событие;

cs – место в исходном коде, из которого осуществлялся вызов функции.

Правило трассировки представляется как:

$$a = \langle f, FAval^{IN}, FAval^{OUT} \rangle \xrightarrow{trrule} e = \langle f, et, EPval, t, d, pr, cs \rangle$$

$$trrule = \langle f, et, EPtemp \rangle$$

где:

$f \in F$ – функция, при выполнении которой трассировщиком создается событие, определяемое правилом;

et – тип события, задаваемый правилом трассировки;

$EPtemp = \{\langle ep_i, Expr_i, kind_i \rangle\}; i = 1,...,N$ – описание параметров события и способа получения их значений, где:

ep_i – параметр события;

$Expr_i : FAval^{IN} \times FAval^{OUT} \to v_i$ – функция для вычисления значения v_i параметра на основе аргументов вызванной функции;

$kind_i \in \{in, out\}$ – характер параметра (*in* – входной, т.е. вычисляемый только на основе входных аргументов функции; *out* – выходной, т.е. формируемый на основе выходных аргументов функции или входных и выходных).

Значения параметров t, d, pr и cs события формируются трассировщиком.

В рамках системы разработан язык для описания правил трассировки. На основе правил генерируются функции-обертки для трассируемых функций. Трассировщик основан на принципе динамического инструментирования [2]. При работе программы он перехватывает вызовы трассируемых функций и вместо них вызывает функции-обертки. Функции-обертки регистрируют соответствующее событие в трассе и вызывают оригинальную функцию.

2.6 Анализ событий трассы MPI приложения

После выполнения трассировки выполняется анализ полученной трассы на предмет выявления проблем производительности. Исходными данными для анализа является трасса MPI приложения, сформированная в результате трассировки и содержащая простые события, а результатом анализа – множество выявленных проблем производительности. Анализ осуществляется на основе правил распознавания проблем производительности *ANRULES* и осуществляется в два этапа:

1. Конструирование составных событий как претендентов на отдельные типы проблем производительности.
2. Выявление проблем производительности из множества сконструированных составных событий.

Составное событие представляется как:

$$ce = \langle cet, CEPval, ME \rangle$$

где:

cet – тип составного события, который задает набор параметров события $CEP = (cep_1,...,cep_K)$;

$CEPval = (cev_1,...,cev_K)$ – значения параметров составного события;

$ME = \{e_i\} \cup \{ce_j\}; i = 1,...,N; j = 1,...,M$ – множество простых и составных событий-членов.

Правило конструирования составного события представляется как:

$$\left.\begin{array}{l} E \xrightarrow{ET} E^R \\ CE \xrightarrow{CET} CE^R \end{array}\right\} \xrightarrow{cerule} ce = \langle cet, CEPval, ME \rangle$$

$$cerule = \langle ET, CET, \sigma, cet, CEPtemp, ET^S \rangle$$

где:

$ET = \{et_i\}; i = 1,...,N$ – множество типов простых событий для выборки из множества простых событий трассы E подмножества событий $E^R = \{\langle f_i, et_i, EPval_i, t_i, d_i, pr_i, cs_i \rangle\} \subseteq E; i = 1,...,N$, релевантных анализируемой проблеме производительности;

$CET = \{cet_j\}; j = 1,...,M$ – множество типов составных событий для выборки из множества составных событий CE подмножества релевантных событий $CE^R = \{\langle cet_j, CEPval_j, ME_j \rangle\} \subseteq CE; j = 1,...,M$ указанных типов;

Введем обозначения:

$$EPV_i = \langle f_i, EPval_i, t_i, d_i, pr_i, cs_i \rangle; i = 1,...,N$$

$$P = EPV_1 \times ... \times EPV_N \times CEPval_1 \times ... \times CEPval_M$$

$\sigma : P \to \{0,1\}$ – условие конструирования составного события (если значение функции $\sigma = 1$, то составное событие может быть сконструировано для множества релевантных событий E^R и CE^R и нет в противном случае);

cet – тип составного события, конструируемый правилом;

$CEPtemp = \{\langle cep_k, Expr_k \rangle\}; k = 1,...,K$ – описание параметров конструируемого события и способов получения их значений, где:

cep_k – параметр составного события;

$Expr_k : P \to cev_k$ – функция для вычисления значения cev_i параметра составного события;

При выполнении условий правила (т.е. существовании подмножества релевантных событий E^R и CE^R и выполнении условия σ) конструируется событие ce указанного типа cet, где:

$CEPval = (cev_1,...,cev_K)$ – значения параметров событий, вычисляемые с помощью описания параметров $CEPtemp$;

$ME = E^R \cup CE^R$ – множество событий-членов;

Параметр правила $ET^S \subseteq ET$ указывает типы событий, которые являются общими для конструируемого события и других составных событий. Остальные простые события, релевантные данному правилу, принадлежат только конструируемому составному событию.

Правила конструирования составных событий задаются в виде последовательности деклараций, имеющих синтаксис:

```
declare composite event <имя типа составного события>
  member(
    [<shared>] <тип события-члена> as <имя события>,
    ...
  )
  when
    <логическое выражение>
  parameters(
    <параметр события> = <выражение для вычисления значения>,
```

```
    ...
);
```

Рис. 4. Синтаксис правил конструирования составных событий

где:

`declare composite event` описание правила конструирования составного события типа cet;

`member` список типов простых ET и составных CET событий для выборки подмножества релевантных событий; флаг *shared* указывается подмножества типов $ET^S \subseteq ET$; имя события используется для ссылки на его параметры в логическом выражении и выражениях для вычисления значений параметров события;

`when` условие конструирования составного события σ;

`parameters` описание параметров конструируемого события и способов получения их значений $CEPtemp$ (с указанием параметров cep_k и выражений для вычисления их значений $Expr_k$);

Правило выявления проблемы производительности представляется как:

$$ce = \langle cet, CEPval, ME \rangle \xrightarrow{pbrule} pb = \langle pd, dur, REC(A^{INFO}) \rangle$$

$$pbrule = \langle cet, \varphi, pd, RECtemp, L_{dur} \rangle$$

где:

cet – тип составного события как претендента на определяемую проблему производительности;

$\varphi : CEPval \to \{1,0\}$ – условие возникновения проблемы (если значение функции $\varphi = 1$, то проблема существует и нет в противном случае);

pd – вербальное описание проблемы, определяемой правилом;

$RECtemp : CEPval \to REC$ – шаблон рекомендации по устранению проблемы;

$L_{dur} : CEPval \to dur$ – функция вычисления длительности проблемы производительности;

При выполнении условий правила (т.е. выполнении для составного события ce типа cet условия φ) делается вывод о наличии проблемы производительности pb. Описание действий приложения, приведших к этой проблеме, формируется на основе функции интерпретатора $Frec$, т.е. $A^{INFO} = Frec(ME)$. Функция $Frec$ осуществляет рекурсивное извлечение простых событий из ME и формирования на их основе информации о множестве действий.

Правила выявления проблемы производительности задаются в виде последовательности деклараций, имеющих синтаксис:

```
declare problem for <имя типа составного события>
  when
    <логическое выражение>
  parameters(
    name = "<название проблемы>",
    description = "<описание проблемы>",
    advice = <шаблон рекомендации>,
    duration = <выражение для вычисления длительности>);
```

Рис. 5. Синтаксис правил выявления проблемы производительности

где:

`declare problem for` описание правила выявления проблемы производительности для составного события-претендента типа cet;

`when` условие возникновения проблемы φ;

`name` название проблемы;

`description` вербальное описание проблемы pd;

`advice` шаблон рекомендации $RECtemp$;

`duration` функция вычисления длительности L_{dur};

3. Состав базы знаний проблем производительности MPI приложений

В состав базы знаний системы Performance Expert были включены описания следующих типовых проблем производительности MPI приложений:

- Проблемы двухточечных операций обмена:
  - поздняя посылка сообщения;
  - поздний прием сообщения;
- Проблемы коллективных операций обмена:
  - задержка перед барьерной синхронизацией;
  - ранний прием данных при операции «от многих к одному»;
  - поздняя посылка данных при операции «от одного ко многим»;
  - задержка перед операцией «от многих ко многим»;
- Проблемы операций удаленного доступа к памяти:
  - задержка при создании «окна» для удаленного доступа к памяти;
  - конкуренция за блокировку «окна» для удаленного доступа к памяти;
  - позднее начало периода предоставления доступа к «окну»;
  - раннее завершение периода предоставления доступа к «окну».

На рис. 6 продемонстрировано правило для выявления поздней посылки сообщения, о которой говорилось в введении (рис. 2). Это правило срабатывает для составного события двухточечного обмена (типа point_to_point), для которого выполнено условие, указанное в when. При выполнении правила анализатор делает вывод об указанной проблеме производительности.

```
declare problem for point_to_point
  when
    send_start_time > recv_wait_start_time
  parameters(
    name = "Поздняя посылка сообщения",
    description = "Функция посылки сообщения вызывается
      позднее функции приема. Из-за этого блокирующая
      функция приема вынуждена простаивать.",
    advice = "Улучшить синхронизацию приемов и передач
      сообщений, оправляя сообщения по
      возможности раньше, или
      использовать неблокирующие функции приема.",
    duration = send_start_time - recv_wait_start_time);
```

Рис. 6. Правило для выявления поздней посылки сообщения

4. Эксперимент по повышению производительности одного MPI приложения

Для проверки разработанной системы было осуществлено повышение производительности MPI программы, реализующей сеточные вычисления, а именно метод итераций Якоби для численного решения задачи Дирихле для уравнения Лапласа [4]. Уравнение Лапласа представлено ниже:

$$\frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} = 0$$

В задаче заданы значения на граничных точках двумерной сетки. Необходимо вычислить значения во внутренних точках. Метод итераций Якоби [4] предусматривает несколько итераций, в которых новые значения в точках вычисляются как среднее из значений четырех ее соседних точек, вычисленных на предыдущей итерации:

```
for (int iter = 0; iter < iterations; iter++)
{
  for (int i = 0; i < height; i++)
    for (int j = 0; j < width; j++)
      new[i][j] = (grid[i-1][j] + grid[i][j-1] +
                   grid[i+1][j] + grid[i][j+1]) / 4;
  copy(grid, new);
}
```

Рис. 7. Метод итераций Якоби

Для реализации метода итераций Якоби на компьютере с распределен ной памятью удобно избавиться от операции копирования матрицы new в grid, продублировав предыдущий цикл и поменяв в нем new и grid местами, а затем каждому процессу необходимо назначить свою прямоугольную полосу. Для вычисления значений на краях нужно использовать данные, размещенные на других процессорах. Поэтому необходимо использовать соответствующие операции обмена MPI. Таким образом, каждый процесс выполняет следующую последовательность действий:

1. вычислить значения во внутренних точках своей полосы;
2. отправить соседям вычисленные значения на краях своей полосы;
3. получить от соседей значения на краях их полос.

В рамках эксперимента MPI программа, реализующая данный алгоритм, была запущена на кластере с 64 процессами и собрана трасса ее выполнения. В результате анализа производительности MPI программы, реализующей описанный алгоритм, с помощью системы Performance Expert были выявлены две проблемы производительности, по которым получены следующие рекомендации по их устранению:

```
Название проблемы: Поздняя посылка сообщения
Описание проблемы: Функция посылки сообщения вызывается позднее
функции приема. Из-за этого блокирующая функция приема вынужде-
на простаивать.
Длительность: 57,96%
Совет: Улучшить синхронизацию приемов и передач сообщений, оп-
равляя сообщения по возможности раньше, или использовать небло-
кирующие функции приема.
Действия:
•     вызов MPI_Send из calculate
•     вызов MPI_Recv из calculate

Название проблемы: Поздний прием сообщения
Описание проблемы: Функция приема сообщения вызывается позднее
функции посылки. Из-за этого блокирующая функция посылки вынуж-
дена простаивать, т.к. ожидает начало приема.
Длительность: 2,83%
Совет: Улучшить синхронизацию приемов и передач сообщений или
использовать неблокирующие функции приема.
Действия:
•     вызов MPI_Send из calculate
•     вызов MPI_Recv из calculate
```

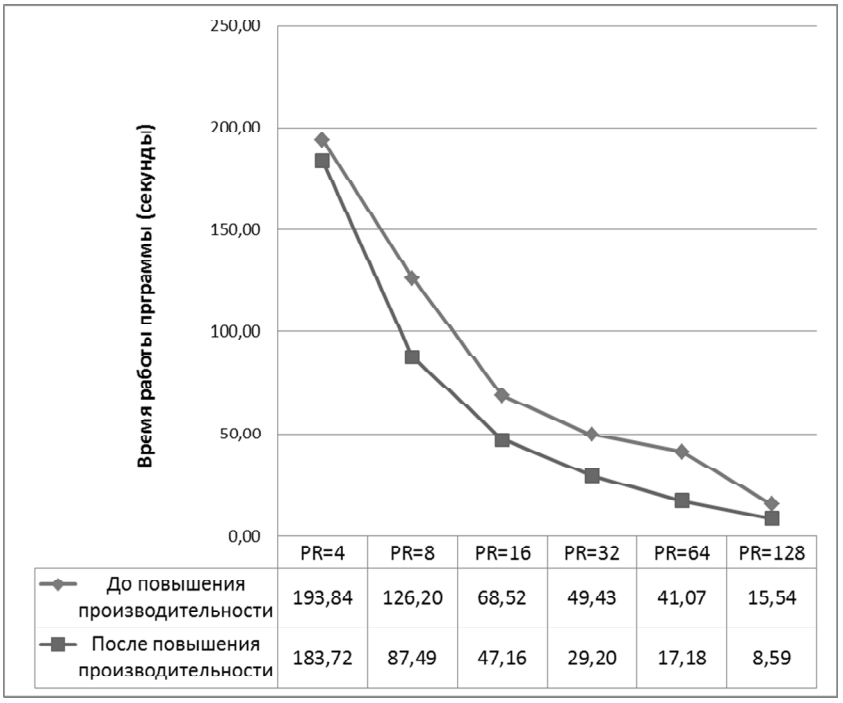
Рис. 8. Рекомендации по устранению проблем производительности

Наибольшую длительность составляет проблема поздней посылки сообщения, поэтому было принято решение о ее устранении. Одним из предложенных советов по устранению этой проблемы было «улучшить синхронизацию приемов и передач данных, отправляя данные по возможности раньше». В этом случае между отправкой и получением сообщений будут выполняться локальные вычисления. Принимая во внимание предложенный совет, исходная программа была изменена так, чтобы каждый процесс выполнял следующую последовательность действий:

1. отправить соседям значения на краях своей полосы;
2. вычислить значения во внутренних точках своей полосы;

3. получить от соседей значения на краях их полос;
4. вычислить значения на краях своей полосы.

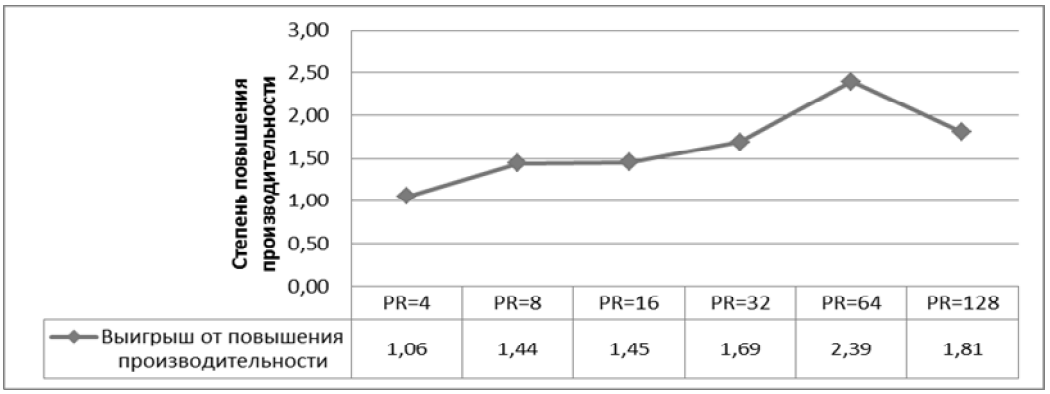
На рис. 9 представлено сравнение времени работы двух версий программы при использовании различного количества процессов *PR*. В расчетах использовалась сетка размером 6400x6400, было выполнено 1000 итераций. Для каждого случая время работы программы улучшилось.



	PR=4	PR=8	PR=16	PR=32	PR=64	PR=128
До повышения производительности	193,84	126,20	68,52	49,43	41,07	15,54
После повышения производительности	183,72	87,49	47,16	29,20	17,18	8,59

Рис. 9. Результат повышения производительности MPI программы

На рис. 10 представлен выигрыш от повышения производительности программы. Наибольший выигрыш достигнут при использовании 64 процессов: время работы программы сократилось в 2,39 раза.



	PR=4	PR=8	PR=16	PR=32	PR=64	PR=128
Выигрыш от повышения производительности	1,06	1,44	1,45	1,69	2,39	1,81

Рис. 10. Выигрыш от повышения производительности MPI программы

Измерения производительности проводились на кластере Нижегородского государственного университета им. Н.И. Лобачевского. Параметры кластера следующие: узлы с двумя двухъядерными процессорами Intel Xeon 5150 2.66 GHz, 4 GB оперативной памяти, сеть Gigabit Ethernet, ОС Windows Server 2008 x64. Запуски программ производились с разным количеством узлов.

5. Заключение

Рассмотрена программная система, которая значительно облегчает задачу анализа производительности взаимодействия процессов в MPI приложений. Для выполнения анализа производительности предложена экспертная методология, базирующаяся на модели проблемы производительности MPI приложения как совокупности действий MPI процессов, определенные сочетания которых при определенных условиях могут привести к возникновению идентифицируемой проблемы производительности, а также на модели правил трассировки действий проблемы и модели правил распознавания проблемы и соответствующих языках.

Разработанная система основана на ином подходе к анализу производительности по сравнению системами для визуализации трассы работы MPI программы (например, Jumpshot [1]). Наиболее близким аналогом является система KOJAK [5]. Но в отличии от разработанной системы Performance Expert, в системе KOJAK описания проблем производительности и способов их устранения фиксированы. В результате эксперту для описания проблем производительности необходимо изменять исходный код системы KOJAK.

В работе приведен список типовых проблем производительности MPI приложений, описанных с использованием разработанной системы. Описан эксперимент по повышению производительности MPI программы с использованием системы Performance Expert. Использование полученных рекомендаций позволило в среднем повысить производительность в 1,64 раза. Максимальное повышение производительности (в 2,4 раза) было достигнуто при 64 процессах.

Литература

1. Toward Scalable Performance Visualization with Jumpshot / O. Zaki [et al.] // High-Performance Computing Applications. – 1999. – Vol. 13. – No. 2. – P. 277-288.
2. Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation / C.-K. Luk [et al.] // Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation, Chicago, IL, 2005. – P. 190-200.
3. Forgy, C. L. Rete: A Fast Algorithm for the Many Pattern: Many Object Pattern Match Problem / C. L. Forgy // Artificial Intelligence. – 1982. – Vol. 19. – P. 17-37.
4. Эндрюс, Г. Р. Основы многопоточного, параллельного и распределенного программирования / Г. Р. Эндрюс; пер. с англ. А. С. Подосельника, Г. И. Сингаевской, А. Б. Ставровского. – М.: Вильямс, 2003. – 512 с.
5. Wolf, F. Automatic performance analysis of hybrid MPI/OpenMP applications / F. Wolf, B. Mohr // Journal of Systems Architecture: the EUROMICRO Journal. – 2003. – Vol. 49, № 10/11. – P. 421-439.